Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

EVOLVING AND CO-EVOLVING META-LEVEL REASONING STRATEGIES FOR MULTI-AGENT COLLABORATION

A thesis presented in partial fulfilment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science
at Massey University, Albany,
New Zealand.

Mona Abdulrahman M Alshehri

2024

Contents

Abstract xvi

Acknowledgements xvii

1 Introduction 1

1.1 Background . . . 1
1.2 Overview of the Problem Domain . . . 2
1.2.1 Tile World Problem . . . 3
1.2.2 Heavy Tile World Problem . . . 5
1.2.3 Heavy Tile in Heterogeneous System . . . 7
1.2.4 Prey and Predator Problem . . . 9
1.3 Challenges of Evolving Meta-Level Reasoning Strategies . . . 10
1.4 Research Objectives . . . 12
1.5 Significance of the Study . . . 12
1.6 Published work . . . 13
1.7 Scope and Limitations of Research . . . 13
1.8 Research Questions and Hypotheses . . . 14

2 Literature Review 16

2.1 Introduction . . . 16
2.2 (GA) Genetic Algorithms . . . 17
2.2.1 Introduction . . . 17
2.2.2 Chromosome structure . . . 18
2.2.3 Initialization of the First Population . . . 19
2.2.4 Evaluation and Selection . . . 19
2.2.5 Evolutionary Operations . . . 19
2.2.6 Summary . . . 20
2.3 (GP) Genetic Programming . . . 21
2.3.1 Introduction . . . 21
2.3.2 Chromosome structure . . . 21

2.3.3 Initialization of the First Population 21
2.3.4 Evaluation and Selection 22
2.3.5 Evolutionary Operations 22
2.3.6 Summary 22
2.4 (GNP) Genetic Network Programming 23
2.4.1 Introduction 23
2.4.2 Chromosome structure 24
2.4.3 Initialization of the First Population 24
2.4.4 Evaluation and Selection 24
2.4.5 Evolutionary Operations 28
2.4.6 Hyperparameters 28
2.4.7 Summary 30
2.5 GNP-RL 31
2.5.1 Introduction 31
2.5.2 Chromosome structure 31
2.5.3 Initialization of the First Population 32
2.5.4 Evaluation and Selection 32
2.5.5 Evolutionary Operations 34
2.5.6 Hyperparameters 35
2.5.7 Summary 36
2.6 DGNP-RL and VSGNP-RL 37
2.6.1 Introduction 37
2.6.2 Chromosome structure 38
2.6.3 Initialization of the First Population 38
2.6.4 Evaluation and Selection 38
2.6.5 Evolutionary Operations 39
2.6.6 Hyperparameters 42
2.6.7 Summary 42
2.7 GNP-RL-CC-OS-TP 43
2.7.1 Introduction 43
2.7.2 Chromosome structure 45
2.7.3 Initialization of the First Population 46
2.7.4 Evaluation and Selection 47
2.7.5 Evolutionary Operations 47
2.7.6 Hyperparameters 47
2.7.7 Summary 47
2.8 Diversity in Evolutionary Algorithms 50

3 Recent Variants of GNP **52**
3.1 Introduction . . . 52
3.2 GNP and VSGNP . . . 53
3.2.1 Tile World problem (Review) . . . 54
3.2.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed) . . . 57
3.2.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed) . . . 60
3.2.4 Prey and Predator Problem (Review) . . . 63
3.3 GNP-CC-OS-TP and VSGNP-CC-OS-TP . . . 65
3.3.1 Tile World problem (Review) . . . 66
3.3.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed) . . . 68
3.3.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed) . . . 72
3.3.4 Prey and Predator Problem (Proposed) . . . 74
3.4 GNP-RL and VSGNP-RL . . . 76
3.4.1 Tile World problem (Review) . . . 76
3.4.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed) . . . 79
3.4.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed) . . . 83
3.4.4 Prey and Predator Problem (Review) . . . 86
3.5 GNP-RL-CC-OP-TP and VSGNP-RL-CC-OS-TP . . . 89
3.5.1 Tile World problem (Review) . . . 89
3.5.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed) . . . 93
3.5.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed) . . . 97
3.5.4 Prey and Predator Problem (Proposed) . . . 101
3.6 Summary . . . 105

4 Using Private Conflict Kernels with GNP **106**
4.1 Motivation . . . 106
4.2 Related work . . . 106
4.3 Proposed Algorithms (Architecture) . . . 107
4.3.1 Using the Private-Conflict-Kernels on the Selection (PCK-GNP (S)) . . . 111

4.3.2 Using the Private-Conflict-Kernels on the Crossover (PCK-GNP (C)) . . . 112
4.3.3 Using the Private-Conflict-Kernels on the Mutation (PCK-GNP (M)) . . . 112
4.3.4 How to prevent losing diversity in the PCK-GNP . . . 113
4.4 Testing and Analysis . . . 113
4.4.1 Solving the Tile World Problem, PCK-GNP . . . 114
4.4.2 Solving the Heavy Tile World Problem, PCK-GNP with one sub-program . . . 118
4.4.3 Solving the Heavy Tile World Problem: PCK-VSGNP with two subprograms . . . 123
4.5 Summary . . . 126

5 Using Conflict Directed A* inside GNP Graph 127
5.1 Motivation . . . 127
5.2 Related work . . . 128
5.3 Proposed Algorithms (Architecture) . . . 135
5.3.1 A*-GNP . . . 136
5.3.2 Decision-Based A*-GNP (DBA*-GNP) . . . 144
5.3.3 CDA*-GNP . . . 147
5.4 Testing and Analysis . . . 163
5.4.1 Normal Tile World Problem Results . . . 164
5.4.2 Tile World Problem with Heavy Tile using one sub-program for the Graph . . . 167
5.4.3 Solving the Heavy Tile World Problem using two sub-programs for the Graph . . . 170
5.4.4 Prey and Predator Problem Results . . . 173
5.5 Summary . . . 176

6 Using CDA*-GNP in a Heterogeneous System 177
6.1 Motivation . . . 177
6.2 Related work . . . 178
6.3 Proposed Algorithms (Architecture) . . . 180
6.3.1 Random Heterogeneous system . . . 183
6.3.2 CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN) . . . 187
6.3.3 CDA*-GNP with Heterogeneous system to explore the Members (CDA*-GNP-HM) . . . 191
6.4 Testing and Analysis . . . 196

6.4.1 Fully Heterogeneous system 196
6.4.2 Partly Heterogeneous system 199
6.5 Summary 201

7 Summary and Conclusion 202

7.1 Introduction 202
7.2 The Algorithms Summary 202
7.2.1 GA 202
7.2.2 GP 202
7.2.3 GNP 203
7.2.4 GNP-RL 203
7.2.5 DGNP-RL and VSGNP-RL 204
7.2.6 GNP-RL-CC-OS-TP 204
7.2.7 GNP-CC-OS-TP 205
7.2.8 PCK-GNP 205
7.2.9 CDA*-GNP 209
7.2.10 CDA*-GNP-H 212
7.3 Managing Genetic Diversity in the Population 213
7.4 Best Chromosomes Extracted by the Proposed Algorithms 215
7.5 Total Computational Costs of the Used Algorithms 218
7.6 Conclusions 219
7.7 Future Work 222
7.8 List of Abbreviations 225

Glossary 226

Bibliography 228

List of Tables

3.1 Node structure. 54
3.2 Parameters for the Tile World in 55
3.3 The GNP results on the Tile World Problem 57
3.4 Parameters for the Tile World with Heavy Tile. 58
3.5 The GNP results on the Heavy Tile World Problem using one sub-program 59
3.6 Parameters for the Tile World with Heavy Tile with two sub-programs with VSGNP. 61
3.7 The VSGNP results on the Tile World Problem with Heavy Tile 62
3.8 Parameters for the Tile World with Heavy Tile with two sub-programs. 63
3.9 The GNP results on the Prey and Predator Problem 64
3.10 Parameters for the Tile World with GNP-CC-OS-TP. 66
3.11 The GNP-CC-OS-TP results on the Tile World Problem 68
3.12 Parameters for the Tile World Problem with Heavy Tile Using one sub-program with GNP-CC-OS-TP. 69
3.13 GNP-CC-OS-TP results on the Tile World Problem with Heavy Tile using one sub-program 70
3.14 Parameters for the Tile World with Heavy Tile with two sub-programs with VSGNP-CC-OS-TP. 72
3.15 The VSGNP-CC-OS-TP results on the Tile World with Heavy Tile with two sub-programs 73
3.16 Parameters for the Prey and Predator with GNP-CC-OS-TP. 74
3.17 GNP-CC-OS-TP results on the Prey and Predator Problem 76
3.18 Parameters for the Tile World Problem with GNP-RL. 77
3.19 GNP-RL results on Tile World Problem. 78
3.20 Parameters for the Tile World Problem with Heavy Tile Using one sub-program with GNP-RL. 80
3.21 GNP-RL results on the Heavy Tile World Problem with one sub-program 81
3.22 Parameters for the Tile World Problem with Heavy Tile Using two sub-programs with VSGNP-RL. 83

3.23 VSGNP-RL results on the Tile World Problem with Heavy Tile using two sub-programs 85
3.24 Prey and Predator with GNP-RL. 86
3.25 GNP-RL results on the Prey and Predator Problem 87
3.26 Parameters for the Tile World Problem with GNP-RL-CC-OS-TP . . . 90
3.27 GNP-RL-CC-OS-TP results on Tile World Problem. 91
3.28 Parameters for the Tile World Problem with Heavy Tile Using one sub-program with GNP-RL-CC-OS-TP. 94
3.29 GNP-RL-CC-OS-TP results on the Heavy Tile World Problem with one sub-program. 95
3.30 Parameters for the Tile World Problem with Heavy Tile Using two sub-programs with VSGNP-RL-CC-OS-TP. 98
3.31 The VSGNP-RL-CC-OS-TP results on the Heavy Tile World Problem using two sub-programs 99
3.32 Parameters for the Prey and Predator with GNP-RL-CC-OS-TP. 102
3.33 GNP-RL-CC-OS-TP results on the Prey and Predator Problem. 103

4.1 Parameters for the Tile World Problem: PCK-GNP. 114
4.2 Comparison of results on Tile World Problem: PCK-GNP 117
4.3 Parameters for the Heavy Tile World Problem: PCK-GNP with one subprogram. 118
4.4 Comparison of Results on Heavy Tile World Problem: PCK-GNP with one subprogram. 122
4.5 Parameters for the Heavy Tile World Problem: PCK-VSGNP with two subprograms 123
4.6 Comparison of Results on Heavy Tile World Problem: PCK-VSGNP with two subprograms. 126

5.1 Node structure. 138
5.2 Parameters for the Tile World Problem: A*-GNP. 139
5.3 Parameters for the Tile World Problem: CDA*-GNP. 149
5.4 The differences between CDA*-GNP and GNP-RL. 157
5.5 Parameters for the Tile World Problem with Heavy Tile Using one sub-program with CDA*-GNP. 159
5.6 Parameters for the Tile World with Heavy Tile with two sub-programs with CDA*-VSGNP. 160
5.7 Parameters for the Prey and Predator with CDA*-GNP. 163
5.8 Performance comparison on the Tile World Problem. 166

5.9 Performance of Algorithms (using one sub-program) on the Heavy Tile World. 169
5.10 Performance of Algorithms (using two sub-programs) on the Heavy Tile World. 172
5.11 Performance of algorithms on the Prey and Predator Problem. 175

6.1 Parameters for the CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN). 184
6.2 Comparing the Algorithms' results on Fully Heterogeneous Tile World Problem. 197
6.3 Comparing the Algorithms' results on Partly Heterogeneous Tile World Problem. 200

7.1 Performance comparison of the different variants of PCK-GNP on the Tile World Problem 206
7.2 Performance comparison of the different variants of GNP with one sub-program (and parameter settings) on the Heavy Tile World 207
7.3 Performance comparison of the different variants of VSGNP with two subprograms (and parameter settings) on the Heavy Tile World 208
7.4 Performance comparison of the different variants of the GNP algorithm (with parameter settings) on the Tile World Problem 209
7.5 Performance comparison of the different variants of the GNP algorithm (with one subprogram and parameter settings) on the Heavy Tile World 210
7.6 Performance comparison of the different variants of the VSGNP algorithm (with two subprograms and parameter settings) on the Heavy Tile World 210
7.7 Performance comparison of the different variants of the GNP algorithm (with their parameters) on the Prey and Predator Problem 211
7.8 Performance comparison of the different variants of the CDA*-GNP-H algorithm on the Fully Heterogeneous Tile World 212
7.9 Performance comparison of the different variants of the CDA*-GNP-H algorithm on the Partly Heterogeneous Tile World 213
7.10 Comparing the computational cost in the proposed algorithms when testing the Tile World Problem 218
7.11 Abbreviations 225

List of Figures

1.1 Tile World Problem Training Set (1) 4
1.2 Heavy Tile World Problem 6
1.3 Heavy Tile in Heterogeneous System Training Set (1) 8
1.4 Heavy Tile in Heterogeneous System Testing Set (2) 9
1.5 Heavy Tile in Heterogeneous System Testing Set (3) 9
1.6 Prey and Predator Training Environment 10
1.7 Challenges of evolving a meta-level reasoning strategies 11

2.1 Genetic Algorithm 18
2.2 Genotype of Genetic Algorithm Chromosome Structure 18
2.3 Phenotype Genetic Algorithm Crossover Operation 20
2.4 Genetic Algorithm Mutation Operation 20
2.5 Phenotype Genetic Programming Chromosome Structure 21
2.6 Genetic Programming Crossover Operation 23
2.7 GNP Genotype expression for Chromosome Structure 24
2.8 GNP Phenotype Chromosome Structure 25
2.9 Genetic Network Programming algorithm (training phase) 26
2.10 Genetic Network Programming algorithm (testing phase) 27
2.11 GNP Crossover modified from Mabu Et. Al. 2010. 28
2.12 GNP Mutation modified from Mabu Et. Al. 2010. 29
2.13 GNP-RL Chromosome Structure. 33
2.14 Genetic Network Programming with Reinforcement Learning algorithm (training phase). 34
2.15 Genetic Network Programming with Reinforcement Learning algorithm (testing phase). 35
2.16 VSGNP-RL Phenotype Chromosome Structure. 38
2.17 VSGNP-RL Crossover. 40
2.18 VSGNP-RL Mutation. 41
2.19 GNP-RL-CC-OS-TP Chromosome Structure. 46

3.1 Average Fitness in Training and Testing Stage for the Normal Tile Problem using GNP. . . . 56
3.2 Average number of dropped tiles in Training Stage for the Normal Tile Problem using GNP. . . . 56
3.3 Average Fitness in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP. . . . 59
3.4 Average number of dropped Tiles in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP. . . . 59
3.5 Heavy Tile Program Graph Structure. . . . 60
3.6 Average Fitness Training and Testing Stages for the Heavy Tile Problem using two sub-programs with VSGNP. . . . 62
3.7 Average number of dropped Tiles in Training and Testing Stages for the Heavy Tile Problem using two sub-programs with VSGNP. . . . 62
3.8 The average Fitness for the Prey and Predator Problem using GNP – Training and Testing Stage. . . . 64
3.9 The average number of caught prey for the Prey and Predator Problem using GNP – Training and Testing Stage. . . . 65
3.10 The average Fitness for the Tile World Problem using GNP-CC-OS-TP in Training and Testing Stage. . . . 67
3.11 The average number of dropped tiles in the Tile World Problem using GNP-CC-OS-TP Training and Testing Stage. . . . 68
3.12 Average Fitness in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP-CC-OS-TP. . . . 71
3.13 Average number of tiles in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP-CC-OS-TP. . . . 71
3.14 Average Fitness in Training and Testing Stages for the Heavy Tile Problem with two sub-programs using VSGNP-CC-OS-TP. . . . 73
3.15 Average number of dropped Tiles in Training and Testing Stages for the Heavy Tile Problem with two sub-programs using VSGNP-CC-OS-TP. . . 73
3.16 The average fitness for the Prey and Predator Problem – Training and testing Stages using GNP-CC-OS-TP. . . . 75
3.17 The average number of prey caught for the Prey and Predator Problem - Training and Testing Stages using GNP-CC-OS-TP. . . . 75
3.18 Average Fitness in Training Stage for the Normal Tile Problem using GNP-RL. . . . 77
3.19 Average number of dropped Tiles in the Training Stage for the Normal Tile Problem using GNP-RL. . . . 78

3.20 Average Fitness in Testing Stage for the Normal Tile Problem using GNP-RL. . . . 78
3.21 Average number of dropped Tiles in the Testing Stage for the Normal Tile Problem using GNP-RL. . . . 79
3.22 Average Fitness in Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL. . . . 81
3.23 Average number of dropped tiles in the Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL. . . . 81
3.24 Average Fitness in Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL. . . . 82
3.25 Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL. . . . 82
3.26 Average Fitness in Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL. . . . 84
3.27 Average number of dropped Tiles in the Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL. . . . 84
3.28 Average Fitness in Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL. . . . 85
3.29 Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL. . . . 85
3.30 The average fitness for the Prey and Predator Problem – Training Stage using GNP-RL. . . . 87
3.31 The average number of caught prey for the Prey and Predator Problem – Training Stage using GNP-RL. . . . 88
3.32 The average fitness for the Prey and Predator Problem - Testing stage using GNP-RL Figure. . . . 88
3.33 The average number of caught prey for the Prey and Predator Problem - Testing stage using GNP-RL. . . . 89
3.34 Average Fitness in Training Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP. . . . 91
3.35 Average number of dropped tiles in Training Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP. . . . 92
3.36 Average Fitness in Testing Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP. . . . 92
3.37 Average number of dropped tiles in Testing Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP. . . . 93
3.38 Average Fitness in Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP. . . . 96

3.39 Average number of dropped Tiles in Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP. 96
3.40 Average Fitness in Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP. 97
3.41 Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP. 97
3.42 Average Fitness in Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP. 99
3.43 Average number of dropped Tiles in Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP. 99
3.44 Average Fitness in Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP. 100
3.45 Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP. 100
3.46 Comparing the using of one and two sub-programs with Heavy Tile World Problem for the GNP extensions. 101
3.47 The average fitness for the Prey and Predator Problem – Training Stage using GNP-RL-CC-OS-TP. 103
3.48 The average number of caught prey for the Prey and Predator Problem – Training Stage using GNP-RL-CC-OS-TP. 103
3.49 The average fitness for the Prey and Predator Problem - Testing stage using GNP-RL-CC-OS-TP. 104
3.50 The average number of caught prey for the Prey and Predator Problem - Testing stage using GNP-RL-CC-OS-TP. 104

4.1 The Structure for the `Private-Conflict_Kernels` The sub-structs 2 and 3 will not be added to the conflict kernel because they contain less than five nodes.. 109
4.2 Average Fitness During Training on Tile World Problem: PCK-GNP. . 116
4.3 Average number of dropped tiles during training on Tile World Problem: PCK-GNP 116
4.4 Average fitness during training on Heavy Tile World Problem, PCK-GNP with one subprogram. 120
4.5 Average Number of Dropped tiles on Heavy Tile World, PCK-GNP with one subprogram. 121
4.6 Average Number of Dropped Tiles on Heavy Tile World, PCK-VSGNP with two subprograms. 124
4.7 Average Fitness on Heavy Tile World, PCK-VSGNP with two subprograms. 125

5.1 Boolean Polycell with Conflict-Directed A* example. 131
5.2 Flow chart GNP with ACO. 133
5.3 Flow chart for A*-GNP. 136
5.4 An example of applying A*-GNP on a graph (Tile World problem). . . . 142
5.5 Average fitness during training on the Tile World Problem: A*-GNP. . 143
5.6 Average number of dropped tiles during training on the Tile World Problem: A*-GNP. 143
5.7 Average fitness during training on Tile World: DBA*-GNP. 146
5.8 Average number of dropped tiles during training on Tile World: DBA*-GNP. 146
5.9 Flow chart for CDA*-GNP. 147
5.10 Example: Solving the Tile World problem with CDA*-GNP. 153
5.11 Heavy Tile Program Graph Structure. 162
5.12 Average fitness during Training (Normal Tile World Problem). 164
5.13 Average number of dropped tiles during Training (Normal Tile Problem). 165
5.14 Average Fitness (Testing Stage), Normal Tile World Problem. 165
5.15 Average number of dropped tiles (Testing Stage), Normal Tile World Problem. 166
5.16 Average fitness during training (Heavy Tile World Problem) using one sub-program. 167
5.17 Average number of dropped tiles during training (Heavy Tile World) using one sub-program. 168
5.18 Average Fitness (Testing Stage, Heavy Tile World) using one sub-program. 168
5.19 Average number of dropped tiles (Testing Stage, Heavy Tile World) using one sub-program. 169
5.20 Average fitness during training on the Heavy Tile World: using two subprograms. 170
5.21 Average number of dropped tiles duing training on the Heavy Tile World: using two subprograms. 171
5.22 Average fitness (Testing Stage, Heavy Tile World): using two subprograms. 171
5.23 Average number of dropped tiles (Testing Stage, Heavy Tile World): using two subprograms. 172
5.24 The average fitness for the Prey & Predator Problem – Training Stage. . 173
5.25 The average number of Prey caught for the Prey & Predator Problem – Training Stage. 174
5.26 The average fitness for the Prey & Predator Problem - Testing stage. . . 174
5.27 The average number of Prey caught for the Prey & Predator Problem - Testing stage. 175

6.1 Merging and Splitting the population in the Teams 181
6.2 Fully heterogeneous system structure 182
6.3 Randomly chosen team 183
6.4 The diference between Euclidean distance and A* 185
6.5 Heterogeneous (Random) 187
6.6 CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN) 190
6.7 Extracting CDA*-GNP path from a heterogeneous Team 190
6.8 Exchanging the nodes' level 191
6.9 CDA*-GNP with Heterogeneous system to explore the Members (CDA*-GNP-HM) 192
6.10 Conflict between members. A1 try to push T1 into the hole, while A2 is in the way between them, so A2 causes a conflict with A1 193
6.11 Exploring Members with CDA*-GNP in Heterogeneous system 194
6.12 Fitness of the CDA*-GNP-H in Fully Heterogeneous system 196
6.13 Number of dropped Tiles of the CDA*-GNP-H in Fully Heterogeneous system 196
6.14 Fitness of the CDA*-GNP-H in Partly Heterogeneous system 199
6.15 Number of dropped Tiles of the CDA*-GNP-H in Partly Heterogeneous system 199

7.1 Usage of the various node types in the best chromosomes returned by PCK-GNP and CDA*-GNP on the three problems 217

Abstract

This research presents a novel hybrid evolutionary algorithm for generating meta-level reasoning strategies through computational graphs to solve multi-agent planning and collaboration problems in dynamic environments using only a sparse training set. We enhanced Genetic Network Programming (GNP) by reducing its reliance on randomness, using conflict extractions and optimal search in computational mechanisms to explore nodes more systematically. We incorporated three algorithms into the GNP core. Firstly, we used private conflict kernels to extract conflict-generating structures from graph solutions, which enhances selection, crossover, and mutation operations. Secondly, we enhanced the GNP algorithm by incorporating optimal search and merged Conflict Directed A* with GNP to reduce the search branching factor. We call our novel algorithm Conflict-Directed A* with Genetic Network Programming (CDA*-GNP), which identifies the most effective combination of processing nodes within the graph solution. Additionally, we investigated the use of a chromosome structure with multiple subprograms of varying sizes that the algorithm automatically adjusts. Thirdly, we applied Conflict-Directed A* to a genetically co-evolving heterogeneous cooperative system. A set of agents with diversified computational node composition is evolved to identify the best collection of team members and to efficiently prevent conflicting members from being in the same team. Also, we incorporated methods to enhance the population diversity in each proposed algorithm. We tested the proposed algorithms using four cooperative multi-agent testbeds, including the prey and predator problem and the original tile world problem. More complex multi-agent and multi-task benchmarking testbeds were also introduced for further evaluation. As compared to existing variants of GNP, experimental results show that our algorithm has smoother and more stable fitness improvements across generations. Using the popular tile world problem as a benchmarking testbed, CDA*-GNP achieved better performance results than the best existing variant of GNP for solving the problem. Our algorithm returned 100% accuracy on the test set compared to only 83% reported in the literature. Moreover, CDA*-GNP is 78% faster in terms of the average number of generations and 74% faster in terms of the average number of fitness evaluations. In general, our findings suggest that a hybrid algorithm that balances the utilization of Genetic Network Programming and Optimal strategies leads to the evolution of high-quality solutions faster.

Acknowledgements

To my beloved Parents, I am forever grateful for the care, attention, and prayers you have showered upon me. Your support and love have been my guiding light in achieving all that I am today. To my dear siblings, I am truly grateful for all the support and prayers you have given me.

To my husband, Bandar, I cannot express how grateful I am for your support, and I cannot thank you enough for believing in me every step of the way. Your constant support and motivation have been the driving force behind our journey towards our goals. Without you, I would not have achieved what I have today.

To my beloved children (Reema & Sara & Mohammed), I eagerly await the moment when you read this thesis. You have been my ultimate inspiration, driving me to work tirelessly towards this accomplishment.

To my supervisor Dr. Napoleon Reyes, I would like to express my sincere gratitude for the support and expert advice you provided to me throughout the years of my studies. Your invaluable input and time were instrumental in enabling me to successfully complete this thesis, and I am privileged to have had the opportunity to study under your supervision.

To my co-supervisor Dr. Andre Barczak, thank you for your invaluable assistance and your expert guidance in refining my thesis. I am grateful for your support.

This thesis serves as a tribute to my home the Kingdom of Saudi Arabia, an esteemed country that provided me with the opportunity and support to accomplish my higher education. I am grateful for the chance to have embarked on this journey, which has enabled me to gain valuable knowledge and insights.

Chapter 1

Introduction

1.1 Background

Over the years, scientists have developed powerful biologically inspired algorithms that are suitable for classification, regression, control, knowledge representation and optimization problems. One class of algorithms, pioneered by Holland [1], borrowed the mechanics of evolution, such as selection, crossover and mutation, eventually emerging under the umbrella of evolutionary algorithms. The algorithm was developed based on simulating the transmission and evolution of well-adapting genes in living organisms from generation to generation.

Holland represented a set of candidate solutions as a population of chromosomes (individuals), where each chromosome is a concatenation of genes representing the different solution parameters or an abstraction of it. Using a fitness function to evaluate and select individuals for mating, a near-optimal solution can be obtained after several generations of applications of evolution operations on the population of individuals. After that, the GA has been developed and enhanced by applying new approaches for each phase (step) [evaluation, selection, crossover, mutation, chromosome structure ...] of the algorithm. The solution structure has also been changed to a tree structure, as in Genetic Programming (GP) by Koza in 1994 [2]. It was a successful algorithm, and it was used in Robot Soccer by Ciesielski et al. in 2002 [3] and the Tile World problem by Li et al. in 2010 [4]. Another genetic algorithm called Genetic Networking Programming (GNP) was produced by Katagiri et al. [5], which used a networking graph structure as a chromosome structure. This graph contains several node types, namely Start, Judgment and Processing nodes. For each graph, there is one start node, but there is no terminal node. The same node can be visited more than once and from multiple agents. The effectiveness of this structure is that it can cover more rules with a smaller size structure than the tree. In 2007, reinforcement learning (RL) using the Sarsa algorithm was added to the GNP algorithm to enable the algorithm to solve more

intricate problems. It allowed several sub-nodes for each node and ran the RL to learn how to use the graph and which sub-node is the best. This algorithm, called (GNP-RL), was introduced by Mabu et al. [6]. This genetic algorithm has outperformed previous versions in solving various problems, such as the problem of the Tile World, which the algorithm was applied to by Mabu et al. [6]. In 2019, we improved the GNP-RL with the multi-agent dynamic benchmark problem by increasing its performance and evolution time. Even though this approach has achieved the highest result when applied to the Tile World problem, it still has some limitations that need improvement.

Firstly, the evaluation time is long, especially when the problem needs to be simulated to evaluate individuals, such as the Tile World Problem. Secondly, genetic algorithms depend on randomization. The benefit of randomization is to find the answer faster than a direct search, but sometimes this randomization depends on luck, and sometimes it leads to losing the optimal solution. The Selection, Crossover, and Mutation still have some randomization in their work. Hence, to reduce the randomization in this algorithm, we used Private-conflict-kernel to extract the conflict-generating sub-structure from the graph and prevent generating children that contain these conflicts. Furthermore, we used them in the selection, crossover, and mutation to decrease the randomization and increase the optimization. This technique also reduces time by speeding up evolution and always finding better children. Moreover, to increase the optimality and decrease the needed evolution time in finding the best solution, we incorporated the optimal search (Conflict-Directed A*) to explore all the possible action nodes, excluding the ones that cause a conflict in the graph. Additionally, as we see from the GNP-RL implementation's success in finding the best graph for all agents in the multi-agent systems, the problem could be solved using multi-agent with just one graph, but what happens if each agent has different abilities, sensors, or action and they should work together to accomplish the same goals? Thus, we tested the genetic heterogeneity in cooperative coevolution in the new approach of GNP using (Conflict-Directed A*). By integrating strategies based on conflict-directed A*, and the genetic heterogeneity with GNP, we developed an algorithm suited to solving multi-agent collaboration. We tested the effectiveness of our approaches in four testbeds: Tile World problem, Heavy Tile World with various weights problem, Prey and Predator problem, and Heavy Tile in Heterogeneous system.

1.2 Overview of the Problem Domain

Stone et al. [7] raised a general problem as a challenge to the AI community: the automatic formation of an ad hoc team capable of collaboration without pre-coordination. He also gave an interesting example explaining the problem domain's importance. A

scenario about a person who just had an accident on a bicycle, and there were a number of people who wanted to help the victim. Assuming that the people do not speak the same language, and each one of them has a different knowledge, background and capabilities, how can they possibly work as a team in order to accomplish one common task. One of them will call the emergency phone number, and another will do the first aid, and so on. But in this case, how will they work to save the victim without conflict while they cannot communicate properly? Each of them will rely on what can be observed from other people's behaviour, and then they will determine the best action to take at the best time. Hence, multi-agents with different sensors and actions which are put together to solve the same task are called ad hoc teams. An ad hoc team appears between robots or agents that have been programmed in different places and times, such as different robots being sent from different places in the world after an earthquake to conduct rescue tasks, where they should be able to adapt to working together without prior coordination.

The Tile World problem and the Prey and Predator Problem are two of the best examples that emphasize the need for collaboration between ad hoc team members, which will be used to test our proposed algorithms. The Tile World Problem will be used as a homogeneous problem, and an extension of the Tile World Problem with a heavy tile will be used as a heterogeneous problem. This thesis will provide a full explanation of these two problems and heterogeneous systems.

1.2.1 Tile World Problem

The Tile World problem is one of the multi-agent, dynamic system benchmark problems. In 1990, Pollak et al. explained this problem [8], which is a 2-D grid world with 5 different objects: Agents, Tiles, Holes, Floors, and Obstacles. The agents should push all the tiles into the holes before the available number of steps is finished. The agent can answer 8 queries and can execute 4 actions:

Queries:

1. What is in the forward position? (JF)
2. What is in the backward position? (JB)
3. What is in the right position? (JR)
4. What is in the left position? (JL)
5. Direction to the nearest Tile? (TD)
6. Direction to the nearest hole? (HD)

7. Direction to the second nearest Tile? (THD)

8. Direction from the nearest Tile to the nearest hole? (STD)

Actions:

1. Go Move forward (MF)

2. Turn Left (TL)

3. Turn Right (TR)

4. Stay (ST)

Training and testing environments

Training set:Figure 1.1(1) from [9], has 10 environments. Each one of them contains the same locations for the agents, obstacles, and holes but with different locations for the tiles. This environment is used as a training set for all the experiments in this thesis. Testing sets: Figure 1.1(2) from [9], has 10 environments. Each one of them contains the same locations of the agents as the training environments but with different locations for the tiles, holes, and obstacles. Figure 1.1(3) from [10] is another set of 10 testing environments with the exact locations as the training set for the agents, obstacles, and holes, but with different locations for the tiles.

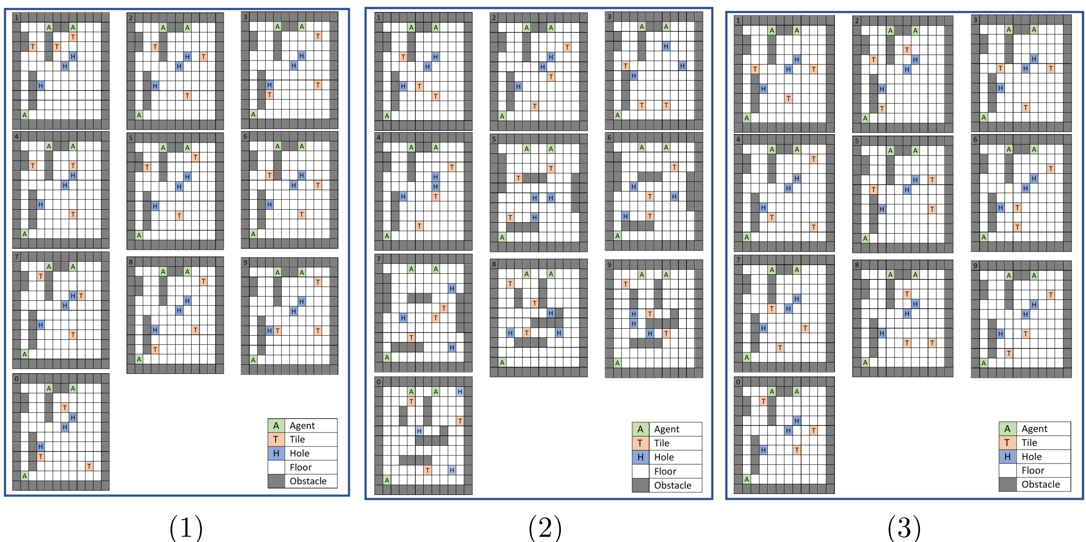


Figure 1.1: Tile World Problem. Training set (1) [9] and Testing sets (2) [9] and (3) [10].

1.2.2 Heavy Tile World Problem

We have extended the complexity of the Tile World problem by changing one tile in each environment to a heavy tile that needs two agents (Go forward together) to push it. The parameter settings for this problem are similar to the typical Tile World problem parameters, except for the following changes:

1. The maximum number of steps for each agent is 100 steps because the agents need extra time to wait for another agent to work collaboratively to push a heavy tile.

2. The training and testing sets for the heavy tile are produced from the training set used in [9] by changing one tile in each environment to a heavy tile. For the testing sets, they are also derived from the same study [9] by converting one tile to a heavy one. (See Figure 1.2).

3. One of the Judgment nodes in the Tile World Problem called (The direction to the second nearest Tile) is changed to return the type of the nearest Tile.

Distinguishing Simulation Elements

Objects and their attributes

The Heavy Tile World Problem is a 2-D grid world with 6 different objects: Agents, Tiles, Htiles, Holes, Floors, and Obstacles. The agents should push all the tiles into the holes before the available number of steps is finished. The agent can answer 8 queries and can execute 4 actions. Htile is the heavy tile that needs two agents to be pushed, one agent can't push the heavy tile (Htile).

Queries and their answers

There are 8 different types of judgment nodes in this problem:

1. What is in the forward position? (JF)

2. What is in the backward position? (JB)

3. What is in the right position? (JR)

4. What is in the left position? (JL)

5. Direction to the nearest Tile? (TD)

6. Direction to the nearest Hole? (HD)

7. Direction to the second nearest Tile? (THD)

8. Type of the nearest Tile (Tile, HTile)? (TT)

For (JF), (JB), (JR), and (JL) the answer can be one of {Agent, Tile, HTile, hole, obstacle, or Floor}. For (TD), (HD), and (THD) the answer can be one of {Forward, Backward, Right, Left, or nothing}. For (TT) the answer can be {Normal or Heavy}.

Actions

There are 4 different actions the agents can execute:

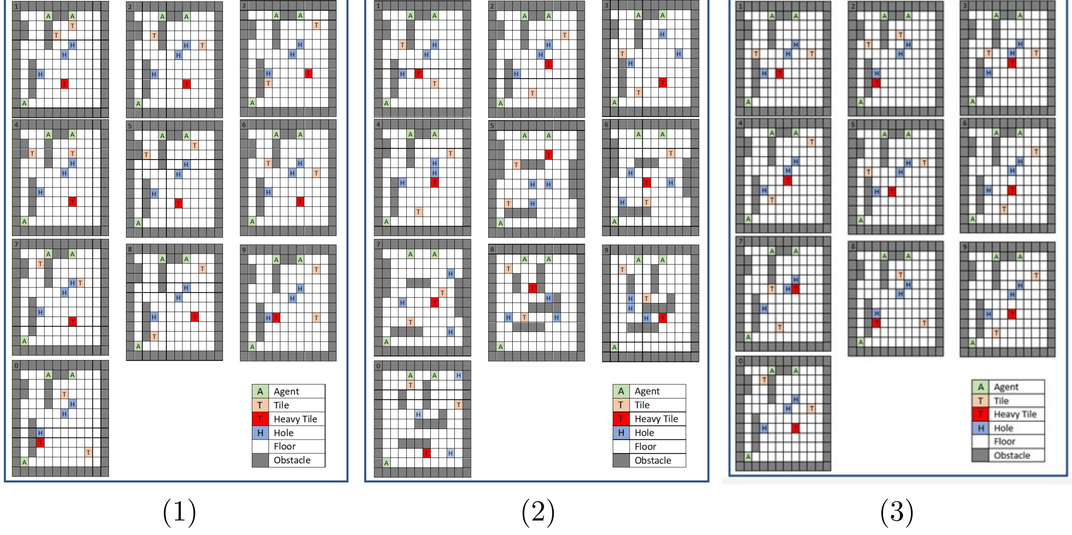
1. Go Forward (MF)
2. Turn Left (TL)
3. Turn Right (TR)
4. Stay (ST)

Simulation

Using the objects, queries, and actions explained above, the agents should push the tiles and htiles to the holes within 100 steps for each agent. The proposed algorithms in this thesis were able to solve this problem and the agents were able to be collaborative together to push the heavy tiles (htiles). This YouTube link (https://youtu.be/pHwvBv6RdVo) is a simulation of one of the best solutions for one of the proposed algorithms. The simulation clearly shows how the agents collaborate to solve the Heavy Tile World Problem.

Training and testing environments

In this work, there are 10 training environments, each one of them has 3 agents, 3 holes, 2 tiles, and 1 heavy tile (see Figure 1.2).



(1) (2) (3)

Figure 1.2: Heavy Tile World Problem. Training sets (1) and testing sets (2)(3).

1.2.3 Heavy Tile in Heterogeneous System

To apply the proposed algorithm in a heterogeneous system, we modified the Heavy Tile World Problem by changing one of the agents from each environment to be a strong agent, that can push a heavy tile alone without help from another agent, it can also push the normal tile. We also combined the ten environments in one environment that has 30 agents, 30 tiles, and 30 holes (see Figures 1.3, 1.4, and 1.5) as the training set (1), testing set (2), and testing set (3), respectively. To incorporate the new agent, the partly heterogeneous system needs at least two different populations, one for each type of agent (`Normal`, `Strong`) because each agent type has its actions that need a different chromosome to represent its actions. And for the fully heterogeneous system, we need 30 different populations, one for each agent. A full explanation of this problem will be provided in Chapter 6.

Distinguishing Simulation Elements

Objects and their attributes

The Heterogeneous Tile World Problem is a 2-D grid world with 7 different objects: Agents, SAgents, Tiles, Htiles, Holes, Floors, and Obstacles. The agents should push all the tiles into the holes before the available number of steps is finished. The agent can answer 8 queries and can execute 4 actions.SAgent is a strong agent which can push the heavy tile.

Queries and their answers

There are 8 different types of judgment nodes in this problem:

1. What is in the forward position? (JF)
2. What is in the backward position? (JB)
3. What is in the right position? (JR)
4. What is in the left position? (JL)
5. Direction to the nearest Tile? (TD)
6. Direction to the nearest Hole? (HD)
7. Direction to the second nearest Tile? (THD)
8. Type of the nearest Tile (Tile, HTile)? (TT)

For (JF), (JB), (JR), and (JL) the answer can be one of {Agent, SAgent, Tile, HTile, Hole, Obstacle, or Floor}. For (TD), (HD), and (THD) the answer can be one of {Forward, Backward, Right, Left, or nothing}. For (TT) the answer can be {Normal

or Heavy}.
Actions

There are 4 different actions the agents can execute:

1. Go Forward (MF)
2. Turn Left (TL)
3. Turn Right (TR)
4. Stay (ST)

Simulation

Using the objects, queries, and actions explained above, the agents should push the tiles and htiles to the holes within 180 steps for each agent. This YouTube link (https://youtu.be/McIzCf200Yc) is a simulation of one of the best solutions for one of the proposed algorithms. The simulation shows how the agents can collaborate to solve the problem.

Training and testing environments

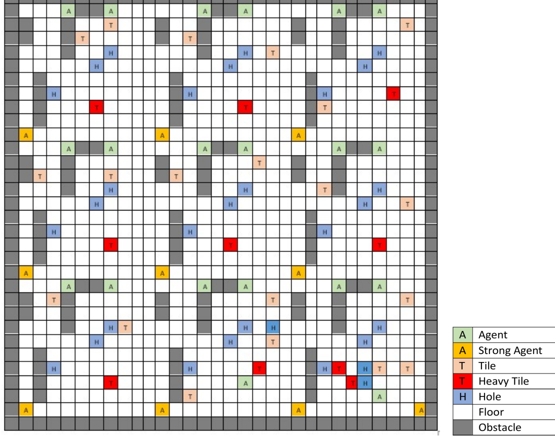


Figure 1.3: Heavy Tile in Heterogeneous System Training Set (1)

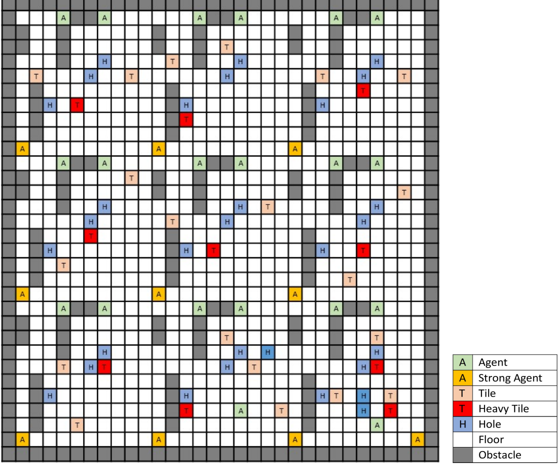


Figure 1.4: Heavy Tile in Heterogeneous System Testing Set (2)

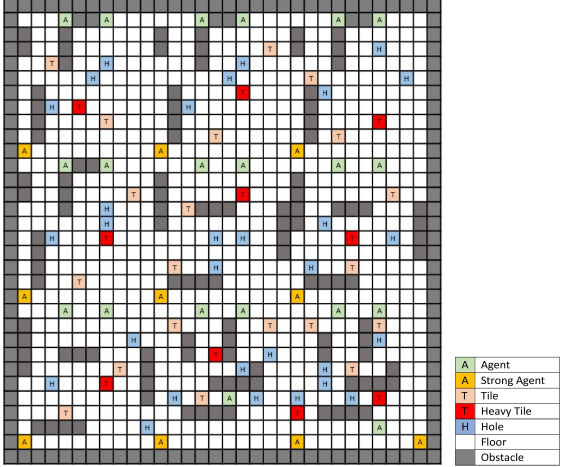


Figure 1.5: Heavy Tile in Heterogeneous System Testing Set (3)

1.2.4 Prey and Predator Problem

As the Prey and Predator problem was also used as a benchmarking testbed by other GNP research in the literature, we also used it to test the algorithms' generalization capability. It has four agents that are working together to catch the Prey which moves randomly in the environment. The problem is solved once the prey is surrounded by all four agents or if the available number of steps is finished, in which case the algorithm will close the current environment and start the next one. For the Prey and Predator problem, we used the same parameters that have been used in [11] and [12].

The agent can answer 5 queries and can execute 4 actions.

Queries:

1. What is in the forward position (JF)

2. What is in the backward position(JB)
3. What is in the right position(JR)
4. What is in the left position(JL)
5. Direction to the Prey (PD)

Actions:

1. Move forward (MF)
2. Turn Left (TL)
3. Turn Right (TR)
4. Stay (ST)

Training and testing environments

This problem has 30 different training and testing environments, each one of them starts with a random location for the Prey and the Predators, and the Prey moves randomly until it is caught by the Predator (see Figure 1.6)

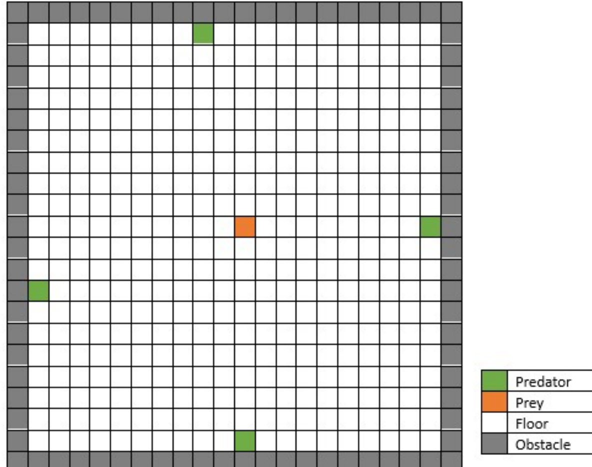


Figure 1.6: Prey and Predator Training Environment

1.3 Challenges of Evolving Meta-Level Reasoning Strategies

Meta reasoning is a process that takes place in an autonomous agent at a higher level than the reasoning algorithms, which understand the environment and determine which

actions should be taken to achieve its goals. This higher level is called the meta-level, monitors and controls the reasoning algorithms at the agent's object level. The object level contains reasoning algorithms that study the environment and determine the best time, way, and actions at the ground level that should be performed to achieve its goals [13]. In a multi-agent system (MAS), agents may use coordination and teaming algorithms. The agent's actions at ground level hold immense power to shape its environment and determine its state in the world. These actions could be processing or judgment actions.

A meta-level reasoning strategy is improved in the form of a graph including computational nodes. The graph is developed using the mechanisms of GNP, while CDA* finds the best actions for the agent to interact with a dynamic environment. A*, on the other hand, is used in a variety of judgement (query) nodes to give accurate answers.

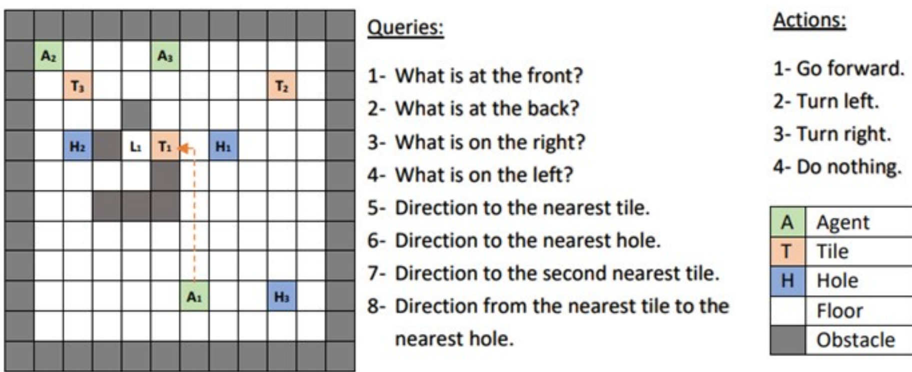


Figure 1.7: Challenges of evolving a meta-level reasoning strategies [14]

In order to demonstrate the importance of developing a meta-level reasoning strategy, Figure 1.7 illustrates a scenario in the Tile World problem: an agent (A1) should push the nearest tile (T1) into the nearest hole (H1). After applying the A* algorithm, the optimal path to reach T1 is the one that leads towards the right-hand side of the tile T1. However, to complete the task, the agent must push tile T1 to the right towards hole H1. To identify optimal paths, path planning must consider trap locations. If the tile lands at L1, it will become trapped as the robot cannot pull any tiles. Moreover, agent (A3) is competing with agent (A1) for tile T1.

All these tasks, including constraint conformance, optimal path planning, sensor querying, and taking action, must be logically and hierarchically organized by the GNP algorithm with some levels of abstraction in order to formulate a graph solution that generalizes to different world arrangements.

1.4 Research Objectives

The main objective of this work is to develop an evolutionary algorithm that is capable of generating intelligence for collaboration between multiple agents. The following outlines the specific objectives:

- To establish a standard GNP-CC-OS-TP algorithm with the three techniques (Constraint Conformance, Optimal Search, and Task Prioritization) for using it as a base to implement the new proposed algorithms.
- To develop a new multi-task and multi-agent testbed generated from the Tile World Problem to increase the complexity of the problem and build a heterogeneous system.
- To develop an extension to GNP that detects conflict-generating structures in the computational graphs and prevents the propagation of such structures in the evolutionary process. It is hypothesized that extracting the conflict sub-structures from the GNP graph and using them to improve the evolutionary operations (selection, mutation, and crossover) could help attain this objective.
- To develop a novel algorithm that effectively combines elements of Conflict Directed A*, and GNP into one algorithm and apply it to genetically homogeneous and heterogeneous systems.
- To evaluate the performance of the new proposed algorithms on four cooperative multiagent benchmarking testbeds (Tile World problem, Heavy Tile World with various weights problem, Prey and Predator problem, and Heavy Tile in Heterogeneous System).
- To compare the performance of the presented algorithms against other existing variants of GNP on the mentioned testbeds, using the standard dataset as well as using our extended and more complex environments.

1.5 Significance of the Study

The proposed algorithm aims to evolve an intelligent controller for multi-agents operating in a dynamic environment, effectively incorporating cooperative behaviour for solving some common global tasks with avoiding conflicts and reducing evolution time. Here are some of the significances of the study:

- The proposed algorithm's output is a computational graph solution (technically referred to as GNP individual or a chromosome), whereas previous works in this

area commonly make use of a tree representation or directed acyclic graphs (GP) or other machine learning without an explanation capability. The graph solution presents a human readable solution, which is highly interpretable, and allows for easier modifications even by a layperson. The proposed extended GNP aims to extend its capability to address problems generally categorised as multi-agent collaboration problems.

- Identification and avoidance of conflicts in the graph structure for faster convergence and efficiency. (This was inspired by Conflict-Directed A*). Our previous work identifies constraints in the environment, but we did not have a generalized technique that records graph structures that produce conflicts and avoid them in the future. The Conflict-Directed A* is able to do this trait, but only for a relatively simple problem of diagnosing circuits (debugging problems). The GNP graph structure is more difficult to refine as the graph involves multiple logical, relational judgment nodes and complex processing nodes.

- The complexity of the four proposed testbeds will help ascertain the effectiveness and the generalization capability of the proposed algorithms. They are multi-agent systems that need cooperative intelligence to accomplish their tasks.

1.6 Published work

As part of our preliminary explorations of this research, we have published in an A*-ranked conference (top 4%) in the field of multi-agent intelligence research worldwide, with highly strict requirements for the quality and novelty of research.

- M. A. Alshehri, N. Reyes and A. Barczak, "Evolving Meta-Level Reasoning with Reinforcement Learning and A* for Coordinated Multi-Agent Path-planning," AAMAS '20: Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems, p. 1744–1746, 2020.

1.7 Scope and Limitations of Research

The scope and limitations of the research are:

- The target solution to be generated by the proposed algorithms is a graph with computational nodes, generally categorized as judgment nodes and processing nodes. These nodes may include simple relational statements or complex intelligence algorithms such as A*, fuzzy logic, reinforcement learning, etc.

- The proposed hybrid evolutionary algorithms are tested on popular benchmarking testbeds commonly used for cooperative multi-agents, such as the Tile World problem and Prey-Predator problem.

- The agents in the Tile World problem have the same sensors and ability, and they have the same behaviour (genetically homogeneous).

- The proposed algorithm (genetically homogeneous) for the Tile World problem is aimed at generating one graph that represents the best computational structure that encodes the intelligence for the entire agents operating in the problem domain. The agents must accomplish the task as a team within the minimum number of steps.

- The proposed algorithm (genetically heterogeneous) for Heavy Tile World with variant Agents Problem will generate one graph for each agent that represents the best computational structure for each agent in order to accomplish the task as a team.

1.8 Research Questions and Hypotheses

The research questions for this thesis are:

- Can the extracted conflict-generating sub-structure (a set of series of nodes that cause a conflict) from a looping graph generated by GNP be used to improve the GNP evolutionary operations (Selection, Crossover, and Mutation)?
  - What type of conflict can be extracted from the environment?
  - Can the extracted conflict sub-structures be used to improve the Selection?
  - Can the extracted conflict sub-structures be used to improve the Crossover?
  - Can the extracted conflict sub-structures be used to improve the Mutation?

- As Conflict-directed A* has been applied only to diagnosis/debugging problems in a tree structure solution, how much improvement can we achieve on the benchmarking testbeds if we combine systematic algorithms (Conflict-directed A*) with GNP?
  - What are the types of conflicts that can be extracted from the graph?
  - How can the CDA* improve the GNP evolution?
  - What is the testing performance of the best chromosome that CDA*-GNP can generate?

- Can the CDA*-GNP be used in the Heterogeneous System?
  - What types of conflicts can be extracted from the graph in the Heterogeneous System?
  - What types of conflicts can be extracted between the team members in the Heterogeneous System?
  - Can the aforementioned algorithms succeed when applied to systems with genetic heterogeneity (multiple agents with different intelligence computational nodes but with the same overall goal)?

This thesis is structured according to seven main parts, including the Introduction. Chapter 2 presents the literature review covering Genetic algorithm (GA), Genetic Programming (GP), Genetic network programming (GNP), Genetic network programming with reinforcement learning (GNP-RL), Distributed Genetic Network Programming (DGNP-RL) and Variable-sized Genetic Network Programming (VSGNP-RL), Genetic Network Programming with Constraint Conformance, Optimal Search and Task Prioritization (GNP-RL-CC-OS-TP). Chapter 3 provides the implementation of experimenting with recent variants of Genetic Networking Programming on the three testbeds (Tile World problem, Heavy Tile World problem when using (one sub-program and two sub-programs), and Prey and Predator problem). Chapter 4 shows the proposed algorithm (Using Private Conflict Kernels with Genetic Network Programming (PCK-GNP)) and presents the results when applying it to the (Tile World problem and Heavy Tile World problem when using (one sub-program and two sub-programs)). Chapter 5 presents our novel algorithm (Using Conflict Directed A* inside Genetic Network Programming Graph) with the results when applying it to (the Tile World problem, Heavy Tile World problem when using (one sub-program and two sub-programs), and Prey and Predator problem). Chapter 6 tests the use of Conflict Directed A* with GNP on the Heterogeneous system. Finally, Chapter 7 summarizes the results and analyses all the previous chapters.

Chapter 2

Literature Review

2.1 Introduction

The development of genetic algorithms was initiated by Holland in 1975 [1], who designed the Genetic Algorithm (GA) to simulate the intricate process of genetic evolution in humans. This algorithm explores potential solutions by generating a sample of randomized solutions and improves by the evolutionary operations (Crossover and Mutation). Using a fitness function to evaluate and select individuals results in obtaining the best solution after several generations. After that, the GA was developed and enhanced by changing the solution structure to a tree structure as in Genetic Programming (GP) by Koza in 1992 and 1994 [15] [2]. Katagiri et al. developed Genetic Networking Programming (GNP), which uses a networking graph structure as a chromosome [5]. The graph is composed of three different types of nodes - Start, Judgment, and Processing nodes. Each graph has one start node, but there is no terminal node. It is possible to visit the same node more than once and by more than one agent. This structure is effective in covering more rules with a smaller size structure compared to a tree. In 2007, the Sarsa algorithm was added to GNP to solve complex problems using reinforcement learning (RL). It allowed running RL on a graph with multiple sub-nodes to learn the best sub-node. This algorithm was introduced by Mabu et al. [6] and called (GNP-RL). In 2012, Yang et al. added a new method to the GNP-RL called (DGNP-RL), which divided the chromosome structure into sub-programs to solve the multi-task problems [16]. In 2014, Mabu et al. created VSGNP-RL, a new version of Distributed GNP-RL that allows for variable-sized genetic networking programs [9]. In 2019, we improved the GNP-RL with the multi-agent dynamic benchmark problem by increasing its performance and evolution time (GNP-RL-CC-OS-TP) [17] [14].

This chapter will provide an explanation of the different algorithms that improved

from GA and compare their characteristics (GA, GNP, GNP-RL, DGNP-RL, VSGNP-RL, GNP-RL-CC-OS-TP). Then, it will explain some techniques used within the literature to improve population diversity in Genetic algorithms.

2.2 (GA) Genetic Algorithms

2.2.1 Introduction

Since Holland introduced his theory in 1975 [1] about simulating evolution in generations, the simulation of the genetic algorithm has continued to develop and improve. The algorithm's success relies on chromosome composition and problem-solving fitness [18]. There are many different types of problems that can be solved using the genetic algorithm, such as credit card fraud detection [19], edge detection [20], and breast cancer detection [21]. The Genetic Algorithm (GA) is an algorithm that can be used to generate a set of random solutions for a given problem using a series of binary numbers representing each individual (chromosome) structure. The sequence of steps involved is depicted in Figure 2.1. The algorithm begins by initializing the first population of solutions randomly. In turn, the algorithm evaluates each chromosome using a specific Fitness Function that can measure the quality of each individual solution and detect whether the optimal solution has been found. Every problem has a fitness equation that is used to evaluate solutions. In the selection stage, the best (elite) chromosomes are chosen to pass to the next generation. With the evolutionary operations (crossover and mutation), new individuals for the next generation will be generated.

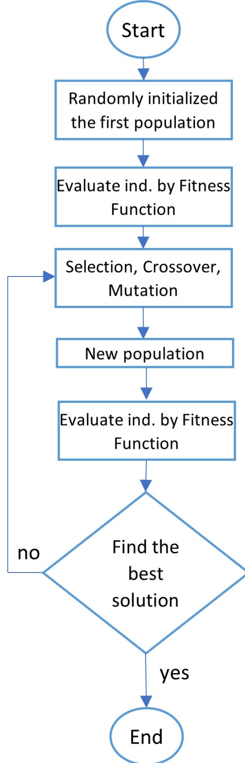


Figure 2.1: Genetic Algorithm

2.2.2 Chromosome structure

Each chromosome property is encoded as a string of zeros and ones [18]. For example, if the solution is to find a specific part in a photo that is represented by a rectangle, the chromosome will be represented by the rectangle features such as height, width, location (x, y), and rotation angle; each one of these features will be represented with a gene in the chromosome that represents the solution as Figure 2.2 [17].

Genetic Algorithm Chromosome Structure

a)	Heigh	Width	Location (x)	Location (y)	Rotation angle
b)	16	21	9	4	91
c)	0 0 0 1 0 0 0 0	0 0 0 1 0 1 0 1	0 0 0 0 1 0 0 1	0 0 0 0 0 1 0 0	0 1 0 1 1 0 1 1

Figure 2.2: Genotype of Genetic Algorithm Chromosome Structure. a) The chromosome features. b) The value of the chromosome features. c) The encoding of the chromosome [17]

2.2.3 Initialization of the First Population

The algorithm starts with randomly generating a series of zeros and ones that represent a set of solutions as a first population [18].

2.2.4 Evaluation and Selection

Evaluation

To evaluate any chromosome, a fitness function is used to calculate the performance of the individuals. This fitness function is always related to the problem, such as the number of correctly dropped tiles in the Tile World problem [6] or the Euclidean distance function as in the Travelling Salesman Problem [22].

Selection

In the selection stage, the algorithm chooses a group of individuals to send to the next generation without any changes. There are many ways to select individuals, such as truncation selection [23], tournament selection, and linear and exponential ranking selection [24].

2.2.5 Evolutionary Operations

Crossover

Crossover is an evolutionary operation that is applied to two parents to generate two new offspring with some features from each parent [1]. In the crossover operation, the algorithm can choose any type of selection method for selecting the two parents. The algorithm selects a point within the string to separate the parents and crossover them on this point. Sometimes, the algorithm utilizes multiple points to divide the chromosomes. Schaffer tested the differences when changing the location of the divided point and proved how effective it is [25]. Figure 2.3 illustrates the mechanism between the two parents to crossover them and produce two new offspring.

Mutation

The mutation is an operation applied on one chromosome to create a new child by changing its genes to change the chromosome feature in the string [1]. The most crucial aim of using the mutation is to ensure the existence of diversity among the individuals of the population. There are many different types of GA mutations, such as in 1990, Fogel and Atmar created Gaussian and uniform mutation in linear systems [26], In 1999 Larranaga examined the insertion and swap method in the Travelling Salesman Problem [27], and polynomial and power mutation, which was established by Deep in

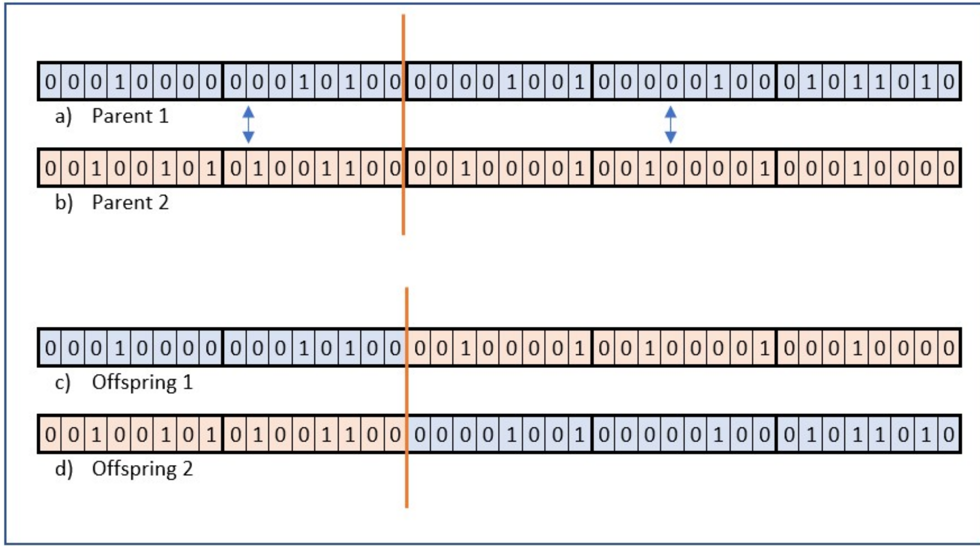


Figure 2.3: Phenotype Genetic Algorithm Crossover Operation [17]

2007 [28]. Figure 2.4 shows mutation operation in the genetic algorithm by swapping the features inside the chromosome.

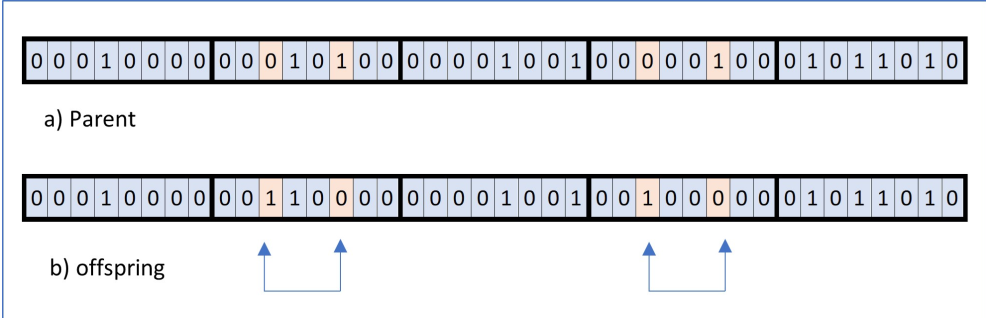


Figure 2.4: Genetic Algorithm Mutation Operation (swap genes) [17]

2.2.6 Summary

The genetic algorithm was developed to mimic genetic evolution in living organisms. It uses the structure of the chromosome to represent the solutions. These chromosomes are improved and enhanced through evolutionary operations, generation by generation, until they find the best solution. The GA has a string structure, so when it comes to complex problems, such as the ones that need complex instructions, it becomes more complex to represent the solution with a string of binary numbers [29]. To overcome this problem, a new algorithm was implemented, which was Genetic Programming (GP). The following section will discuss it.

2.3 (GP) Genetic Programming

2.3.1 Introduction

In 1992, Koza devised a new structure that can represent non-linear solutions, especially the ones that use if-then states [15] [2]. To represent the chromosome, he utilized a tree structure that can simulate making decision rules. Genetic programming algorithms can solve more complex problems that need a set of rules to solve the problem, such as biochemistry applications [30], problems in the financial area [31], breast cancer diagnosis [32], and so on.

2.3.2 Chromosome structure

A tree structure with nodes represents each chromosome. The starting point for the solutions is the root of the tree. All the nodes at the terminal are action nodes. "If-then" nodes are decision-making nodes used to make judgments. Figure 2.5 shows the structure of the Genetic Programming chromosome.

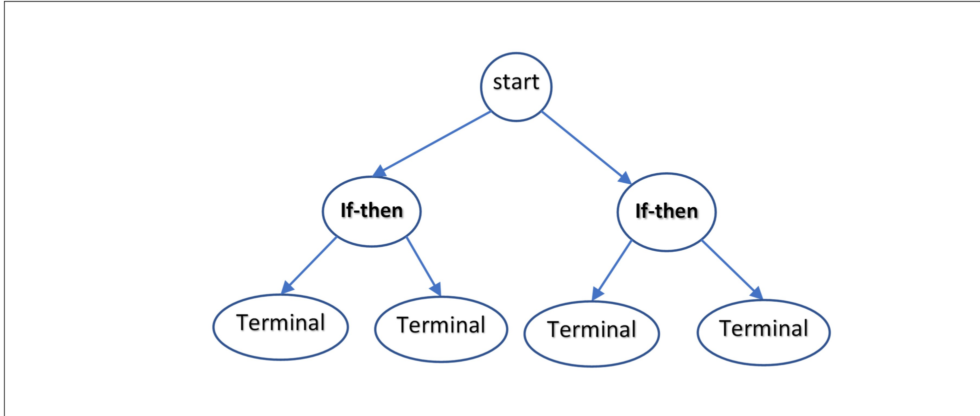


Figure 2.5: Phenotype Genetic Programming Chromosome Structure

2.3.3 Initialization of the First Population

The first step of the algorithm is to initialize the first population. It creates several random trees which represent the initial solutions. The length and depth of the tree can be predefined (called the Full method) and can be equal to or less than a detected value (called the Grow method) [29]. In the Full method, the length of all the paths between the root and the terminal node is equal to a predefined value (depth). While in the grow method, the trees have variable shapes, so, each path from the root to the terminal is equal to or less than a predefined maximum value (depth). The algorithm can adjust the depth of each tree through crossover and mutation techniques.

2.3.4 Evaluation and Selection

Once the first population is initialized, a fitness function must evaluate each chromosome. The fitness function can be any evaluated function that is appropriate for the given problem. For example, fitness is calculated by subtracting the solution's optimal value from its error [33]. In a classification problem, fitness is the ratio of correctly classified instances to the total classifications [34], and so on. The chromosome with superior fitness value is the chromosome that has a greater chance of being chosen [34].

2.3.5 Evolutionary Operations

Crossover

The selection method of the algorithm picks out two parents to create two new children using crossover. In individuals that have a tree structure, a random node is chosen from each parent to exchange the nodes between them and then produce their offspring [29] (see Figure 2.6).

Mutation

A mutation must be performed to maintain diversity in a population by inducing a modification in the genetic composition of an individual. In genetic programming (GP), a random node and its children are replaced with a newly generated node. Since the GP chromosome has a tree structure, losing diversity is not a significant issue. So, the crossover method in the GP algorithm is more effective than mutation [29]. As a result of crossover and mutation, the depth and length of the tree may increase. Consequently, a significant issue may arise, which is a significant increase in the search space and time required to evaluate individuals. That requires a larger memory size and higher processor performance. In order to prevent this issue, a new algorithm called Genetic Networking Programming has been implemented. The next section will cover the discussion of this algorithm.

2.3.6 Summary

A non-linear chromosome structure was developed using a genetic programming algorithm to solve decision-making problems. The system utilizes a tree structure with a root starting point, if-then nodes, and terminal nodes. One major limitation of genetic programming is the potential for the tree to become too large due to evolutionary operations. A solution for this problem has been developed, called the Genetic Networking Programming Algorithm (GNP), which will be discussed in the next chapter.

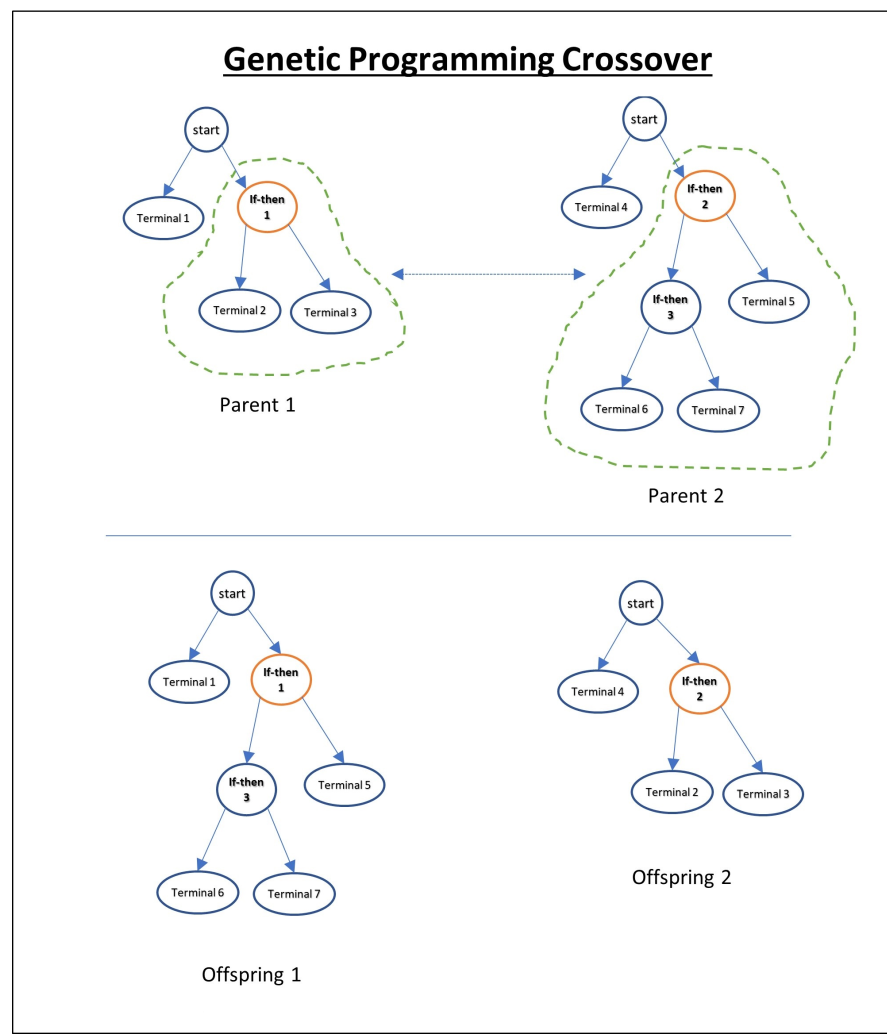


Figure 2.6: Genetic Programming Crossover Operation [17]

2.4 (GNP) Genetic Network Programming

2.4.1 Introduction

To address the issue of inadvertently generating bloated programs in GP, a new algorithm was introduced by Katagiri et al. in 2000 called Genetic Networking Programming (GNP) [5]. A networking graph structure is used to represent a chromosome, which contains several types of nodes, namely start, judgment, and processing nodes that are connected together using directed connections. Each graph has only one start node, and there is no terminal node. A node can be visited more than once and by more than one agent. The benefit of this structure is that it can cover more rules with a smaller size structure compared to a tree structure. With the use of this mechanism, the issue of oversizing has been resolved. It has been used with multi-agent systems,

such as ants behaviour [35], online learning [36], and the Prisoner's Dilemma Game [37].

2.4.2 Chromosome structure

The chromosome structure of GNP is in the form of a network (graph) that contains a set of nodes which are connected to each other (see Figure 2.7 and Figure 2.8). There are three types of nodes; the nodes in the graph could be one of three types: the start node, the Processing node that refers to the terminal node in the GP, and the Judgment node that matches the (if-then) node in GA [5]. The start node is the node where the simulation starts, the Processing node is used to perform an action or make a decision, and the Judgment node is utilized to test a specific condition. The algorithm starts by visiting the start node; then, it keeps track of the connections to move to the next node. It is possible to visit each node multiple times, and no terminal node is present. This means the algorithm will keep moving on the graph until one of these two conditions happens: either the available time for the agents runs out or the algorithm detects the solution. There are connections for each node that track the agent to the next node when it is visited. For the Judgment nodes, there are a number of connections depending on the number of answers for this judgment since each connection will refer to an answer of the judgment node. The processing node has one connection that leads to the next node when the agent visits this node and applies the action [37].

Node-id	Node type	Delay time	Connections
Gene(node)#1	Judgment node	Dt#1	C#1, C#2, …
Gene(node)#2	Processing node	Dt#2	C#1, C#2, …
Gene(node)#3	Processing node	Dt#3	C#1, C#2, …
.	.	.	….
.	.	.	….
.	.	.	….

Figure 2.7: GNP Genotype expression for Chromosome Structure modified from [5]

2.4.3 Initialization of the First Population

In the beginning, the first population is randomly generated. Every chromosome present within the population has an equal number of nodes. However, these nodes are of different types and are connected randomly.

2.4.4 Evaluation and Selection

To evaluate an individual, the agent begins from the starting node and then keeps tracking the connections from node to node by applying an action on the processing

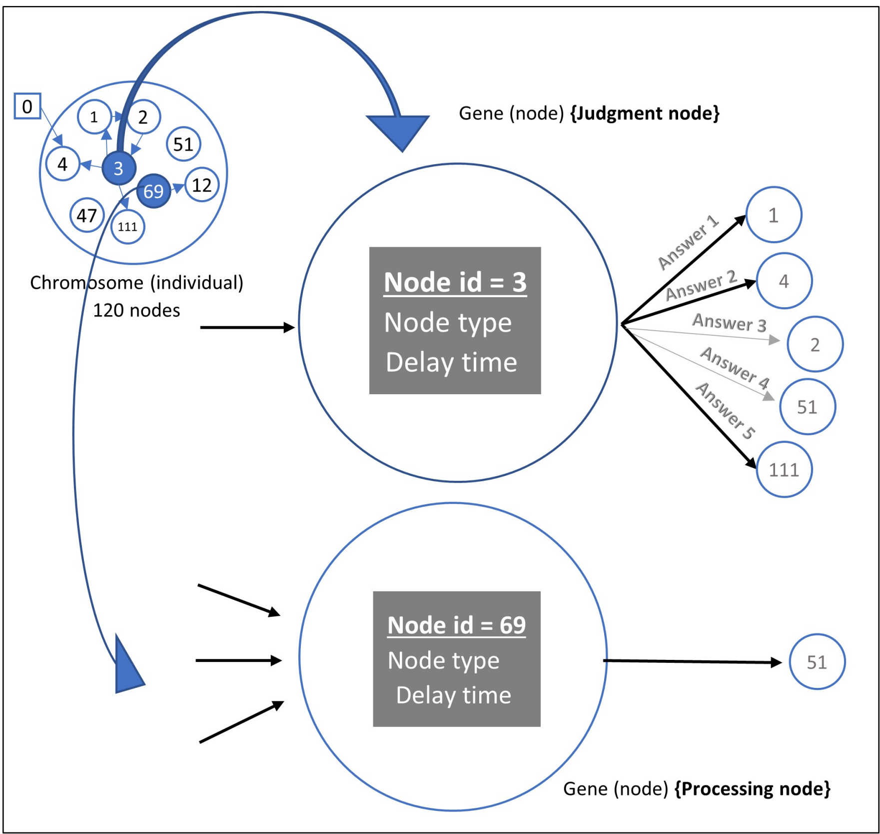


Figure 2.8: GNP Phenotype Chromosome Structure modified from [5]

nodes or giving an answer to the judgment nodes. When visiting each node, the algorithm calculates a delay time, which is a predefined number for each type of node that is used to detect the available time (steps) for each agent to stop the simulation for one chromosome and work on another [37]. The simulation is kept running until the maximum number of steps reached or if the problem is solved (see Figure 2.9 for the training phase and Figure 2.10 for the testing phase). A problem specific fitness function is used to evaluate each individual [36].

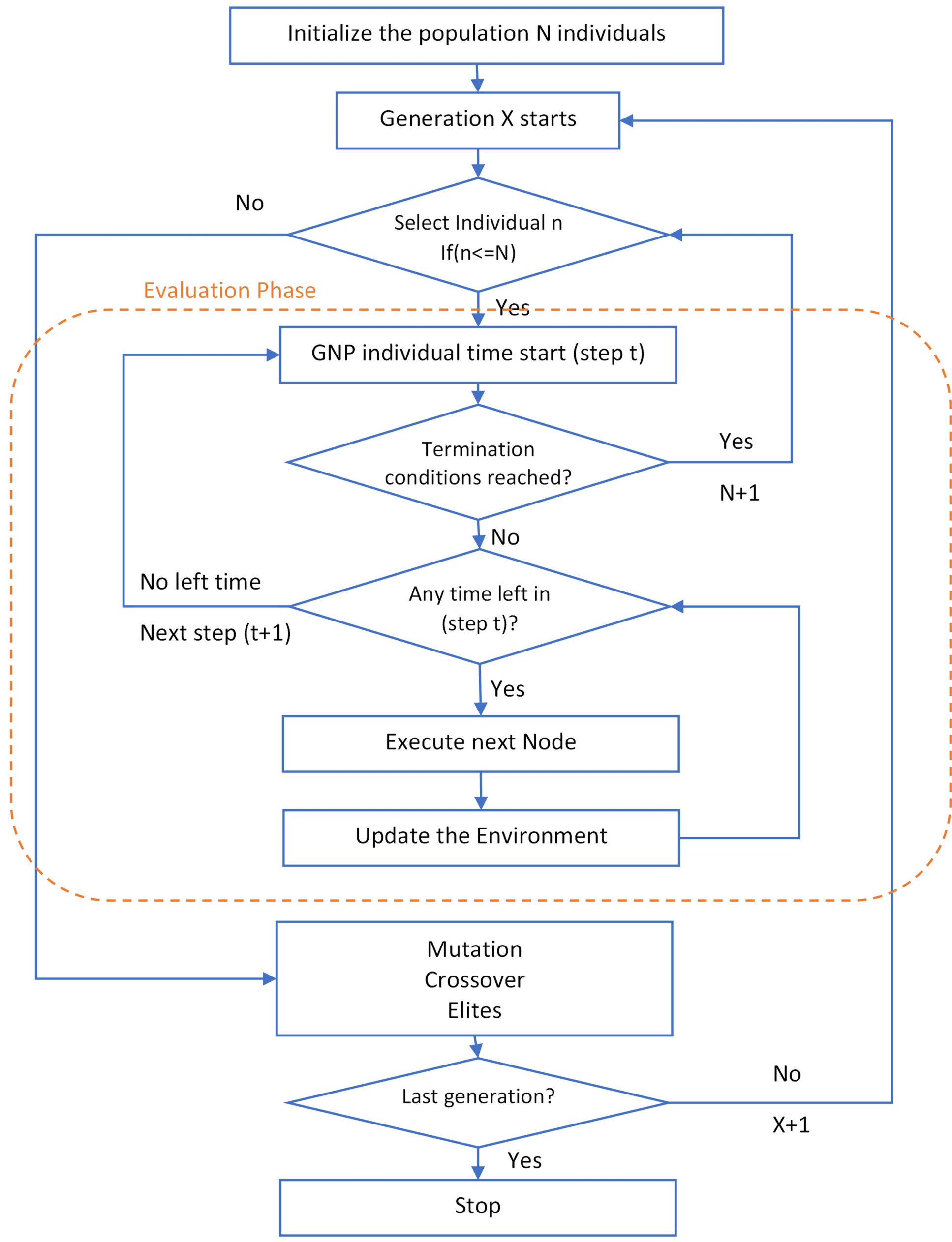


Figure 2.9: Genetic Network Programming algorithm (training phase) edited from [38] [39].

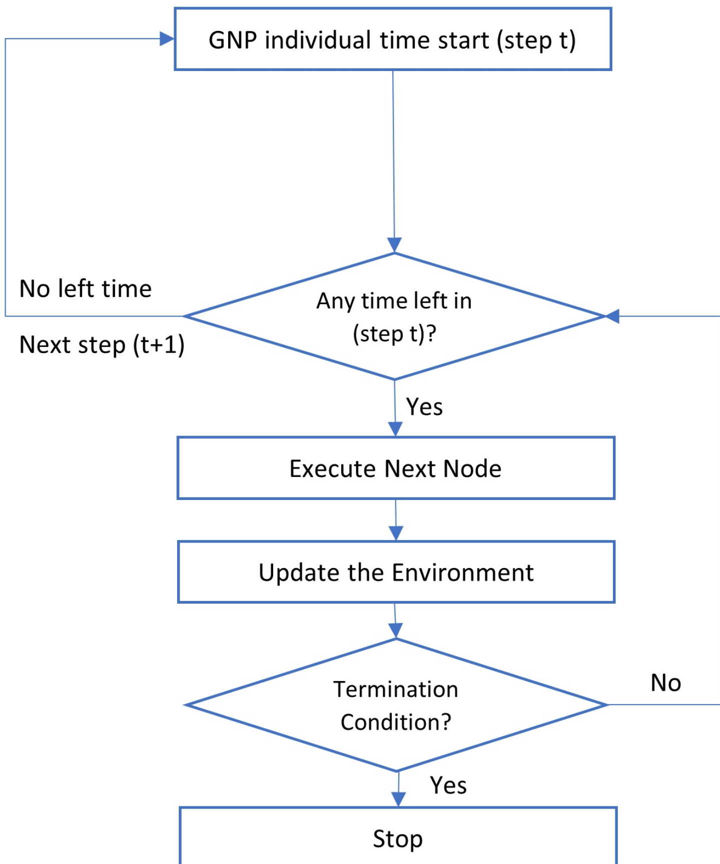


Figure 2.10: Genetic Network Programming algorithm (testing phase) edited from [38] [39].

2.4.5 Evolutionary Operations

Crossover

GNP's crossover exchanges genetic traits from parent 1 and parent 2 to produce two new offsprings that share some features from each of the parents. In the crossover stage, the algorithm uses one of the selection techniques to choose two parents, and then randomly, a per centage of the nodes from parent 1 is selected to be crossed over with another from parent2, as shown in Figure 2.11 [40].

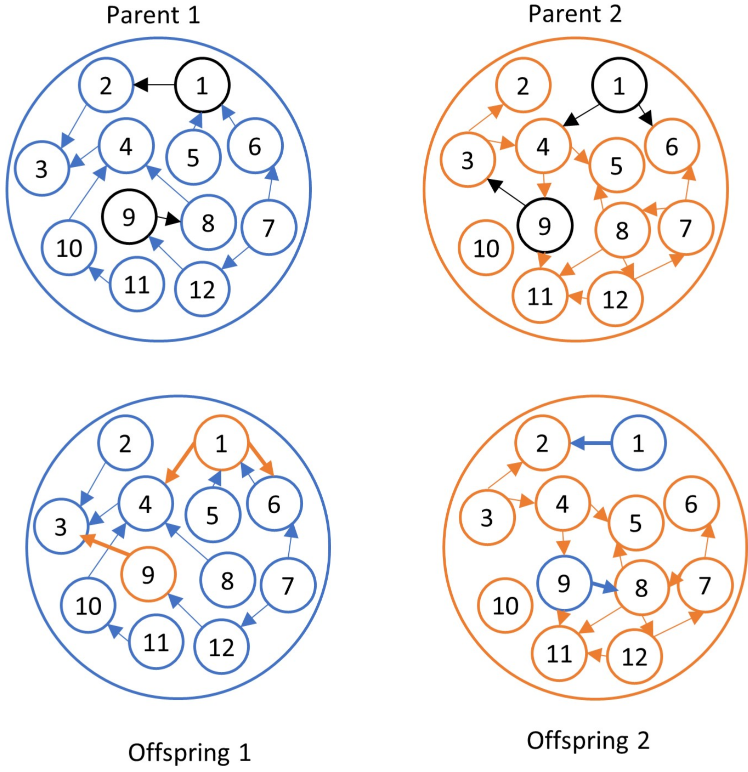


Figure 2.11: GNP Crossover modified from [40].

Mutation

There are many ways to apply mutation to the GNP. One of them was used by Mabu et al. in 2010, which selects a random node from a graph and then changes its connection to a new random one [40]. Figure 2.11 illustrates this.

2.4.6 Hyperparameters

The performance of the GNP is based on many parameters:

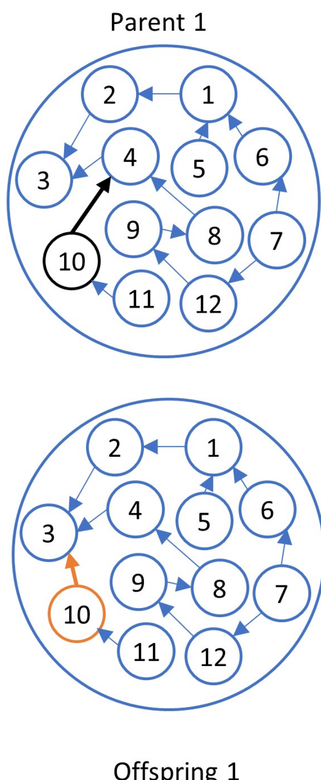


Figure 2.12: GNP Mutation modified from [40].

1. The size of the graph (number of nodes); this factor is essential in choosing the perfect number of nodes in the graph that can represent all the possible rules for a specific problem. Selecting a smaller number could lead to not finding the solution, and a larger size will increase the probabilities, leading to spending more time to find the best solution. The study conducted by Hirasawa et al. examined how the number of nodes in a GNP individual affects its performance when simulating the behaviour of ants [35].

2. Number of individuals in the generation: choosing the best number of individuals per generation is very important, as having too few individuals would cause the algorithm to face genetic diversity loss issues, while having too many individuals would cause the algorithm to spend more time evaluating the population of individuals per generation. GNP can easily find the optimal solution despite having fewer individuals compared to GP, as shown in a study that simulates ant behaviours [35].

3. Fitness function: correctly evaluating individuals is a very influential factor because good evaluation leads to the perfect choice of the individuals to be passed

on to the next generation and the ideal choice of the parents for the evolutionary operations.

4. The number of offspring that are considered to be elites, the evolutionary operations (crossover and mutation), selection of parents, and the (crossover & mutation) rates: all of these factors are essential because of their influence on the efficiency of the evolutionary process. Increasing the mutation rate will increase genetic diversity. On the other hand, if the diversity is high, we need to increase the crossover rate to help improve the fitness of the solutions. Increasing the mutation rate when the solution is nearly found could cause losing the best solution. So, the mutation and crossover rates are very important parameters. In [41] setting the mutation and crossover rate are done automatically after measuring the diversity between the individuals and the similarity between the generations.

5. Number of steps for each agent: The algorithm performance could be affected by the number of steps allowed for each agent to take (the available time for the execution of nodes). A small number of steps would not give the algorithm enough chance to solve the problem and prove its efficiency, while sometimes, a high number of steps will not increase the algorithm's performance. The higher the level of success achieved in fewer steps, the better the quality of the solution [42].

2.4.7 Summary

Result

GNP has proven to be an effective measure in solving various types of problems, such as stock trading [43], robot control [44], and data mining [45], etc. In [46], Li et al. tested GNP on the Tile World Problem using the training set (1) as in (Figure 1.1(1), with 60 steps for each agent, 300 individuals, 0.1 for the crossover rate, 0.01 for the mutation rate, and the following fitness function:

$$Fitness = \sum_{env=0}^{ENV} [(100 \times DT) + (20 \times (D_t - d_t)) + 3 \times (ST_{remain})][46] \qquad (2.1)$$

Where DT is the number of dropped tiles, ST is the remaining steps for the agents after pushing all the tiles into the holes, Dt is the initial distance from the tile t to the nearest hole, and dt is the final distance from tile t to the nearest hole after finishing. This experiment was able to achieve an average fitness value of 4171.4 with an average of 23/30 dropped tiles. The set-up of the experiments involved using 10

environment configurations. The only difference between the test set and the training set is the locations of the tiles. The algorithm's performance is characterized to have an average fitness of 2276.3 with 11.7/30 dropped tiles, and for the testing set with different locations for the holes, tiles, and obstacles, the average fitness was 547.3 with a 3.6/30 dropped tiles within 300000 number of fitness evaluation.

Strengths and Limitations

In the GNP algorithm, the chromosome oversized problem is tackled by using a network structure (graph) that enables multi-use nodes without end nodes to solve complex problems. Although the algorithm has been utilized to solve complex issues, more complex problems need more probabilities to cover all the rules. In this case, the number of nodes in the graph needs to increase, increasing the chromosome size. To overcome this problem, a new algorithm called Genetic Network Programming with Reinforcement Learning (GNP-RL) has been produced, which will be discussed in the next chapter.

2.5 (GNP-RL) Genetic Network Programming with Reinforcement Learning

2.5.1 Introduction

In 2007, Mabu et al. [6] combined learning with evolution in Genetic Networking Programming with Reinforcement Learning (GNP-RL) by adding some sub-nodes for each node, which allowed the RL to run in each graph to detect the best sub-node after the exploration and exploitation methods (see Figure 2.13). This algorithm succeeded in many applications, such as mobile robot behaviour [47] and the Stock trading model [16]. It also provided a good example when applied to multi-agent problems, such as the problem of Tile World, where the algorithm was applied by Mabu et al. [6]. The graph is an effective structure for finding shared rules that can make multiple agents work together effectively since the RL can give accumulative feedback from all the agents' activities. This is done by updating the Q-value for each node each time this node is visited by an agent using the ϵ-greedy technique [48].

2.5.2 Chromosome structure

The chromosome structure for the GNP-RL is a graph containing a number of nodes, each node is determined by these characteristics:

1. Node-id: Each node should have a unique ID.

2. Node-type: There are three types of nodes: start, judgment, and processing nodes.

3. Delay-time: The delay time is the time chosen for each node type to provide a theoretical value for node execution. Mabu et al. were the first to use it in 2007 [6]. This is a tool that calculates the number of steps taken by the agent during algorithm execution.

4. Q-value: Since each node has multiple sub-nodes, the agents were trained using reinforcement learning (Sarsa) to navigate the graph. Each sub-node is assigned a Q-value that starts at 0 during initialization. Whenever the agent visits a node, it selects one sub-node to execute. The method by which the agent chooses the sub-node will be discussed in the section (Evaluation and Selection) below.

5. Number of sub-nodes: Each node has a number of sub-nodes that are defined in the first population, and it could be changed during the evolutionary operations.

6. For each sub-node:

   a. ID: Each sub-node has a function or task (ID). This function should be included as one of the processing and judgment functions. If a node is a judgment node, all its sub-nodes should also be judgment nodes and vice versa.

   b. Connections: Each sub-node has connections that direct the agent to the next node. The number of connections for each sub-node is subject to its function.

2.5.3 Initialization of the First Population

Initially, each chromosome in the population has the same number of nodes. The nodes are of varying types and are connected randomly. Each node has a random number of sub-nodes with a predefined maximum sub-node number. Each sub-node has a random function, and it is initialized with a 0 as a Q-value.

2.5.4 Evaluation and Selection

Each agent begins at the start node and then follows the connections to visit the subsequent nodes. In order to proceed when visiting the node, it is necessary for the agent to choose one of the available sub-nodes. In order to choose a specific sub-node, the Sarsa method is employed with the use of an ϵ-greedy approach, which was first implemented by Sutton and Barto [48]. Exploration is when a sub-node is randomly selected with a 0.1 probability and 0.9 for exploitation that chooses the sub-node with the maximum Q-value. The Q-value will be updated using Equation 2.2 [6] after visiting

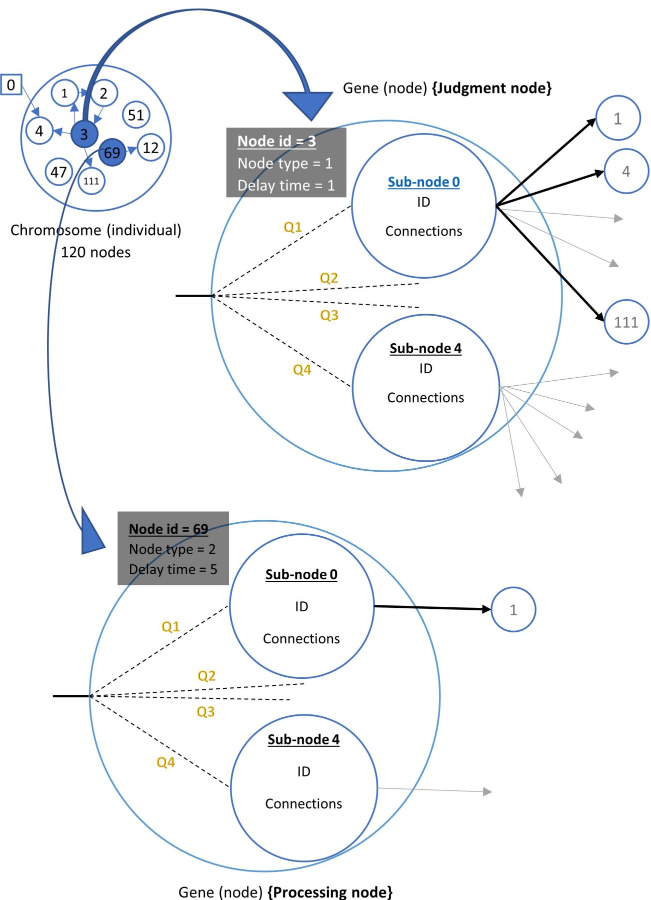


Figure 2.13: GNP-RL Chromosome Structure [17].

the node. Where Qip is the Q value for the visited sub-node, Qjq is the Q-value for the chosen sub-node in the next node, and α is a learning rate it could be any value from 0 to 1, and γ is a discount rate it is a value from 0 to 1. The Reward depends on the problem domain.

$$Q_{ip} = Q_{ip} + (\alpha * (Reward + (\gamma * Q_{jp}) - Q_{ip}))[6] \tag{2.2}$$

Finally, after finishing the simulation, the fitness function is calculated. The variation between the training phase and the testing phase is that the training phase implements the ϵ-greedy technique, balancing exploration and exploitation. In the testing phase, the agent selects the sub-node with the highest Q-value from the results of the training stage. The Q-values will no longer be updated during testing (see Figure 2.14 and Figure 2.15).

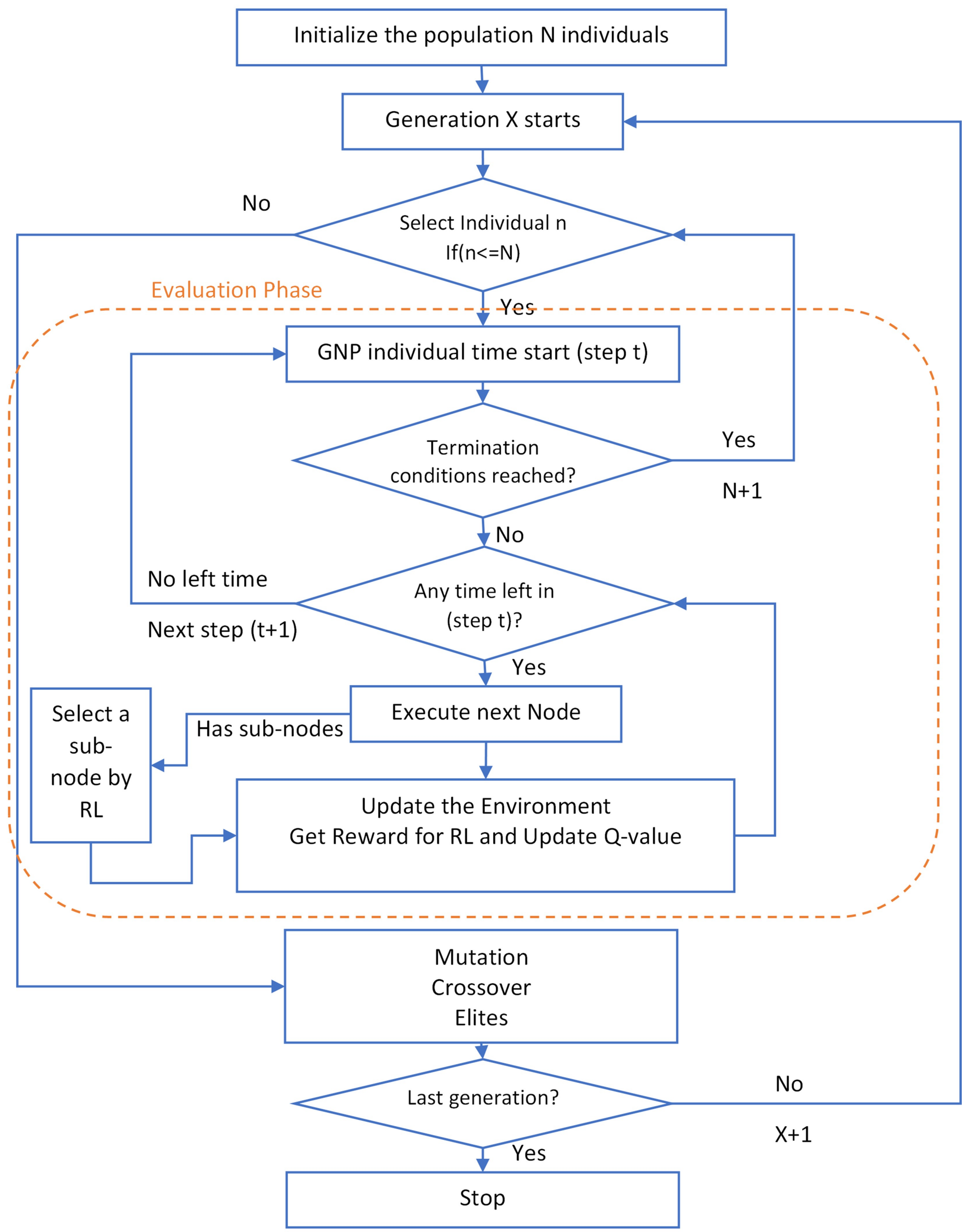


Figure 2.14: Genetic Network Programming with Reinforcement Learning algorithm (training phase) edited from [38].

2.5.5 Evolutionary Operations

Crossover

The crossover operation in the GNP-RL is similar to what happened in GNP. So, the algorithm will choose random nodes from each parent and cross them over together to

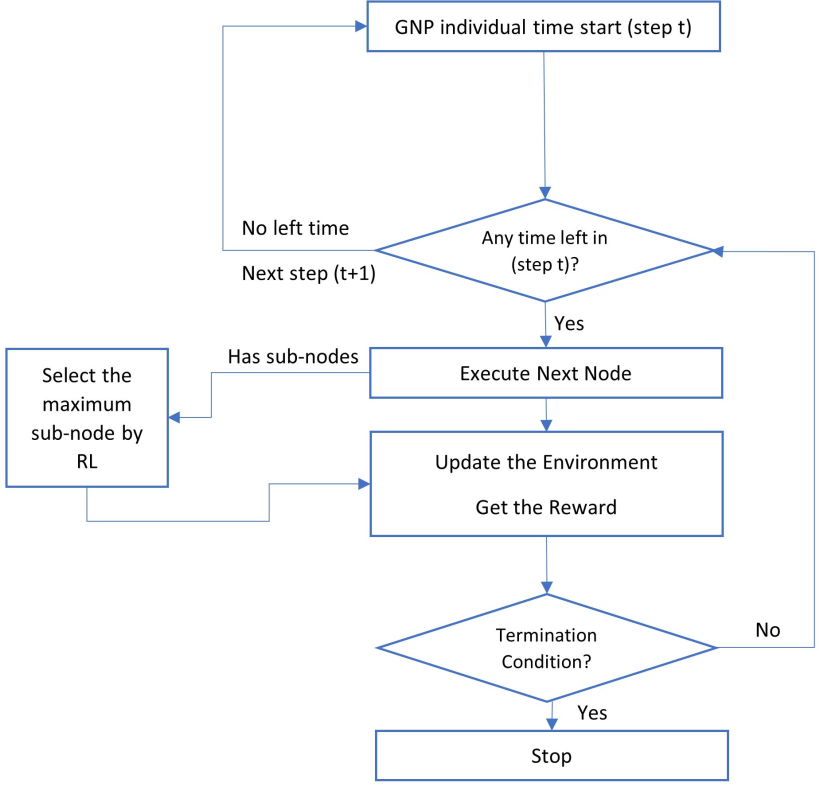


Figure 2.15: Genetic Network Programming with Reinforcement Learning algorithm (testing phase) edited from [38].

create new offspring, taking into consideration that the nodes will be moved with their sub-nodes.

Mutation

The mutation operation in the GNP-RL is similar to what is comprised in GNP, taking into consideration that the mutation could also occur on the sub-nodes. So, it can change the sub-node ID or the number of sub-nodes by adding or deleting sub-nodes from a chosen node.

2.5.6 Hyperparameters

The same hyperparameters that affect the GNP also affect the GNP-RL, adding to them:

- The parameters used in the learning phase of GNP-RL (the step size parameter α, discount rate γ, and the value of ϵ) are set as follows: α is set to 0.9 to quickly

find solutions, γ is set to 0.9 to sufficiently consider future rewards and the value of ϵ is set at 0.1. This value takes into account the balance between exploiting known nodes and exploring unknown nodes. Low epsilon programs fall into local minima, while high epsilon programs take too many random nodes [6].

- Reward: GNP-RL has the ability to modify its programs incrementally by taking into account the rewards received during the execution of a task. When an agent takes a positive action (reward) in a given state, it is reinforced and more likely to be repeated when the state is revisited [6].

2.5.7 Summary

Results

In [6], the GNP-RL has been tested using a Tile World Problem. The experiments were applied to an environment with 30 tiles and 30 holes. It used 300 individuals (120 Crossover, 175 Mutation, 5 Elites), 0.1 crossover rate, 0.01 mutation rate, α is 0.9, γ is 0.9, ϵ is 0.1, maximum of 4 sub-nodes, and 60 steps for each agent. The fitness function was the number of dropped tiles, the reward was 1 when the agent dropped a tile into a hole. The algorithm was able to achieve an average of 21.23/30 dropped tiles after 5000 generations.

Strengths and Limitations

The GNP-RL algorithm is a combination of genetic network programming and reinforcement learning Sarsa. It works by dividing each node into several sub-nodes and then training the agent on the graph using RL to select the best sub-node based on the maximum Q-value. This approach allows the chromosome to contain more nodes, enabling the solution of complex problems while keeping the chromosome structure simple. GNP-RL offers online learning which allows for incremental program modifications based on rewards obtained during task execution. This approach reinforces good actions with positive rewards and increases the probability of their execution when the agent visits the same state again. GNP-RL combines a diversified search via GNP with an intensified search via RL. Evolution creates rough structures, while RL determines the optimal path. The diversified search for evolution can alter programs significantly, helping them escape local minima. While RL's immediate reward-based execution makes intensified search more efficient [49] [50]. GNP-RL continuously searches for improved solutions during task execution (judgment and processing) and the evolutional operations executed after the task. It utilizes both the diverse search ability of evolution and the intensive search ability of learning.

Although the GNP-RL is a strong algorithm and succeeds in solving many problems that the previous ones could not, it has some limitations. Firstly, within the same simulation on the same chromosome, every time any agent visited a node, it could use a different function type. So, within the same evaluation, the node could be used with a different type of function, which makes the testing result for the chromosome different from the training result in the same training environment because of the randomization on the exploration in the Reinforcement Learning. Also, we found from our last approach [17] (under Chapter 9) that the GNP-RL may suffer from overfitting. Adding to that, it is a non-systematic algorithm as it could lose the optimal solution because of incorrectly choosing the elite depending on the training results that used exploration and exploitation. Finally, even if the GNP-RL has been implemented to increase the available possible solutions, as a consequence, that also means the size of the graph is increased as the algorithm needs to save the nodes with all their sub-nodes, including their functions and connections. In effect, that also needs more memory size.

This algorithm outperformed the GNP [6], but it still needs improvement in solving the Tile World Problem. To conclude, GNP-RL needs further improvement to tackle the complex problem of the Tile World application. Over the years, several techniques and mechanisms have been added to enhance GNP-RL. Mabu has worked on this algorithm and improved it by incorporating Distributed and Variable-Sized GNP chromosomes. The next section will provide a detailed explanation of this approach.

2.6 (DGNP-RL) Distributed Genetic Network Programming and (VSGNP-RL) Variable-sized Genetic Network Programming

2.6.1 Introduction

The divided structure method with the GNP-RL was first proposed by Yang et al. [16]. He demonstrated the effectiveness of this method using a model for trading stocks. This method breaks down the problem into smaller tasks to simplify the chromosome structure and enhance the algorithm's functionality. Distributed GNP-RL yielded superior results in the stock trading model compared to GNP-RL. Following that, in 2014, Mabu et al. proposed a new version of the Distributed GNP-RL called the variable-sized genetic networking program with reinforcement learning (VSGNP-RL), and he applied it to the Tile World problem [9]. In this version, the distributed structure could have a sub-program with different sizes. Each sub-program may have a different number of nodes (genes). In this algorithm, Mabu showed how this mechanism improved the results of the Tile World problem.

2.6.2 Chromosome structure

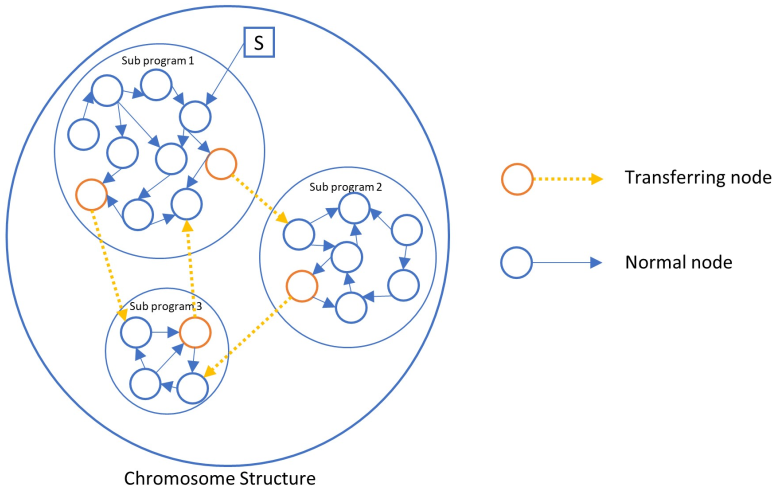


Figure 2.16: VSGNP-RL Phenotype Chromosome Structure. The orange coloured nodes identify the nodes that directly connect to another subprogram modified form [9].

The structure of the individual resembles that of GNP-RL, except for its division into sub-programs. There are two types of nodes: transfer nodes and normal nodes. Transfer nodes lead to nodes in a different sub-program, while normal nodes lead to nodes within the same sub-program. The nodes are distributed among the sub-programs. (See Figure 2.16)

2.6.3 Initialization of the First Population

The same technique used to initialize the first population on GNP-RL is also used here with VS-GNPRL with some additions. Each node has two additional features: sub-program number (AF) and node connection type (TR). Sub-program number (AF) detects the number of sub-programs that this node belongs to. Node connection type (TR) indicates the type of the connection if it is a transfer node that directs to a node from another sub-program or a normal node which leads to a node from the same sub-program. In the first generation, every subprogram has an equal number of nodes. However, the number of nodes in each subprogram is altered during mutation.

2.6.4 Evaluation and Selection

The mechanism employed by GNP-RL to initialize the first population remains unchanged in VS-GNP-RL.

2.6.5 Evolutionary Operations

Crossover

The crossover operation selects two parents using one of the selection methods, then randomly chooses two random nodes, one from each parent with a probability Pc to change them. The selected nodes should share the same type and the same sub-program number. This implies that if the random node that has been selected from parent 1 is related to sub-program 2, the other node that will be chosen from parent 2 should belong to sub-program 2, too. To maintain a balance of node types in each subprogram, for each sub-program, a detected number of processing and judgment nodes should be crossed over. As shown in Figure 2.17, for example, every sub-program should cross three judgment nodes and one processing node from each parent.

Mutation

In this implementation, two types of mutation are utilized, as in Figure 2.18: The mutation operation in VS-GNPRL is the operation that makes the sub-program change its sizes. Two types of mutation lead to this feature:

1. Changing the connection type (TR), when the algorithm changes the type of the connection, it will change it from transferring to normal or vice versa. When the connection type is changed, the connection of this node should be taken into consideration, as if the connection type changed from normal to transfer, all the connections in this node should be changed to direct to nodes from another sub-program and vice versa.

2. Changing the sub-program number that this node belongs to, taking into consideration when changing the sub-program, changing the connections too. Depending on the type of the connection, if it is normal or transfer, the connection should be adjusted.

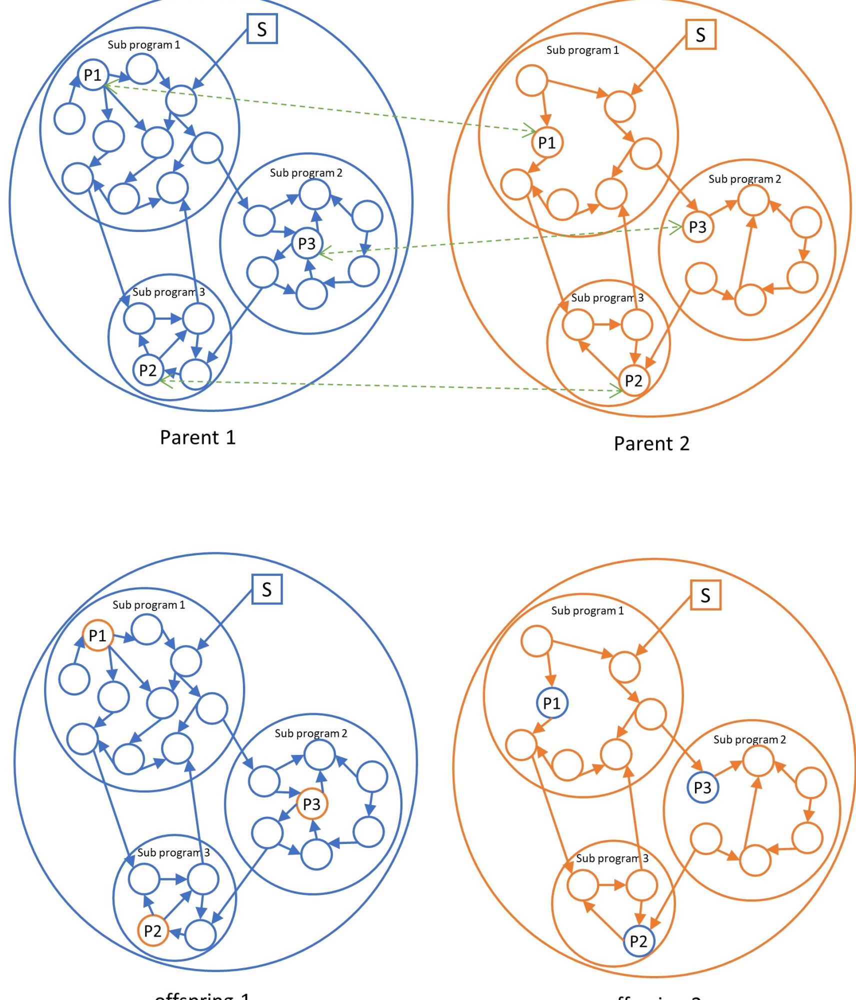


Figure 2.17: VSGNP-RL Crossover the different colours are the nodes after applying crossover between the parents. The nodes crossed over with another same type of node from the same subprogram [9].

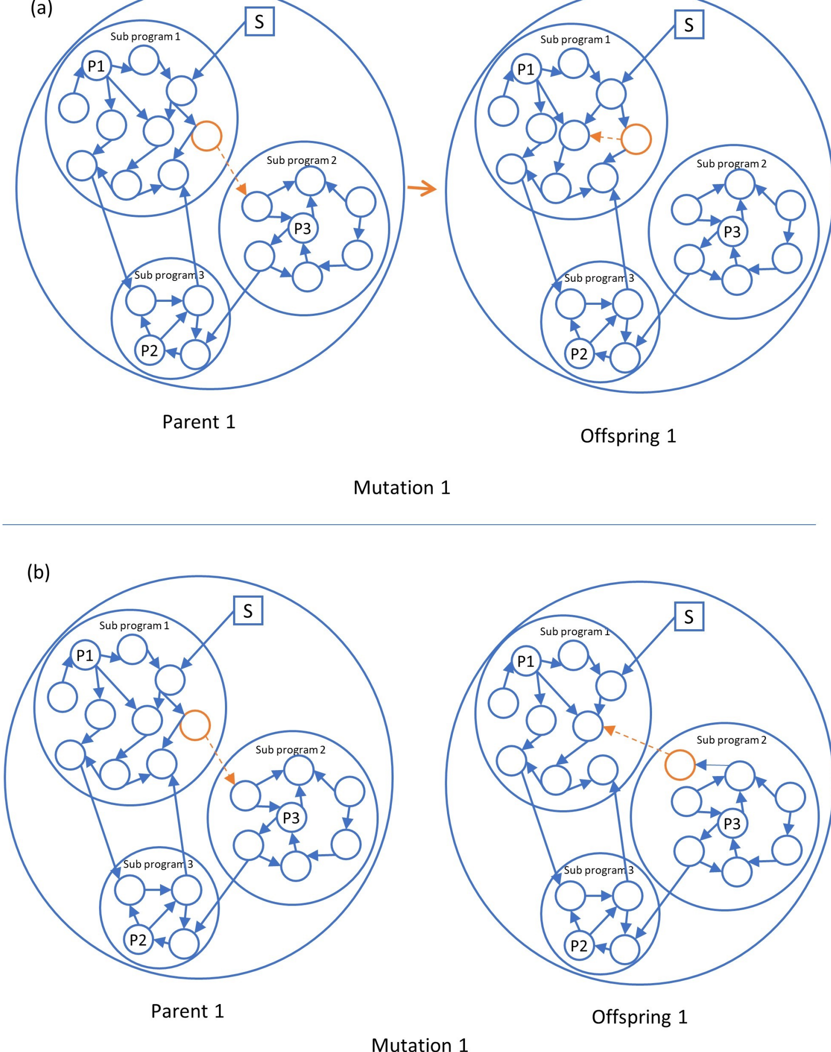


Figure 2.18: VSGNP-RL Mutation. (a) The Mutation here allows changing the type of the connection from normal to transfer or vice versa. (b) The Mutation here allows changing the subprogram that the node belongs to, tacking into account changing the connections also [9].

2.6.6 Hyperparameters

The new parameters in the VS-GNP-RL are:

1. The sub-program size: The subprograms in VS-DGNP have the ability to exchange nodes with other subprograms, which helps to optimize their complexity based on the specific problem at hand. This allows for the efficient creation of rules. In contrast, DGNP has a fixed number of nodes in each subprogram and must create action rules within the given size of the structure. It is important to carefully determine the parameters for evolution in order to avoid excessively large changes to the structure of the program, which can destroy good building blocks [9].

2. Connection Type: Internal connections branch from normal nodes to other nodes in the same subprogram, while external connections link transferring nodes to nodes in other subprograms [9].

2.6.7 Summary

Results

In [9], Mabu et al. tested VS-GNP-RL on the Tile World Problem using the training set (1) as in 1.1, with 60 steps for each agent, 300 individuals (120 Crossover, 175 Mutation, 5 Elites), 0.1 crossover rate, 0.01 mutation rate, α is 0.9, γ is 0.9, ϵ is 0.1, maximum of 3 sub-nodes, 0.1 for the crossover rate, 0.01 for the mutation rate, and this fitness function:

$$Fitness = \sum_{env=0}^{ENV} [(100 \times DT) + (20 \times (D_t - d_t)) + (ST_{remain})][9] \tag{2.3}$$

Where DT is the number of dropped tiles, ST is the remaining steps for the agents after pushing all the tiles into the holes, Dt is the initial distance from the tile t to the nearest hole, and dt is the final distance from tile t to the nearest hole after finishing. The reward was 1 when the agent dropped a tile into a hole. This experiment was able to achieve an average fitness value of 5,333 for VS-GNP-RL and 4,818 for DGNP-RL within 5000 generations in the training stage. For the testing phase, the environment in (Figure 1.2) has been used as a testing set with 100 steps for each agent, the result was an average fitness value of 2,010 for VS-GNP-RL, and 1424 for DGNP-RL.

Strengths and Limitations

VSGNP-RL improves upon GNP-RL by dividing the chromosome into subprograms for simpler problem-solving. The variable-sized structure can learn general action rules

in graph structures to adapt to various situations. Determining the parameters for evolution is a crucial task that must be executed with care to prevent significant modifications to the program structure, which could eradicate important building blocks. Nevertheless, the program's adaptable size can make this process challenging. Furthermore, there is a need to investigate the learning or evolution algorithm of the parameters, which is still an unresolved issue. Moreover, this algorithm could not fully solve the Tile World problem [17] [14], as it succeeded on getting 29/30 dropped tiles on the training set, and 13/30 on the testing set. So, in the next chapter, we will discuss a new improvement of the GNP-RL algorithm that we introduced in our previous work called GNP-RL -CC-OS-TP [17] [14].

2.7 (GNP-RL-CC-OS-TP) Genetic Network Programming with Reinforcement Learning, Constraint Conformance, Optimal Search, and Task Prioritization (Review and new application)

2.7.1 Introduction

In our last approach [17], we added three new techniques to the GNP-RL, which are Constraint conformance, optimal search, and task prioritization. These three approaches have improved the performance of GNP-RL in multi-agent systems by learning how to avoid unwanted situations when using the punishment signals with the agent's learning strategies (constraint conformance), giving accurate paths when using the A* algorithm (Optimal search), and arranging the tasks for each agent by using a scalar reward and punishment scheme (task prioritization) [17] [14]. This approach has succeeded in the Tile World problem with 100% training accuracy. It also achieved 83.3% testing accuracy.

The Constraint Conformance

Every problem comes with certain conditions or limitations that restrict the behaviour of an agent. For instance, agent actions must prevent collisions between agents, avoid hazardous paths or walls, avoid placing objects in undesired locations, and so on. To apply the constraint conformance mechanism to any real-world problem, we must first identify any conflicting situations that exist within the problem domain and integrate the appropriate functions for detecting those conflicts. Next, we need to train the agents to avoid all the undesired cases by penalizing them every time their actions create one of those situations. We impose the penalty through penalties when updating the Q-values in RL. In this algorithm, the processing node that executes the action has two

connections, unlike GNP-RL where only one connection leads to the next node after processing. The first connection is used when there is no conflict, and the second one is used when there is a conflict. For example, in the Tile World Problem, the node that has a (Go forward) function has two connections. The first connection is used when the tile is pushed to a safe location, but the next connection is used when the following location is a trapped vertex.

When executing the algorithm, the agents start carrying out the nodes from the graph. If the agent encounters a (Trapped node) Processing node, before performing the action, the agent verifies the next step:

If the location is trapped, the algorithm will take three actions:

1. The algorithm is designed to stop the agent from pushing the tile.
2. Punishing the agent means subtracting 1 from the reward when updating the Q-value.
3. To proceed to the next node, select the second connection in the trapped node.

In the case where the new location is safe:

1. The algorithm will execute the 'go forward' action.
2. If the agent moves the tile closer to the hole, reward the agent with a value of 1 when updating the Q-value.
3. To proceed to the next node, select the first connection from the trapped node.

The Task Prioritization

To encourage agents to prioritize their tasks, it is recommended to provide them with motivational or promotional rewards upon completion of each task. Each task should be assigned a specific reward that is proportionate to its level of importance or priority. The higher the level of importance, the greater the reward should be. By implementing this approach, agents will be incentivized to complete the most important tasks first in order to earn more rewards before moving on to less important tasks.

In the Tile World Problem, during the evaluation stage, we use the "Own Tile Priority" technique to prioritize task execution. The method's implementation is split into two parts. The first step is determining which tile and hole is nearest to each agent using the A* algorithm before the algorithm starts. Each agent has its own tile and hole, determined by the shortest paths discovered through the A* algorithm. In order to teach the agent how to prioritize following its own tiles and holes, the rewards assigned in the Q-value update formula vary based on which tiles the agent has moved and to each hole. The rewards and punishments can be summarized as follows. When

the agent follows its prioritization when performing the actions, it will get a reward. Each time the agent does not follow the prioritization, it will get a lower reward.

The Optimal Search

Using the A* algorithm within the GNP-RL graph has produced more accurate directions for judgment nodes which request directions. This method has been implemented after realizing that sometimes the answer provides the direction, regardless of obstacles or agents in the way. Using A*, the algorithm finds the shortest path to the goal and provides the direction to the first vertex as the output. For instance, if the judgment node is to determine the direction to the nearest tile, the A* algorithm sets the agent as the starting point and tiles as the goals. By determining the shortest paths to available tiles, the nearest tile will have the smallest cost path, and the direction will be from the agent to the first point in the shortest path.

It is not enough to use the optimal search component alone to solve a problem. To solve a problem, the GNP-RL needs to develop a meta-level reasoning strategy that selectively combines a number of functions from a given library of functions specific to the problem domain. For instance, when asking about the shortest path using an optimal search algorithm like A*, the proposed framework will not provide the agent with a complete path. Instead, GNP-RL will identify the next action for the agent by considering several questions that A* can answer, such as where the nearest tile, hole or the second nearest tile is located. A* can answer these questions by determining the shortest paths to each query and providing the direction to the first waypoint along that path. It is crucial to note that A* does not provide the ultimate next action for the agent. GNP cleverly combines the answers to these queries using multiple judgement nodes.

By combining optimal search with constraint conformance and task prioritization, along with the use of GNP-RL, we can achieve a meta-level reasoning strategy. In this framework, A* is used for lower-level decision-making, while GNP-RL is at the top of the hierarchy for decision-making.

2.7.2 Chromosome structure

The chromosome structure for this approach is similar to the one that is used with GNP-RL but with two connections for the processing nodes that tack an action (Go Forward Node). In the normal structure for the GNP-RL, there is just one connection for all the processing nodes, while in this approach, there is one connection for each processing node except the processing node that makes an action such as (Go Forward), which has two connections. The algorithm makes the action and uses the first connection when there is no conflict, whereas, in case the action leads to a conflict when it happened it

will stop making the action and will use the second connection that will lead the agent to another node (See Figure 2.19).

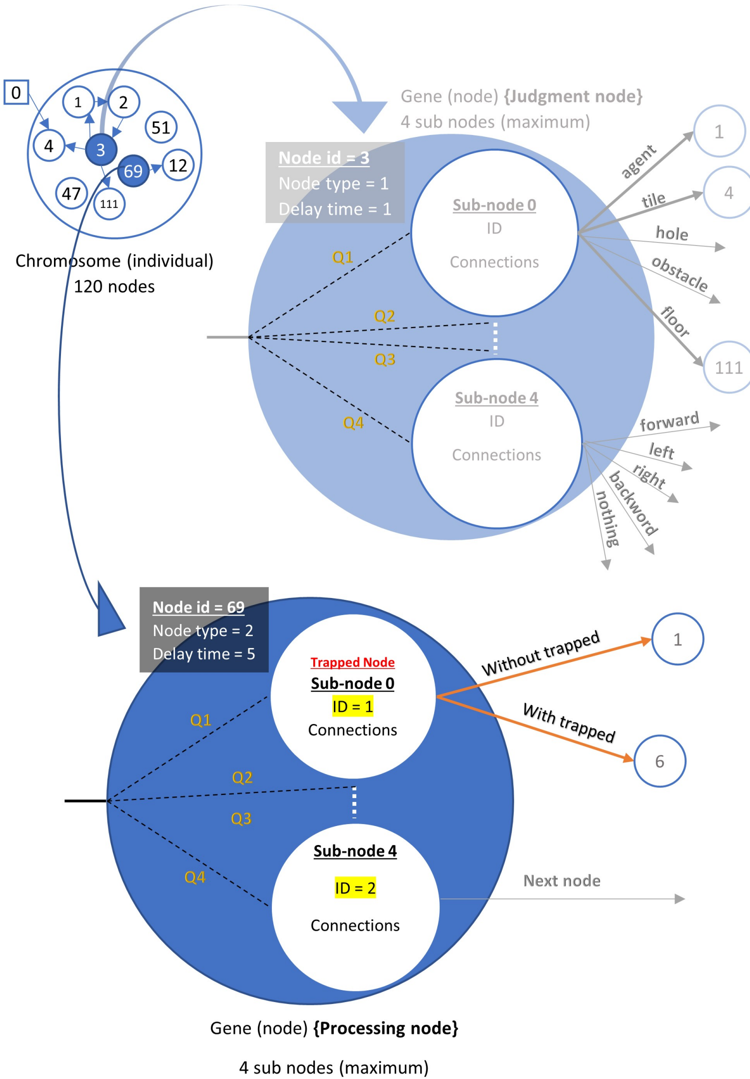


Figure 2.19: CC-OS-TP-GNP-RL Chromosome Structure [17] [14].

2.7.3 Initialization of the First Population

In order to create the initial group of individuals, we need to generate the first population, which has the same features as the GNP-RL. In addition to that, the two connections for the trapped nodes are chosen randomly for the first population.

2.7.4 Evaluation and Selection

For this work, we made alterations to the structure of the chromosome to identify and manage situations where conflicts arise (See Figure 2.19). The new structure utilizes two output pathways: one for detecting conflicts and another for all other cases. To evaluate the CC-OS-TP-GNP-RL, firstly, it is important to determine all the inquiries and actions that the agent may utilize and then any potential conflicts that may arise within the problem domain. Secondly, utilize the optimal search algorithm to prioritize the tasks assigned to each agent. After that, to train the agent to follow their own task first, the Reward has different values depending on scalable rewards and punishments. Thirdly, the A* algorithm was used to identify the nearest object and the shortest path to this object. Using the A* algorithm within the GNP-RL graph has resulted in more precise directions for nodes requesting directions. This method was implemented because sometimes the answer provides directions without considering obstacles or agents. Using the A* algorithm, it finds the shortest path to the goal and returns the direction to the first vertex in this path as the answer. (See Pseudocode 1).

2.7.5 Evolutionary Operations

The same crossover and mutation used with basic GNP-RL are used here, too, with CC-OS-TP-GNP-RL, considering the trapped node that could be crossed over with any node from another parent. For the mutation, both connections on the trapped node could also be changed randomly.

2.7.6 Hyperparameters

GNP-RL-CC-OS-TP has the same parameters as the GNP-RL, adding to them: Scalable rewards and punishments: this parameter helps the agents consider the priority when performing the tasks. So, choosing suitable scalable rewards and punishments is very important.

2.7.7 Summary

Results

In [17], the GNP-RL-CC-OS-TP has been tested using a Tile World Problem. The experiments were applied to an environment similar to Figure 1.1. It used 300 individuals (120 Crossover, 175 Mutation, 5 Elites), 0.1 crossover rate, 0.01 mutation rate, α is 0.9, γ is 0.9, ϵ starts at 0.9 and then decreased by 0.1 every 10 generation until reached 0.1 for the rest of the simulation, maximum of 4 sub-nodes, and 60 steps for each agent. The fitness function was:

Algorithm 1 General Node Execution [14]

```
 1: Inputs: current locations of Agents L, Set of actions A[], Set of queries Q[], Set of
    constraints C[], Set of priorities P[], Set of Rewards R[], Set of States S[], objects
    in the environment O[]
 2: Output: nextNode
 3: if NodeType is an Action Node then
 4:     With probabilistic selection a in A[] based on Q-value
 5:     if a causes any conflict c in C[] then
 6:         Reward ← -low reward from R[]
 7:         NextNode ← first_connection                    ▷ deal with conflict
 8:     else
 9:         Apply action a, update state s
10:         if ((a,s) has priority p in P[]) then
11:             r = R[p]                 ▷ includes both punishments and rewards
12:         end if
13:         NextNode ← second_connection                    ▷ without conflict
14:     end if
15: else if NodeType is a Query Node then
16:     With probabilistic selection q in Q[] based on Q-value
17:     if q is a directional query then
18:         path = A*(q,L,O)              ▷ use the A* algorithm to answer q
19:         response = q(Direction(first_waypoint(path)))
20:     else
21:         response = q( )
22:     end if
23:     NextNode ← connection(response)
24: end if
25: Update Q-value(r)
26: receivedo the NextNode
```

$$Fitness = \sum_{env=0}^{ENV} [(1000 \times D_{tile}) + ((D_{distance})) + (T_{remain})][14] \quad (2.4)$$

Where Dtile represents the number of tiles that were successfully dropped in the hole for each environment, Tremain is the remaining steps from each agent when the simulation is finished, and Ddistance refers to the number of times the agent pushes the tile nearer to the hole, and it is calculated as below:

- 2 points → when the distance between the tile and the hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the distance between the tile and the hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the distance between the tile and all holes after pushing the tile is more than the distance between them before the push.

When updating the Q-value, the Equation 2.2 from [6] was used and the Reward was calculated as below:

- 4 points → when the agent pushes the nearest tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes the nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 10 points → when the agent pushes the tile to a hole.

The algorithm was able to achieve an average of 30/30 dropped tiles in the training stage. After 5000 generation of training, the algorithm was able to get 26/30 dropped tiles when testing it in testing set (2) (Figure 1.1(2)), and 29/30 dropped tiles in testing set (3) (Figure 1.1(3)).

Strengths and Limitations

Using A* as a judgment node inside the graph leads to an increase in the feedback accuracy from the environment leading to better decisions for the agents to take. Using the punishment in updating the Q-value when the agents make a wrong decision and then directing the agent to another decision leads to a quick increase in the agents' performance and prevents the agent from making the wrong decision again. Choosing scalable rewards and punishments when updating the Q-value makes the agents consider the priority when performing the tasks.

Although using A* as a judgment node increases the performance, this technique will take a long time when the environment is big, and the number of objects is greater. The constraint conformance technique is effective in detecting the conflict and preventing the agent from taking the action that leads to this conflict, but it is not solving the function node in the graph that led to this conflict. This algorithm outperformed the GNP-RL, but it still has the same limitation as the GNP-RL as it's an extension from it. To conclude, it needs further improvement to tackle complex problems such as the Tile World application.

2.8 Diversity in Evolutionary Algorithms

A common issue with evolutionary algorithms is that they often converge to local optima. This premature convergence occurs due to certain algorithmic features, particularly selection and excessive gene flow between population members. Diversity decreases over time, leading to a population of similar individuals, making it challenging for the algorithm to escape the local optimum. Lowering the selection pressure is rarely an option due to its slow convergence speed. Another factor that affects the spread of genes in a population is the level of gene flow, which is often determined by the population structure. In simple evolutionary algorithms, any individual can mate with any other individual, leading to a rapid spread of genes throughout the population. However, this can also result in a decrease in genetic diversity and a stagnation of fitness levels [51].

Evolutionary algorithms have traditionally employed diversity measures for analysis purposes only, rather than utilizing them to guide the algorithms. In various studies,

diversity measures have been employed to regulate Evolutionary Algorithms (EAs). The Diversity-Control-Oriented Genetic Algorithm [52] employs a diversity measure based on Hamming distance to determine the survival probability of individuals. The individuals with a low Hamming distance from the current best individual are assigned low survival probabilities, thereby ensuring the preservation of diversity during the selection process. An alternative approach to genetic algorithms is the Shifting-Balance Genetic Algorithm [53]. This algorithm calculates a containment factor that measures the Hamming distance between two sub-populations. The distance is computed by comparing each member of population A with all members of population B. The containment factor determines the proportion of individuals selected for their fitness and those chosen to increase the distance between the two populations. Another approach uses specialized diversity measures to create a sub-population from a subset of the population called Forking GA [54]. There are two types of Forking GA - one operates on the genotype, whereas the other is based on distances in the search space (on the phenotype).

Langdon conducted a study on the stack problem [55], examining the impact of the crossover operator on variety. He observed that genetic programming tends to lose the ability to enhance fitness after 20-30 generations, which is most likely due to crossover causing a loss of variety. During his research, Langdon noticed that in the stack problem, runs with better fitness allowed for crossover to produce a larger number of fitter and non-duplicate offspring than their parents. Eshelman and Schaffer [56] explored the benefits of pairwise mating in genetic algorithms. They chose individuals for replacement and recombination based on Hamming distances, which improved selection strategies for genetic algorithms over hill-climbing. Ryan's "Pygmie" algorithm [57] was created to address issues with premature convergence and elitism in small populations when evolving minimal sorting networks. The algorithm works by building two lists based on fitness and length to facilitate selection for reproduction. By using these lists, Ryan's algorithm was able to maintain more diversity, prevent premature convergence, and use simple measures to promote diversity. Damia et al. introduced a new technique in 2021 to prevent loss of diversity [58]. This method involves calculating the similarity between the genes of chromosomes and the diversity between each generation and the previous one. Based on these calculations, the selection, crossover, and mutation rates are adjusted in each generation. In this thesis, we implemented appropriate methods for each proposed algorithm to address the problem of loss of diversity.

Chapter 3

Experiments with Recent Variants of Genetic Networking Programming

3.1 Introduction

This chapter discusses the general methodology used for solving the three problems (the original Tile World problem, the Heavy Tile World, and the prey-and-predator problem) when applying the variants of the GNP algorithm. We chose the Tile World Problem because it is a multi-agent planning and collaboration problem in dynamic environments that uses a sparse training set, which has not been 100% solved previously using Genetic Network Programming Algorithms. We also implemented the Heavy Tile World Problem to increase the problem's difficulty in terms of collaboration requirements, which will be used to test the variable-sized GNP-RL. Moreover, we tested the algorithm on the Prey-and-Predator Problem to examine the algorithm's generalization capability. In 2019, we implemented GNP-RL-CC-OS-TP, as an extension to GNP-RL; in this chapter, we implemented a new approach that applies the three techniques that have been used in [17] on the basic GNP and called it (GNP-CC-OS-TP), then compared its results with the basic GNP, the basic GNP-RL, and GNP-RL-CC-OS-TP using the three presented problems.

This chapter has been added to this thesis to measure the performance of the algorithms (GNP and GNP-RL) when adding the three techniques from [17] and [14] to them:

1. Optimal Search: utilised as part of a judgment node inside the chromosome graph.
2. Constraint conformance: when executing the processing node Move forward, it will check if it accrues a conflict (when pushing a tile, check if it will be pushed

to a trapped location that will not allow for it to be pushed again anywhere else). In this case, the algorithm will not apply the action; instead, it will choose the second connection in this node to change the agent's direction in the graph.

3. Task prioritization:

   a. In GNP, a scalable distance will be added to the fitness function to increase the fitness value each time the agent performs the task in correct prioritization.

   b. In GNP-RL, a scalable reward will be used to update the Q-value each time the agent performs the task in correct prioritization.

All the details of applying these techniques and their results will be explained in this chapter.

3.2 GNP (Genetic Networking Programming) and VS-GNP (Variable-sized Genetic Networking Programming)

GNP Individual

A GNP individual or a chromosome is represented as a directed graph with a fixed number of nodes whose functionalities are pre-defined in a library of functions. G = (set of nodes, library of functions)

Types of Nodes

Three types of nodes are considered, namely a start, processing or judgement node.

- A start node directs the agent to the first node to execute in the graph.
- A judgement node performs the role of querying the exploratory world and utilizing the sensory information that answers them. After a judgement node executes a query, control is directed to the next node associated with the answer to the query.
- Lastly, a processing node executes an action; it may represent a robot movement for a control system, as an example.

The nodes in the graph are connected together to formulate a sequence of processes that are invoked according to some conditions represented by the judgement nodes. The conditional transitioning of processes in the graph may form a loop that can

Table 3.1: Node structure.

<table>
<tr><th>Property</th><th colspan="2">Description</th></tr>
<tr><td>1. Node ID</td><td colspan="2">Unique node ID</td></tr>
<tr><td rowspan="4">2. Node Type</td><td colspan="2">One of the three possible node types:</td></tr>
<tr><td colspan="2">Start Node (SN)</td></tr>
<tr><td colspan="2">Judgement Node (JN)</td></tr>
<tr><td colspan="2">Processing Node (PN)</td></tr>
<tr><td>3. Number of sub-nodes</td><td colspan="2">In the GNP, the node is not divided into sub-nodes.
So, there are no sub-nodes in the GNP graph's nodes.</td></tr>
<tr><td>4. Function ID</td><td colspan="2">each node has a function definition that is indexed in the library of functions using its Function ID.</td></tr>
<tr><td>5. Set of connections</td><td colspan="2">each node has 1 or more connections that direct the agent to the next node.</td></tr>
<tr><td rowspan="5">6. Delay time</td><td colspan="2">Used to represent the number of steps taken by the agent after executing a node. An agent is stopped once the available number of steps for the agent is already used.
Each agent is allowed to take a maximum of predefined steps to solve the problem. In the literature, the number of steps is also referred to as the delay time, as each step incurs some delay.
Moreover, each step is counted as being equivalent to a number of micro steps. Depending on the node type executed, the number of micro steps taken varies.</td></tr>
<tr><td>Unit</td><td>Description</td></tr>
<tr><td>Single step</td><td>Each step is equivalent to 8 micro steps</td></tr>
<tr><td>Judgement node execution</td><td>Each judgement node execution is equivalent to 1 micro step.</td></tr>
<tr><td>Processing node execution</td><td>Each processing node execution is equivalent to 5 micro steps.</td></tr>
</table>

be effectively likened to having reusable modular functions in the graph. The node structure is shown in Table 3.1

For each problem domain that we have considered, we have explicitly defined a set of judgement and processing nodes, connections, as well as delay time, number of sub-nodes and number of sub-programs.

3.2.1 Tile World problem (Review)

For the Tile World problem, we used the parameters utilized by Mabu et al. in [6]. The parameters are shown in table 3.2.

Table 3.2: Parameters for the Tile World in [6].

Problem Domain	**Tile World Problem with GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent - Tile
	J2	Judge what is in back position (JB)	- Hole
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward
	J6	Direction to the nearest Hole (HD)?	- Left
	J7	Direction to the second nearest Tile (THD)?	- Right
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	- None
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the ten environments one by one (as in Figure 1.1(1)). The three agents start working from the first node in the graph and follow the directions (connections) until all the three tiles drop into the three Holes, or if the available number of steps is finished, in this case, the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 2.4 from [14].

Where D_{tile} is the number of dropped tiles in holes, T_{remain} is the remaining steps for the agents after the simulation finished, $D_{distance}$ is the total points that the agents spend in the simulation, and it is calculated as below:

- 2 points → when the distance between the tile and the hole after pushing the tile is less than the distance between them before the push.

- 0 point → when the distance between the tile and the hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the distance between the tile and all holes after pushing the tile is more than the distance between them before the push.

Results

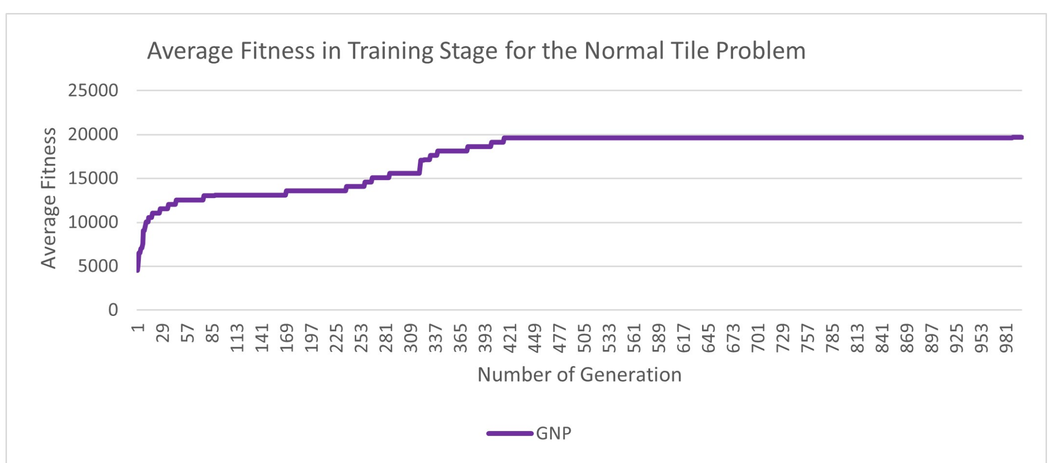


Figure 3.1: Average Fitness in Training and Testing Stage for the Normal Tile Problem using GNP.

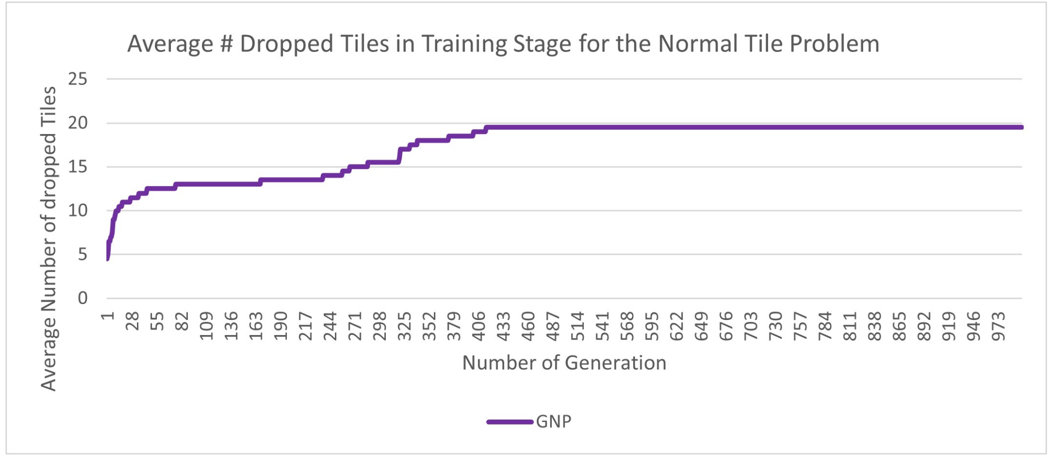


Figure 3.2: Average number of Tiles in Training Stage for the Normal Tile Problem using GNP.

Figure 3.1 and Figure 3.2 show the average fitness and number of dropped tiles training results for three experiments when applying GNP on the Tile World Problem for 1000 generations, which was the same as the testing results. The GNP algorithm could not reach the top training results within the first 1000 generations. Table 3.3

Table 3.3: The GNP results on the Tile World Problem

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result (*)	# evaluation until get the best chromosome (**)	# generation for the best chromosome (***)
GNP	17475	19672	4515	21/30 S1 02/30 S2 08/30 S3	-	282300	940

illustrates the mean, maximum and minimum fitness values for the average results, which were 17475, 19672, and 4515, respectively. The maximum testing results were (21/30), (2/30), and (8/30) for the training set (1) in Figure 1.1 (1), testing set (2), and testing set (3), respectively. This result was achieved in the generation number 940.

3.2.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed)

To increase the difficulty of the Tile World problem, we change one of the tiles in each environment to be a heavy tile that needs two agents to push it at the same time, as explained in Chapter 1.

To apply GNP to the Heavy Tile World problem, we have made some changes to it. The Judgment node number 8 has been changed from (giving the direction to the second nearest tile) to (checking the type of the nearest tile is normal or heavy).

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the ten environments one by one (as in Figure 1.2(1)). The three agents start working from the first node in the graph and follow the directions (connections) until all three Tiles drop into the three holes or if the available number of steps is finished, in which case the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 2.4.The distance $D_{distance}$ is calculated the same way used in section 3.2.1.

Table 3.4: Parameters for the Tile World with Heavy Tile.

Problem Domain	**Tile World Problem with Heavy Tile Using one sub-program with GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile**
	J2	Judge what is in back position (JB)	- Hole
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left
	J6	Direction to the nearest Hole (HD)?	- Right
	J7	Direction to the second nearest Tile (THD)?	- None
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal, Heavy		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Results

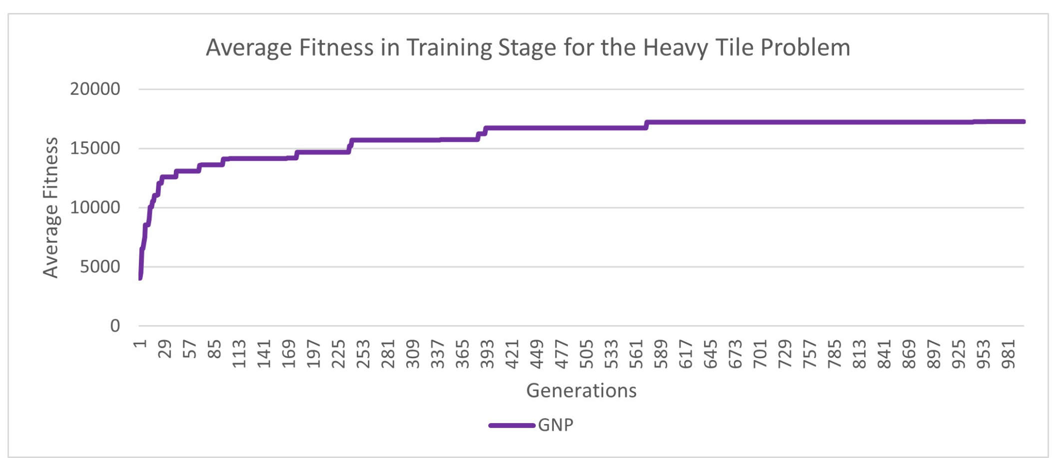


Figure 3.3: Average Fitness in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP.

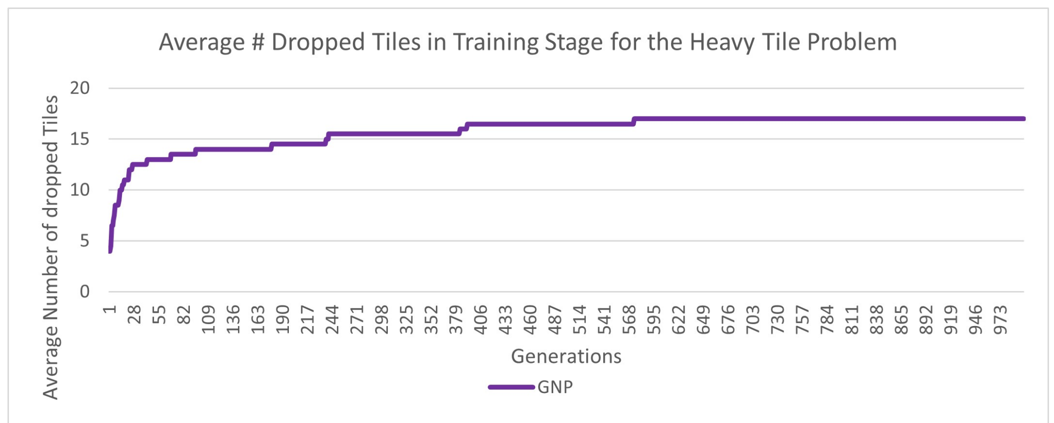


Figure 3.4: Average number of Tiles in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP.

Table 3.5: The GNP results on the Heavy Tile World Problem using one sub-program

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP	16018	17247	4029	19/30 S1 02/30 S2 08/30 S3	-	260700	868

Figure 3.3 and Figure 3.4 show the average fitness and number of dropped tiles for three experiments of the GNP algorithm with the Heavy Tile World Problem for 1000 generations. The algorithm was not able to reach the top results. The best

results for the algorithm in terms of the number of correctly dropped tiles were (19/30), (2/30), and (8/30) for training set (1), testing set (2), and testing set (3) (Figure 1.2), respectively, which appeared on generation number 868 as shown in Table 3.5.

3.2.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed)

The Heavy Tile World problem with two sub-programs has the same mechanism as the techniques with one sub-program except for some changes: We used variable-sized Genetic Networking Programming (VSGNP) on the chromosome structure to divide the graph into two sub-programs, one to solve the normal tiles and the other to solve the heavy tile. The parameters are shown in Table 3.6.

Chromosome Evaluation (Fitness):

The same fitness function is used here, too. the fitness value is calculated by equation 2.4.The distance $D_{distance}$ is calculated the same way used in section 3.2.1.

Evolutionary Operators:

The same Crossover and Mutation process in Normal Tile are used with the Heavy Tile World Problem considering this point:

On the evolutionary operation, the algorithm should ensure that the connection between the sub-programs (the connection for the Judgment node 8) is stable after the change. So, the normal connection should direct to sub-program 0, and the heavy connection should direct to sub-program 1 as shown in Figure 3.5.

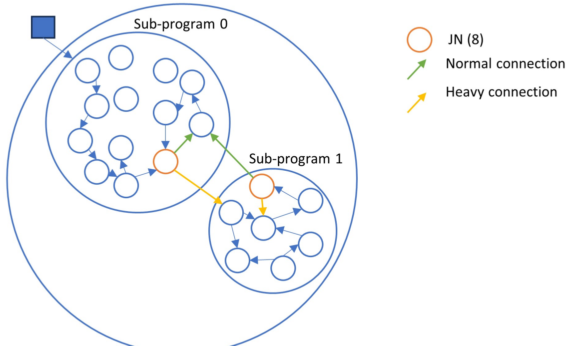


Figure 3.5: Heavy Tile Program Graph Structure.

Table 3.6: Parameters for the Tile World with Heavy Tile with two sub-programs with VSGNP.

Problem Domain	**Tile World Problem with Heavy Tile Using two sub-programs with VSGNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	

Connections	Judgement Node Connections	Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal → sub-program 0; Heavy → sub-program 1 For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.	MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.	

Chromosome Structure	
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.
Number of sub-nodes	Only one sub-node in this experiment.
Number of Sub-programs	Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.

Algorithm Parameters	
Number of individuals	300
Number of Elites	5
Number of Crossover individuals	120
Number of Mutation individuals	175
Mutation rate	Pm: 0.1, Pm1: 0.01
Crossover rate	Pc: 0.1
Tournament selection size	7

Results

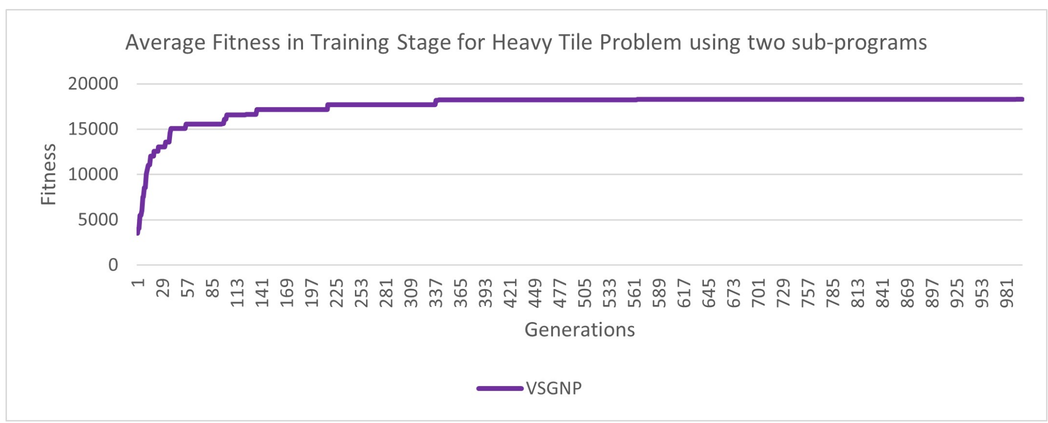


Figure 3.6: Average Fitness Training and Testing Stages for the Heavy Tile Problem using two sub-programs with VSGNP.

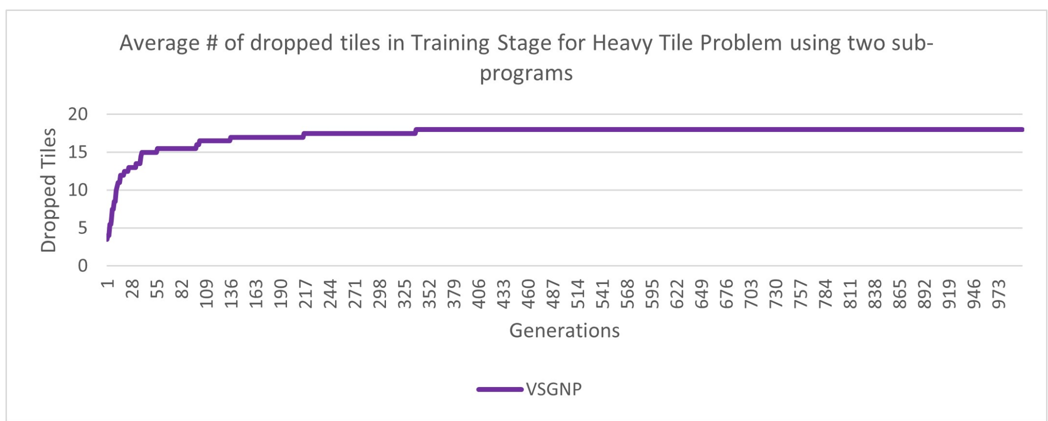


Figure 3.7: Average number of Tiles in Training and Testing Stages for the Heavy Tile Problem using two sub-programs with VSGNP.

Figure 3.6 and Figure 3.7 show the average fitness and the average number of dropped tiles for three experiments of the VSGNP algorithm with the Heavy Tile World Problem for 1000 generations. The algorithm was not able to reach the top results, but it has more performance than the GNP with one sub-program. The best results for the

Table 3.7: The VSGNP results on the Tile World Problem with Heavy Tile

Algorithm	**Mean of Fitness**	**Max of Fitness**	**Min of Fitness**	**Best training results and Best Chromosome**	**Average # evaluation until get the top training result**	**# evaluation until get the best chromosome**	**# generation for the best chromosome**
VSGNP	17606	18297	3520	19/30 S1 02/30 S2 09/30 S3	-	229800	765

algorithm were (19/30), (2/30), and (9/30) for the training set (1), testing set (2), and testing set (3) (Figure 1.2), respectively, which appeared on generation number 765, as shown in Table 3.7. Using multi-sub-programs with GNP shows a little increase in the results, but the differences are insignificant.

3.2.4 Prey and Predator Problem (Review)

For the Prey and Predator problem, we used the same parameters used in [11] and [12].

Table 3.8: Parameters for the Tile World with Heavy Tile with two sub-programs.

Problem Domain	**Prey and Predator with GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent
	J2	Judge what is in back position (JB)	- Prey
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Prey (PD)?	- Forward - Backward - Left - Right - None
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Prey; J5, → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	50		
Number of Elites	1		
Number of Crossover individuals	20		
Number of Mutation individuals	29		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	2		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the 30 environments one by one (as in Figure 1.6). The four agents start working from the first node in the graph and follow the directions (connections) until the Prey is rounded by all four agents or if the available number of steps is finished in which case the algorithm will close the current environment and start the next one. The parameters are shown in Table 3.8 Then, the fitness value is calculated by this formula [12]:

$$Fitness_{r,w} = \left[\sum_{i=1}^{S_{used}} \sum_{j=1}^{NoP} + \left(\frac{ES}{DP2P_i^j} \right) \right] + \left[\sum_{i=1}^{cPos} ES \right] + [4 \times ES \times (S_{UB} - S_{used} + 1)] \tag{3.1}$$

Where ES is the environment size $DP2P_i^j$ is the distance of predator j from the prey. S_{used} the number of used steps to catch prey. NoP the number of predators in the environment. S_{UB} is the maximum number of steps each agent is allowed to move. $cPos$ the number of adjacent cells occupied by a predator after S_{UB} of steps. Because the prey and predator are in a dynamic environment, each chromosome runs R times on a W environment, each with a different location of predators and prey. Finally, the final fitness is calculated as below [12]:

$$Fitness = \left(\sum_{r=1}^{R} \sum_{w=1}^{W} Fitness_{r,w} \right) \;\; / \;\; (R \times W) \tag{3.2}$$

Results

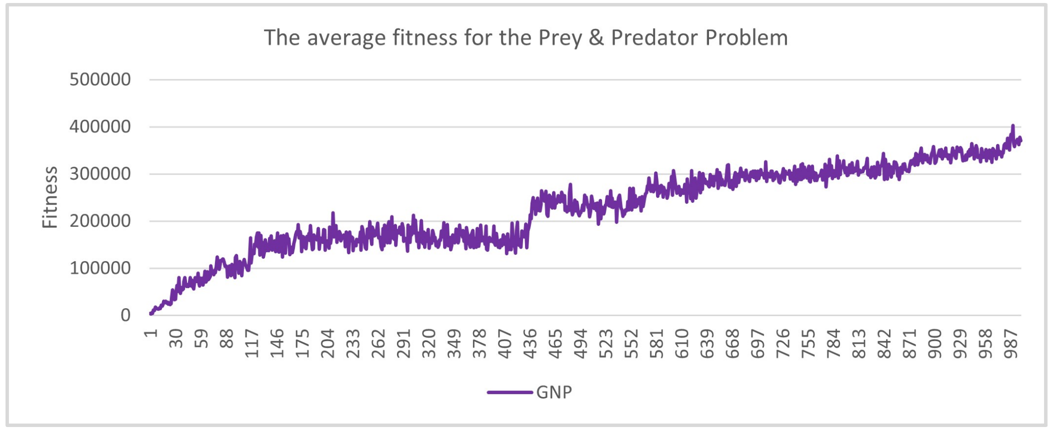


Figure 3.8: The average Fitness for the Prey and Predator Problem using GNP – Training and Testing Stage.

Table 3.9: The GNP results on the Prey and Predator Problem

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP	224856	402885	3662	831/900 27/30	30050	50000	999

Figure 3.8 and Figure 3.9 show the average Fitness and caught prey for three experiments applying GNP on Prey and Predator problem for 1000 generations. Table 3.9

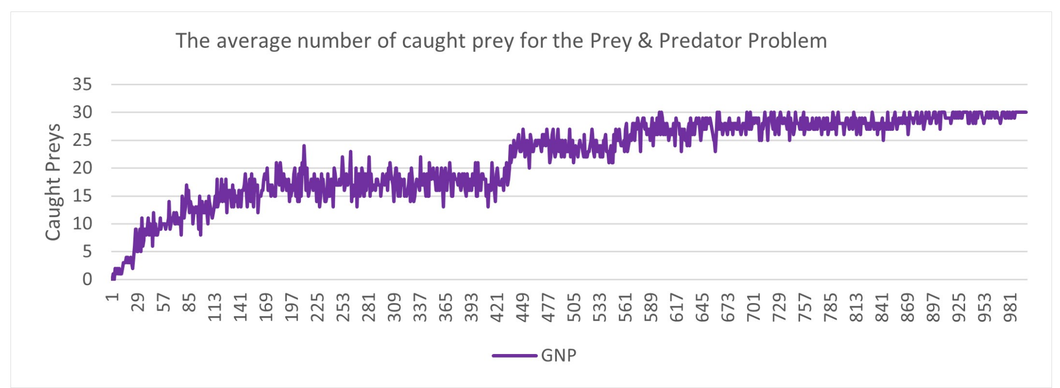


Figure 3.9: The average number of caught prey for the Prey and Predator Problem using GNP – Training and Testing Stage.

shows that the maximum average fitness for the GNP is 402885, and the best individual appears on generation number 999 that could succeed in catching 831/900 prey with an average of 27/30 caught prey.

3.3 GNP-CC-OS-TP (Genetic Networking Programming with Constraint Conformance – Optimal Search – Task Prioritization) and VSGNP-CC-OS-TP (Variable-sized Genetic Networking Programming with Constraint Conformance – Optimal Search – Task Prioritization)

This section will apply the three techniques from [17] and [14] to the GNP on the four testbeds:

1. Optimal Search: as a judgment node inside the chromosome graph.
2. Constraint conformance: when checking if the (processing nodes Move forward) it will accrue a conflict (when pushing the tile, it will be pushed to a trapped location that will not be able to be pushed a gain to anywhere). In this case, the algorithm will not apply the action; instead, it will choose the second connection in this node to change the agent's direction in the graph.
3. Task prioritization:
   a. In GNP, a scalable distance will be added to the fitness function to increase the fitness value each time the agent performs the task in prioritization.
   b. In GNP-RL, a scalable reward will be used to update the Q-value each time the agent performs the task in prioritization.

3.3.1 Tile World problem (Review)

For the Tile World problem, we used the parameters utilized in [17] and [14]. The parameters are shown in Table 3.10.

Table 3.10: Parameters for the Tile World with GNP-CC-OS-TP.

Problem Domain	**Tile World Problem with GNP-CC-OS-TP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole; J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the ten environments one by one (as in Figure 1.1 (1)). The three agents start working from the first node in the graph and follow the directions (connections) until all three tiles drop into the three holes or if the available number of steps is finished, the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 2.4 from [14], and the $D_{distance}$ will be calculated using the total of the below points:

- 4 points → when the agent pushes the nearest tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between

them before the push.

- 2 points → when the agent pushes the nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 10 points → when the agent pushes the tile to a hole.

Using this technique, we ensure the use of the task prioritization techniques from [17] and [14].

Results

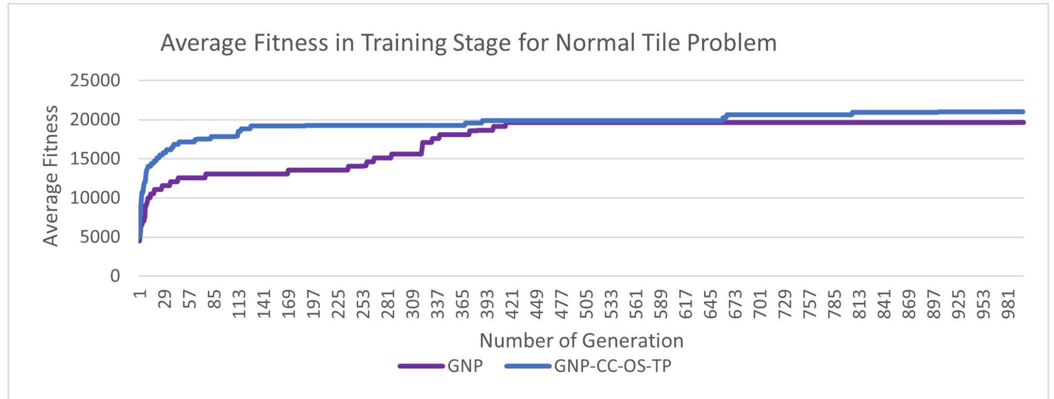


Figure 3.10: The average Fitness for the Tile World Problem using GNP-CC-OS-TP in Training and Testing Stage.

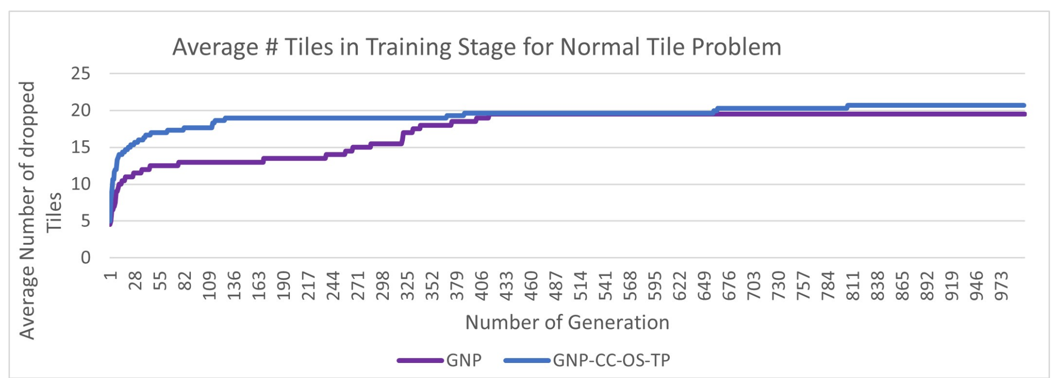


Figure 3.11: The average number of dropped tiles in the Tile World Problem using GNP-CC-OS-TP Training and Testing Stage.

Table 3.11: The GNP-CC-OS-TP results on the Tile World Problem

Algorithm	**Mean of Fitness**	**Max of Fitness**	**Min of Fitness**	**Best training results and Best Chromosome**	**Average # evaluation until get the top training result**	**# evaluation until get the best chromosome**	**# generation for the best chromosome**
GNP-CC-OS-TP	19642	21001	5010	24/30 S1 06/30 S2 15/30 S3	-	266700	888

Figure 3.10 and Figure 3.11 show the average of three experiments for the training and testing results when applying GNP-CC-OS-TP on the Tile World Problem. The GNP-CC-OS-TP algorithm achieved better results than the origin GNP after using the three techniques from [14] and [17]. But it could not reach the 30/30 results within the first 1000 generations. The maximum testing results were (24/30), (6/30), and (15/30) for the training set (1) (Figure 1.1 (1)), testing set (2) (Figure 1.1 (2)), and testing set (3) (Figure 1.1(3)), respectively. This result was achieved in the generation number 888, as shown in Table 3.11.

3.3.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed)

To apply GNP-CC-OP-TP on the Heavy Tile World problem, the same parameters for the Normal Tile World problem are used here but with some modifications (Table 3.12). The Judgment node number 8 has been changed from (giving the direction to the second nearest Tile) to (checking the type of the nearest Tile is normal or heavy).

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the ten environments one by one (as in Figure 1.2). The three agents start working from the first node in the graph and follow the directions (connections) until all three Tiles drop into the three holes or if the

Table 3.12: Parameters for the Tile World Problem with Heavy Tile Using one sub-program with GNP-CC-OS-TP.

Problem Domain	**Tile World Problem with Heavy Tile Using one sub-program with GNP-CC-OS-TP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile**
	J2	Judge what is in back position (JB)	- Hole
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left
	J6	Direction to the nearest Hole (HD)?	- Right
	J7	Direction to the second nearest Tile (THD)?	- None
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile; J5, J6, J7 → Forward, Backward, Left, Right, Nothing; J8 → Normal, Heavy		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
	Chromosome Structure		
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
	Algorithm Parameters		
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

available number of steps is finished, the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 2.4 from [14], and the $D_{distance}$ will be calculated using the total of the below points:

- 8 points → when the agent pushes the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

- 4 points → when the agent pushes the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 4 points → when the agent pushes the nearest Normal tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

- 4 points → when the agent pushes not the nearest heavy tile and the distance

between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

- 2 points → when the agent pushes not the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes the nearest Normal tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 10 points → when the agent pushes a normal tile to a hole.
- 20 points → when the agent pushes a heavy tile to a hole.

Using this technique, we ensure the use of the task prioritization techniques from [14].

Results

Table 3.13: GNP-CC-OS-TP results on the Tile World Problem with Heavy Tile using one sub-program

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-CC-OS-TP	16971	17749	6022	24/30 S1 03/30 S2 0830 S3	-	300000	999

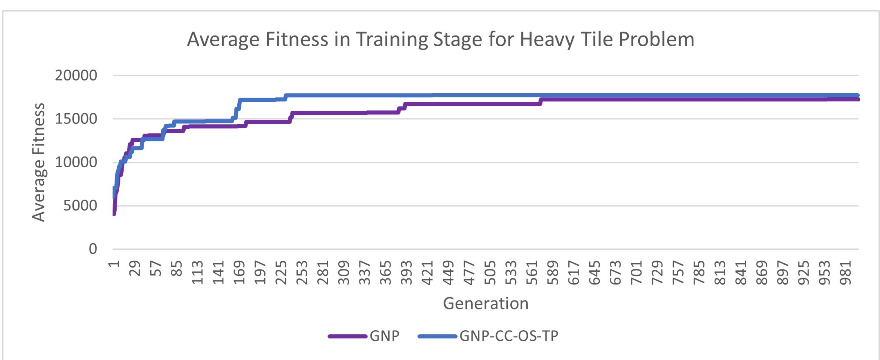


Figure 3.12: Average Fitness in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP-CC-OS-TP.

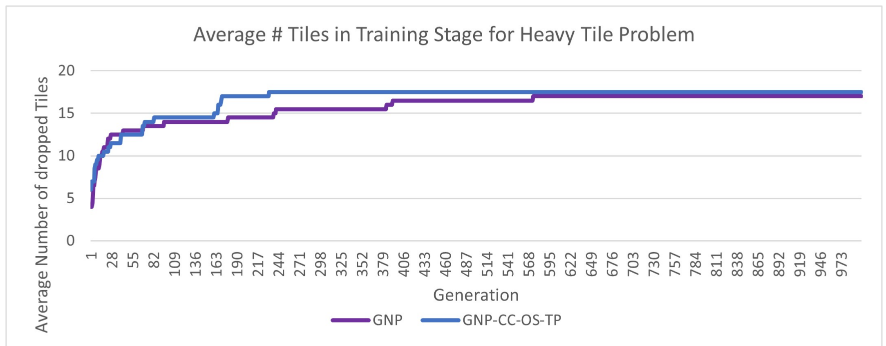


Figure 3.13: Average number of tiles in Training and Testing Stages for the Heavy Tile Problem with one sub-program using GNP-CC-OS-TP.

Figure 3.12 and Figure 3.13 show the average fitness and number of dropped tiles for three experiments with 1000 generations for each when applying the GNP-CC-OS-TP algorithm with the Heavy Tile World Problem using one-sub-program. The algorithm was not able to reach the top results. The best results for the algorithm were (24/30), (3/30), and (8/30) for the training set 91) (Figure 1.2 (1)), testing set (2) (Figure 1.2 (2)), and testing set (3) (Figure 1.2 (3)), respectively, shown on generation number 999, as shown in Table 3.13.

3.3.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed)

For the heavy tile, using two sub-programs, we used variable-sized Genetic Networking Programming (VSGNP) on the chromosome structure to divide the graph into two sub-programs, one to solve the normal tiles and the other to solve the heavy tile.

Table 3.14: Parameters for the Tile World with Heavy Tile with two sub-programs with VSGNP-CC-OS-TP.

Problem Domain	**Tile World Problem with Heavy Tile Using two sub-program with VSGNP-CC-OS-TP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal → sub-program 0; Heavy → sub-program 1 For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.		MF → With Conflict, Without Conflict TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

The same process that is used to evaluate the chromosome with one sub-program is used with two sub-programs.

Results

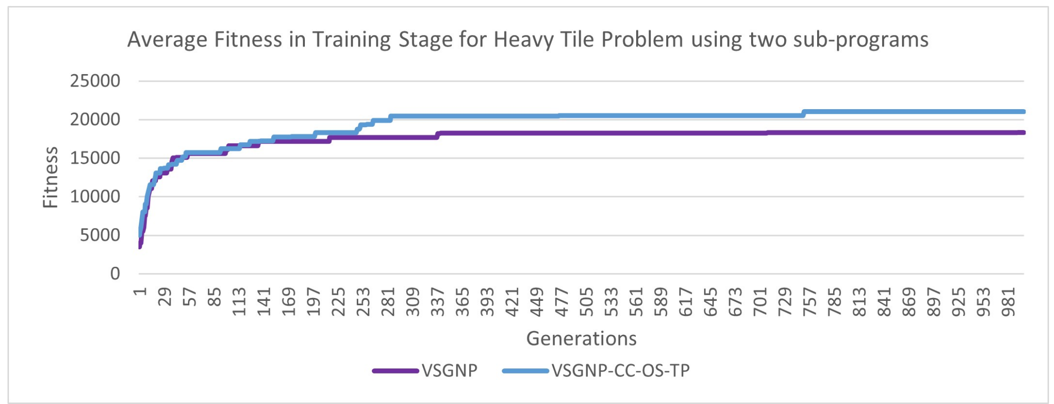


Figure 3.14: Average Fitness in Training and Testing Stages for the Heavy Tile Problem with two sub-programs using VSGNP-CC-OS-TP.

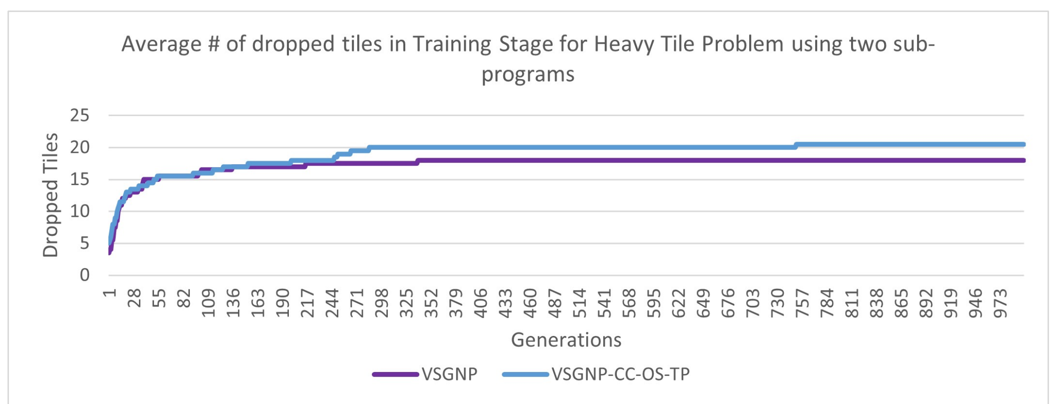


Figure 3.15: Average number of dropped Tiles in Training and Testing Stages for the Heavy Tile Problem with two sub-programs using VSGNP-CC-OS-TP.

Table 3.15: The VSGNP-CC-OS-TP results on the Tile World with Heavy Tile with two sub-programs

Algorithm	**Mean of Fitness**	**Max of Fitness**	**Min of Fitness**	**Best training results and Best Chromosome**	**Average # evaluation until get the top training result**	**# evaluation until get the best chromosome**	**# generation for the best chromosome**
VSGNP-CC-OS-TP	19533	21022	5016	21/30 S1 06/30 S2 13/30 S3	-	163800	545

Figure 3.14 and Figure 3.15 show the average fitness and number of dropped tiles for three experiments within 1000 generations for each when applying the VSGNP-CC-OS-TP algorithm with the Heavy Tile World Problem using two sub-programs. Although the algorithm could not reach the top results, the results were better than the ones from VSGNP and GNP-CC-OS-TP with one sub-program. The best results for the algorithm were (21/30), (6/30), and (13/30) for the training set (1) (Figure 1.2 (1)), testing set (2) (Figure 1.2 (2)), and testing set (3) (Figure 1.2 (3)), respectively, shown on generation number 545, as shown in Table 3.15.

3.3.4 Prey and Predator Problem (Proposed)

For the Prey and Predator problem, we used the same parameters used by [11] and [12], seen in table 3.16.

Table 3.16: Parameters for the Prey and Predator with GNP-CC-OS-TP.

Problem Domain	Prey and Predator with GNP-CC-OS-TP		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent
	J2	Judge what is in back position (JB)	- Prey
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Prey (PD)?	- Forward - Backward - Left - Right - None
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Prey J5, → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node As there is no trapped location for the prey in this problem, we didn't use the constraint conformance technique in this experiment.
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	50		
Number of Elites	1		
Number of Crossover individuals	20		
Number of Mutation individuals	29		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	2		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the 30 environments one by one (as in Figure 1.6). The four agents start working from the first node in the graph and follow the directions (connections) until the prey is rounded by all four agents or if the available number of steps is finished, in which case the algorithm will close the current

environment and start the next one. Then, the fitness value is calculated by equation 3.1.

Because the prey and predator are in a dynamic environment, each chromosome is run R times on a W environment, each with a different location of predators and prey. Finally, the final fitness is calculated as equation 3.2.

The task periotization techniques have not been applied here as there is just one task, which is to catch the one prey in the environment.

Results

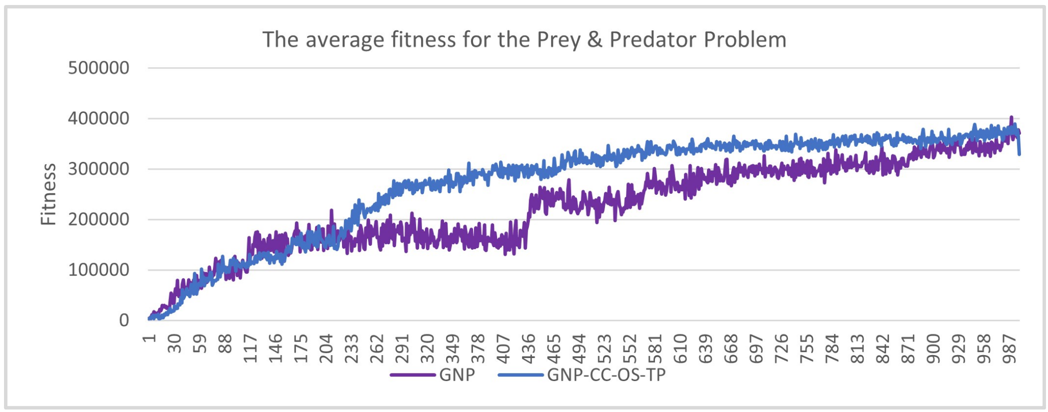


Figure 3.16: The average fitness for the Prey and Predator Problem – Training and testing Stages using GNP-CC-OS-TP.

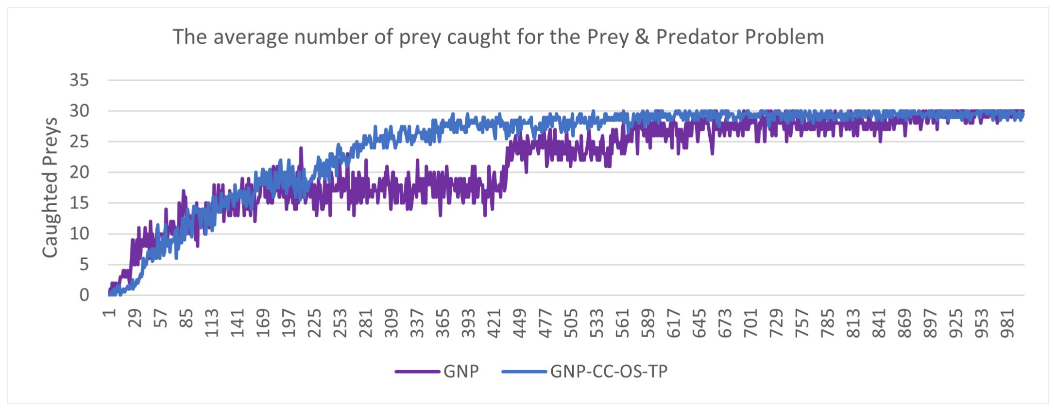


Figure 3.17: The average number of prey caught for the Prey and Predator Problem - Training and Testing Stages using GNP-CC-OS-TP.

Figure 3.16 and Figure 3.17 show the average Fitness and caught prey for three experiments when applying GNCC-OS-TP on the Prey and Predator Problem for 1000 generations. Table 3.17 shows that the maximum average fitness for the GNP is 389224,

Table 3.17: GNP-CC-OS-TP results on the Prey and Predator Problem

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-CC-OS-TP	273117	389224	3426	826/900 27/30	18850	50000	999

and the best individual appears on generation number 999 with an average of 27/30 caught prey.

3.4 GNP-RL (Genetic Networking Programming with Reinforcement Learning) and VSGNP-RL (Variable-sized Genetic Networking Programming with Reinforcement Learning)

3.4.1 Tile World problem (Review)

For the Tile World problem, we used the parameters utilized by Mabu et al. in [6].

Chromosome Evaluation (Fitness):

Agents begin at the starting node and proceed to the next connected nodes. When the agent visits the node, they must select one of the available sub-nodes. In order to choose a sub-node, the Sarsa algorithm with an ϵ-greedy technique is utilized, which was first implemented by Sutton and Barto [48]. The probability of randomly selecting the sub-node is set at 0.1, called exploration, and 0.9 for the sub-node with the maximum Q-value, called exploitation. Once the node is visited, the Q-value will be updated according to [6], where Q_ip is the Q value for the visited sub-node, Q_jq is the Q-value for the chosen subnode in the next node, and the chosen (α) learning and (γ) discount rates were both 0.9 in this work. Each time a tile is pushed into a hole, the reward equals 1. Equations 2.2 and 2.4 are used in this section.

Once the ten environments are completed, the fitness function is then calculated using Equation 2.4. The distance $D_{distance}$ is calculated the same way as in section 3.2.1.

The difference between the training and testing stages is the use of the ϵ-greedy technique in the training phase, which balances the exploration and the exploitation. In contrast, in the testing phase, the agent chooses only the sub-node with the maximum Q-value established in the training stage. The Q-values will no longer be updated during testing, which is the second difference.

Table 3.18: Parameters for the Tile World Problem with GNP-RL.

Problem Domain	**Tile World Problem with GNP-RL**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent - Tile - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Results

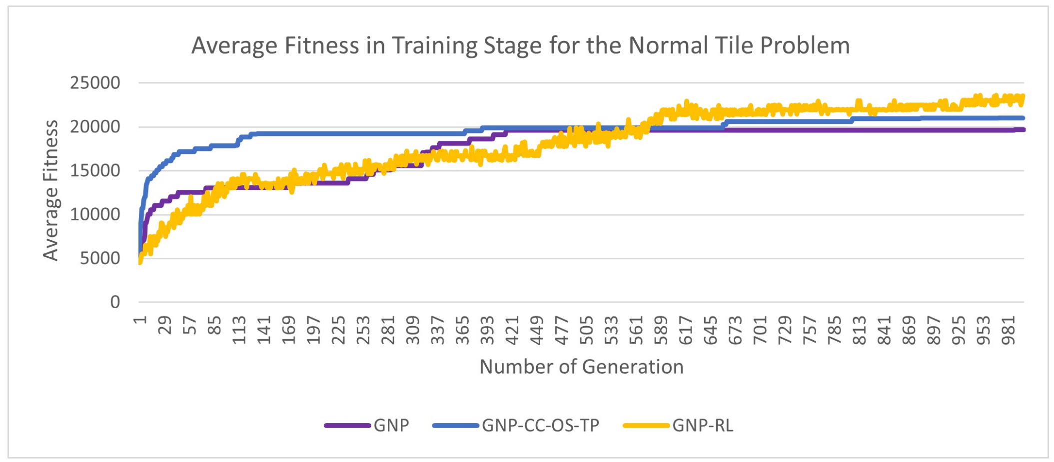


Figure 3.18: Average Fitness in Training Stage for the Normal Tile Problem using GNP-RL.

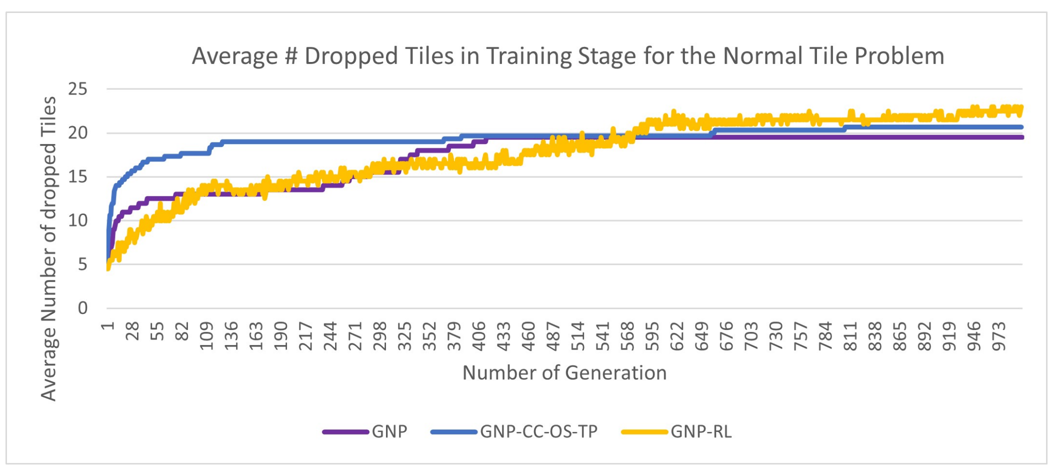


Figure 3.19: Average number of dropped Tiles in the Training Stage for the Normal Tile Problem using GNP-RL.

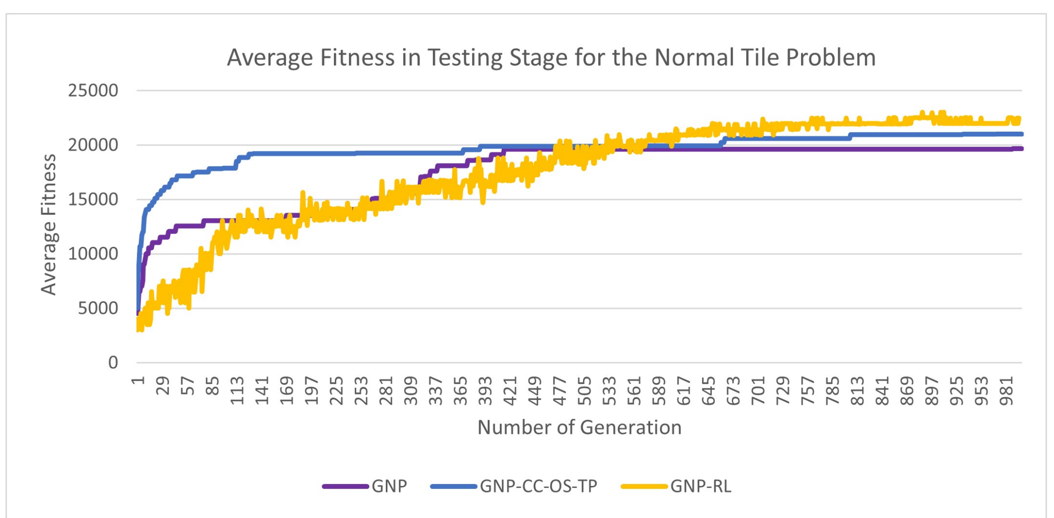


Figure 3.20: Average Fitness in Testing Stage for the Normal Tile Problem using GNP-RL.

Table 3.19: GNP-RL results on Tile World Problem.

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-RL	18098	23587	4520	17/30 S1 02/30 S2 13/30 S3	-	273300	910

Figure 3.18, Figure 3.19, Figure 3.20, and Figure 3.21 illustrate the average fitness and number of dropper Tiles for three experiments in the training and testing results when applying the GNP-RL on the Tile World Problem. GNP-RL could achieve better

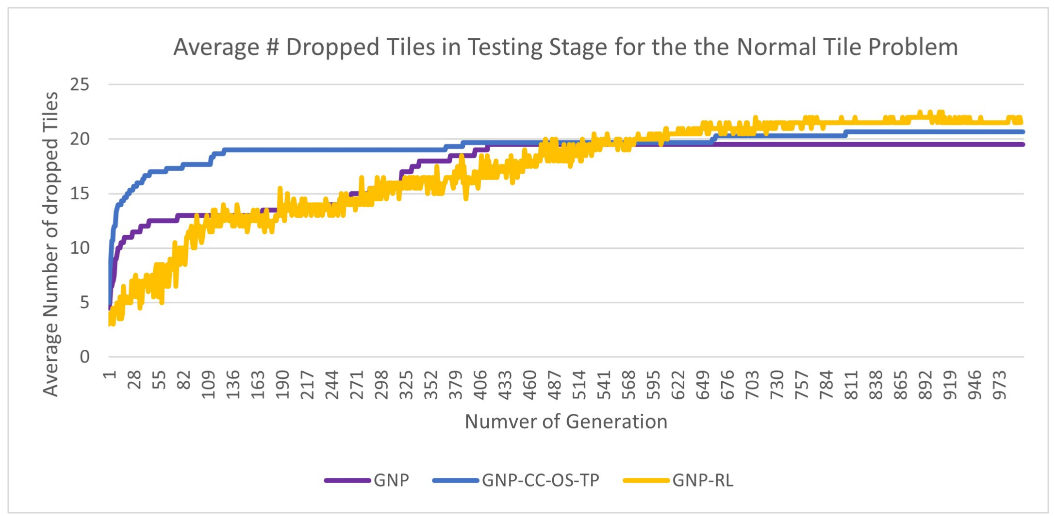


Figure 3.21: Average number of dropped Tiles in the Testing Stage for the Normal Tile Problem using GNP-RL.

results than GNP and GNP-CC-OS-TP. The maximum testing results for the GNP-RL in the Normal Tile World problem within the first 1000 generations were (17/30), (2/30), and (13/30) for the training set (1) (Figure 1.1 (1)), testing set (2) (Figure 1.1 (2)), and testing set (3) (Figure 1.1 (3)), respectively. This result was achieved in generation number 910, as shown in Table 3.19.

3.4.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed)

Applying GNP-RL on the Heavy Tile World problem has the same parameters as applying GNP on the same problem, which changes the judgment node (8) to check for the Tile type instead of getting the direction to the second nearest Tile.

Chromosome Evaluation (Fitness):

The same evaluation procedure that has been used with Normal Tile World with GNP-RL is used here, too, with the Heavy Tile World, as explained below: Agents begin at the starting node and proceed to the next connected nodes. When the agent visits the node, they must select one of the available sub-nodes. In order to choose a sub-node, the Sarsa algorithm with an ϵ-greedy technique is utilized, which was first implemented by Sutton and Barto [48]. The probability of randomly selecting the sub-node is set at 0.1, called exploration, and 0.9 for the sub-node with the maximum Q-value, called exploitation. Once the node is visited, the Q-value will be updated according to 2.2 Where Qip is the Q value for the visited sub-node, Qjq is the Q-value for the chosen sub-node in the next node, and the chosen (α) learning and (γ) discount rates were

Table 3.20: Parameters for the Tile World Problem with Heavy Tile Using one sub-program with GNP-RL.

Problem Domain	**Tile World Problem with Heavy Tile Using one sub-program with GNP-RL**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile; J5, J6, J7 → Forward, Backward, Left, Right, Nothing; J8 → Normal, Heavy		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

both 0.9 in this work. Each time a tile is pushed into a hole, the reward equals 1.

Once the ten environments are completed, the fitness function is then calculated using Equation 2.4, Where Dtile represents the number of tiles that were successfully dropped in the hole for each environment, Tremain is the remaining steps from each agent when the simulation is finished, and Ddistance refers to the total points that the agents accumulate during the simulation, and is calculated in the same way as Section 3.2.1.

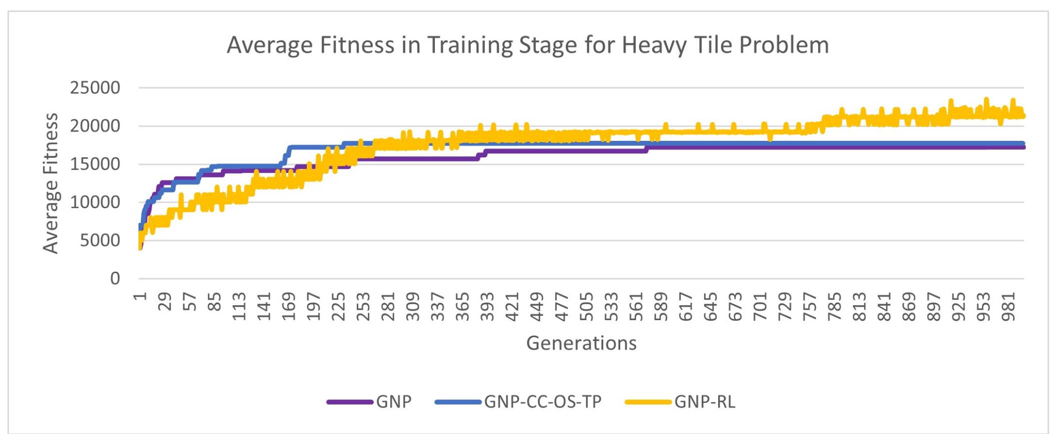


Figure 3.22: Average Fitness in Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL.

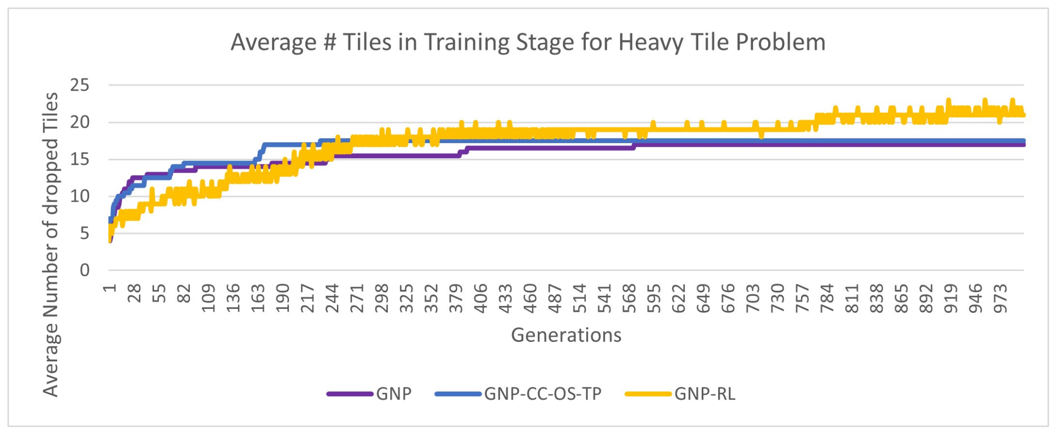


Figure 3.23: Average number of dropped tiles in the Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL.

Table 3.21: GNP-RL results on the Heavy Tile World Problem with one sub-program

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-RL	17519	23499	4037	21/30 S1 07/30 S2 15/30 S3	-	291300	970

Results

Figure 3.22, Figure 3.23, Figure 3.24, and Figure 3.25 show the average Fitness and number of dropped tiles on the training and testing results for three experiments when applying GNP-RL on the Heavy Tile World Problem using one sub-program for 1000

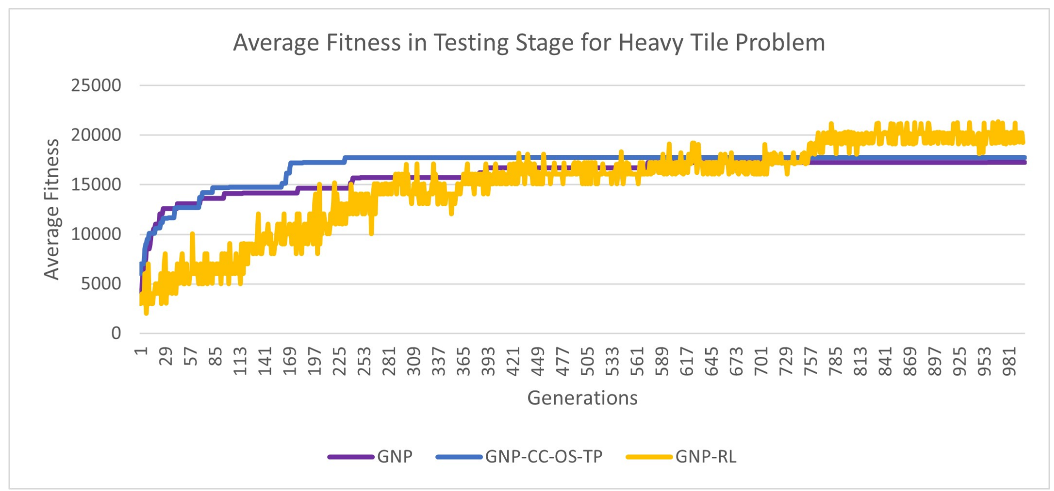


Figure 3.24: Average Fitness in Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL.

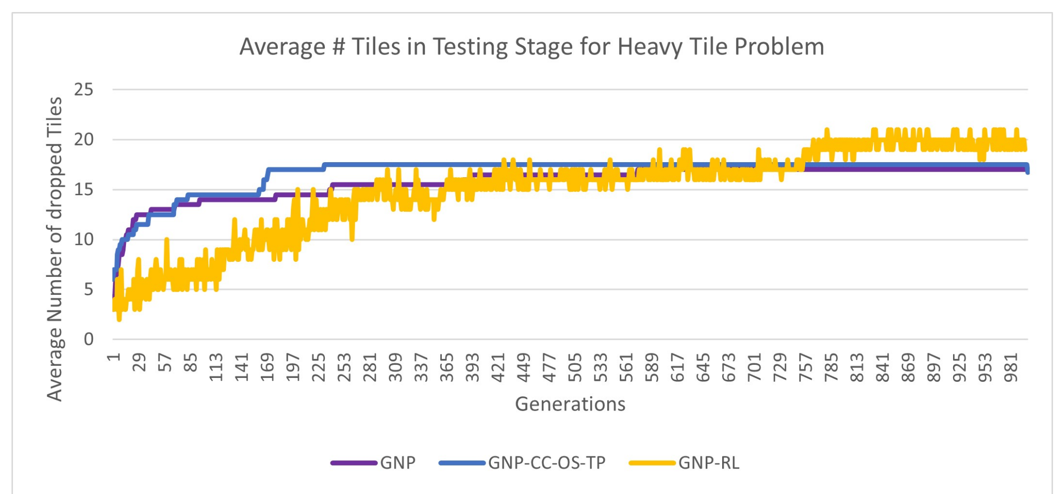


Figure 3.25: Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL.

generations. The difference in GNP-RL results between training and testing can be referred to the exploration and exploitation phases of training. GNP-RL received better results in the training and testing phases than the other algorithms (GNP, GNP-CC-OS-TP). Table 3.21 shows that the maximum fitness for the GNP-RL is 23499, and the best chromosome appears on generation number 970 with 21/30, 7/30, and 15/30 dropped tiles in training set in Figures 1.2 (1), (2) and (3).

3.4.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed)

Applying VSGNP-RL on the Heavy Tile World problem has the same parameters as applying VSGNP on the same problem, which divides the graph into two sub-programs and changes the judgment node (8) to check for the tile type instead of getting the direction to the second nearest tile.

Table 3.22: Parameters for the Tile World Problem with Heavy Tile Using two sub-programs with VSGNP-RL.

Problem Domain	**Tile World Problem with Heavy Tile Using two sub-program with VSGNP-RL**		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal → sub-program 0; Heavy → sub-program 1 For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		

Chromosome Structure	
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.
Number of Sub-programs	Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.

Algorithm Parameters	
Number of individuals	300
Number of Elites	5
Number of Crossover individuals	120
Number of Mutation individuals	175
Mutation rate	Pm: 0.1, Pm1: 0.01
Crossover rate	Pc: 0.1
Tournament selection size	7

Chromosome Evaluation (Fitness):

The same evaluation procedure that has been used with Normal Tile World with GNP-RL is used here, too, with the Heavy Tile World problem.

Results

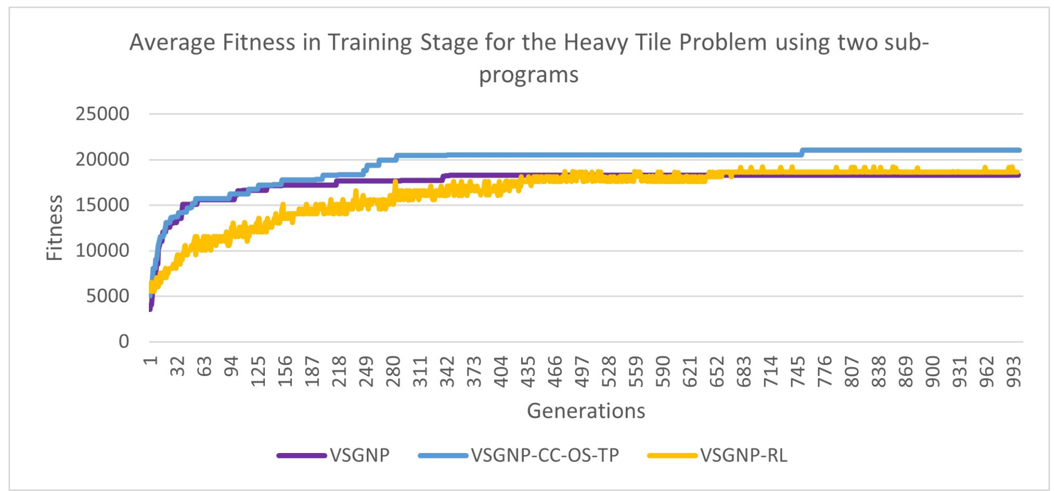


Figure 3.26: Average Fitness in Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL.

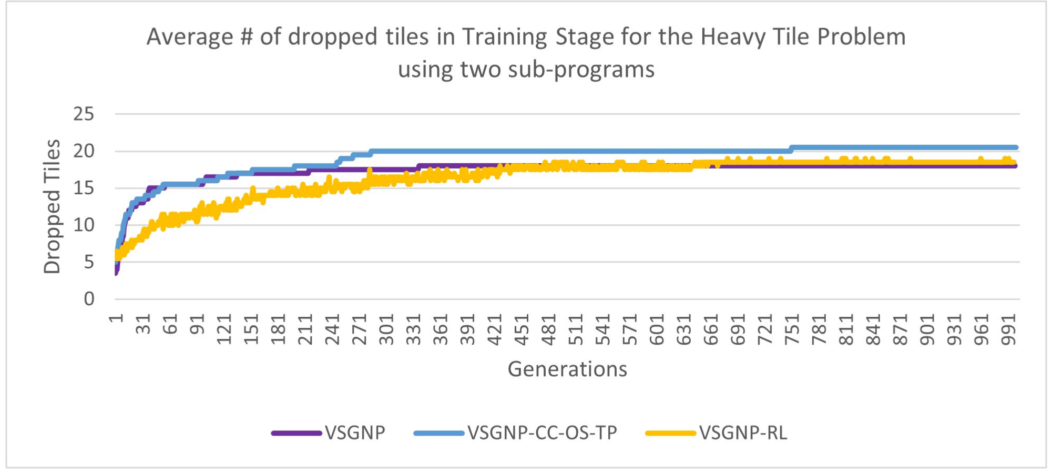


Figure 3.27: Average number of dropped Tiles in the Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL.

Figure 3.26, Figure 3.27, Figure 3.28, and Figure 3.29 show the average Fitness and number of dropped tiles on the training and testing results for three experiments when applying VSGNP-RL on the Heavy Tile World Problem with two sub-programs for 1000 generations. VSGNP-RL results differ between training and testing due to

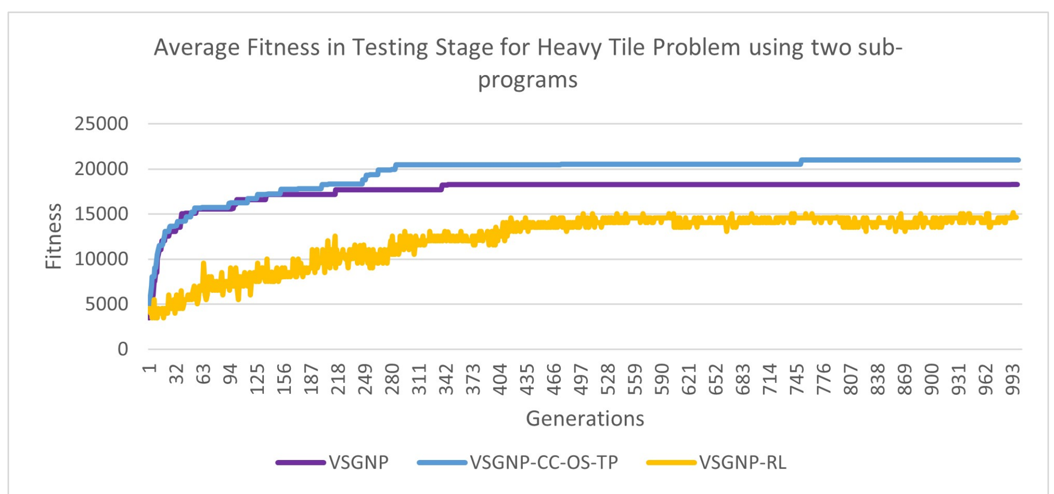


Figure 3.28: Average Fitness in Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL.

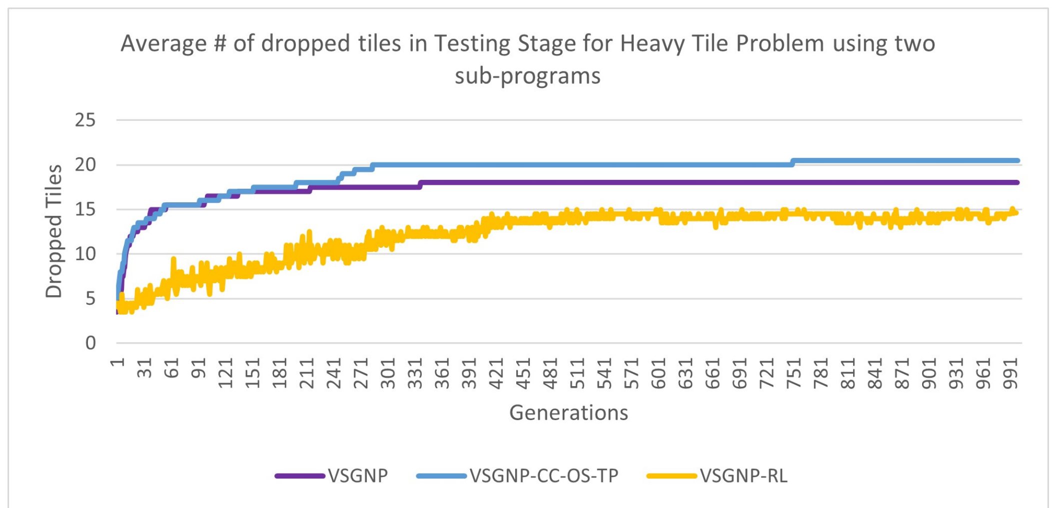


Figure 3.29: Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL.

Table 3.23: VSGNP-RL results on the Tile World Problem with Heavy Tile using two sub-programs

Algorithm	**Mean of Fitness**	**Max of Fitness**	**Min of Fitness**	**Best training results and Best Chromosome**	**Average # evaluation until get the top training result**	**# evaluation until get the best chromosome**	**# generation for the best chromosome**
VSGNP-RL	16480	19198	5525	16/30 S1 07/30 S2 13/30 S3	-	173700	578

the exploration and exploitation phases. Using two sub-programs with VSGNP-RL on the Heavy Tile World Problem could not get better results in the training and testing

phases than the other algorithms (VSGNP, VSGNP-CC-OS-TP). Table 3.23 shows that the maximum fitness for the GNP-RL is 19198, and the best chromosome appears on generation number 578 with 16/30, 7/30, and 13/30 dropped tiles in the training set (1), (2) and (3) respectively (Figures 1.2 (1), (2) and (3)).

3.4.4 Prey and Predator Problem (Review)

For the Prey and Predator problem, we used the same parameters used by [11] and [12].

Table 3.24: Prey and Predator with GNP-RL.

Problem Domain	Prey and Predator with GNP-RL		
Judgement Nodes	ID	Query Definition	Possible response to query
	J1	Judge what is in front position (JF)	- Agent
	J2	Judge what is in back position (JB)	- Prey
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Prey (PD)?	- Forward - Backward - Left - Right - None
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Prey; J5, → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	50		
Number of Elites	1		
Number of Crossover individuals	20		
Number of Mutation individuals	29		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	2		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the 30 environments one by one (as in Figure 1.6). Agents begin at the starting node and proceed to the next connected nodes. When the agent visits the node, they must select one of the available sub-nodes. In order to choose a sub-node, the Sarsa algorithm with an ϵ-greedy technique is utilized, which was first implemented by Sutton and Barto [48]. The probability of randomly selecting the sub-node is set at 0.1, called exploration, and 0.9 for the sub-node with

the maximum Q-value, called exploitation. Once the node is visited, the Q-value will be updated according to 2.2, Where Qip is the Q value for the visited sub-node, Qjq is the Q-value for the chosen sub-node in the next node, and the chosen (α) learning and (γ) discount rates were both 0.9 in this work. When the prey is surrounded by the predators, the reward is equal to 1.

The four agents start working from the first node in the graph and follow the directions (connections) until the prey is rounded by all four agents or if the available number of steps is finished, in which case the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equations 3.1 and 3.2.

Results

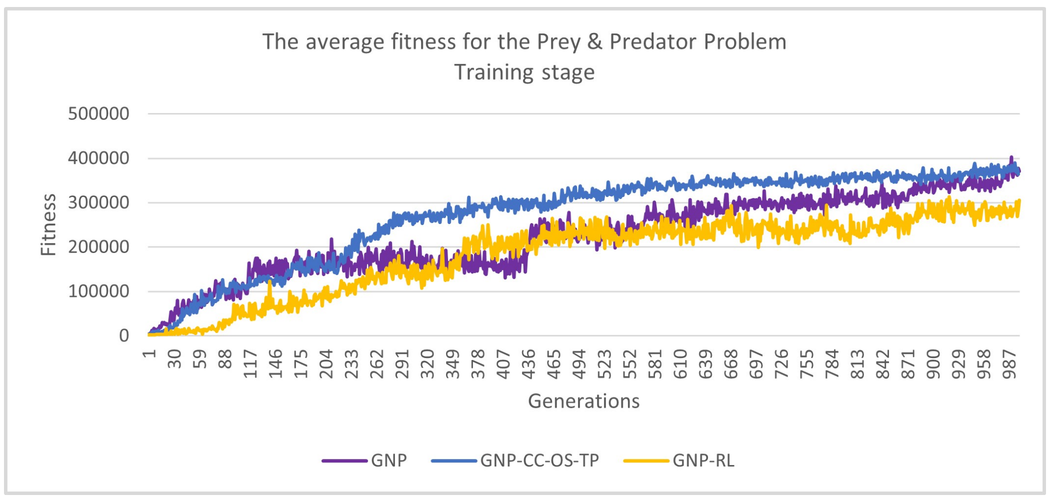


Figure 3.30: The average fitness for the Prey and Predator Problem – Training Stage using GNP-RL.

Table 3.25: GNP-RL results on the Prey and Predator Problem

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-RL	184172	313855	2700	782/900 26/30	46350	49200	983

Figure 3.30 and Figure 3.31 show the average Fitness and caught prey on the training results for three experiments when applying GNP-RL on Prey and Predator Problem for 1000 generations. Figure 3.33 and Figure 3.33 show the average fitness and caught prey on the testing results for three experiments for 1000 generations. The

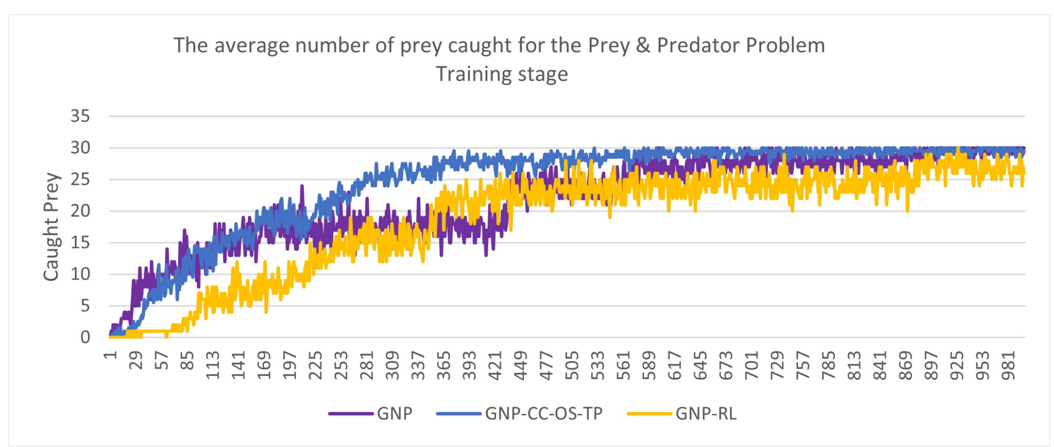


Figure 3.31: The average number of caught prey for the Prey and Predator Problem – Training Stage using GNP-RL.

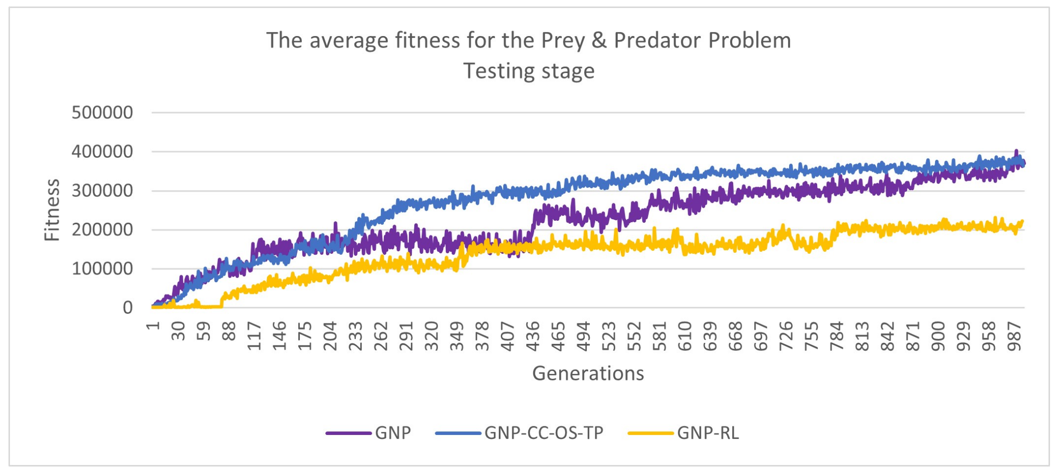


Figure 3.32: The average fitness for the Prey and Predator Problem - Testing stage using GNP-RL Figure.

training results are different from the testing results in GNP-RL, especially in the fitness value, because of the exploration and exploitation phases in the training stage. Also, because the environment has a prey that moves randomly, the results for the GNP-RL were less than the GNP and GNP-CC-OS-TP. Table 3.25 shows that the maximum fitness for the GNP-RL is 313855, and the best chromosome appears on generation number 983 with an average of 26/30 caught prey.

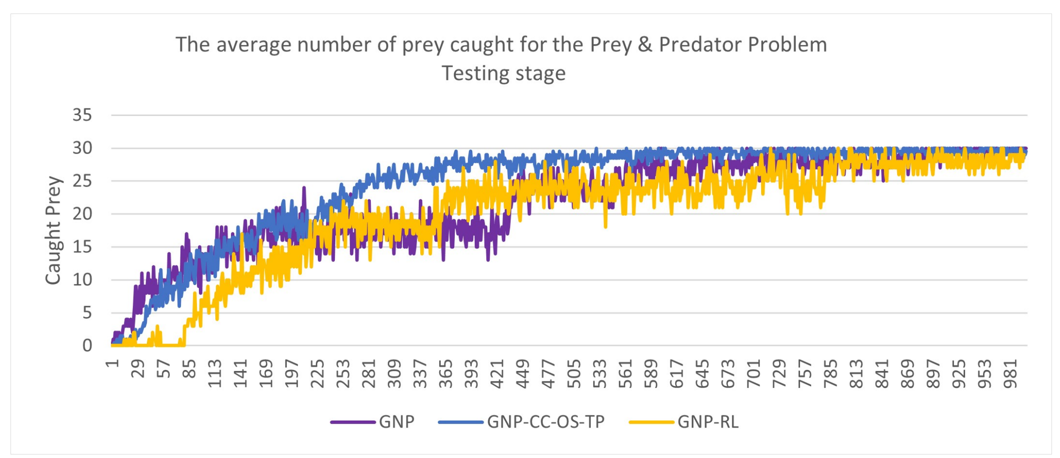


Figure 3.33: The average number of caught prey for the Prey and Predator Problem - Testing stage using GNP-RL.

3.5 GNP-RL-CC-OP-TP (Genetic Networking Programming with Reinforcement Learning - Constraint Conformance – Optimal Search – Task Prioritization) and VSGNP-RL-CC-OS-TP (Variable-sized Genetic Networking Programming with Reinforcement Learning - Constraint Conformance – Optimal Search – Task Prioritization)

This section will apply the three techniques from [17] [14] to the GNP-RL on the four testbeds.

3.5.1 Tile World problem (Review)

For the Tile World problem, we used the parameters utilized by Mabu et al. in [6].

Chromosome Evaluation (Fitness):

Agents begin at the starting node and proceed to the next connected nodes. When the agent visits the node, they must select one of the available sub-nodes. In order to choose a sub-node, the Sarsa algorithm with an ϵ-greedy technique is utilized, which was first implemented by Sutton and Barto [48]. The probability of randomly selecting the sub-node is set at 0.1, called exploration, and 0.9 for the sub-node with the maximum Q-value, called exploitation. Once the node is visited, the Q-value will be updated according to 2.2, where Q_{ip} is the Q value for the visited sub-node, Q_{jq} is the Q-value for the chosen sub-node in the next node, and the chosen (α) learning and (γ) discount

Table 3.26: Parameters for the Tile World Problem with GNP-RL-CC-OS-TP

Problem Domain	**Tile World Problem with GNP-RL-CC-OS-TP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole; J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

rates were both 0.9 in this work. In the GNP-RL algorithm, each time a tile is pushed into a hole, the reward is equal to 1, but in GNP-RL-CC-OS-TP, to apply the task prioritization technique, the reward has a scalable value as follows [17] [14]:

- 4 points → when the agent pushes the nearest tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes the nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.

- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 10 points → when the agent pushes a normal tile to a hole.

Once the ten environments are completed, the fitness function is then calculated using Equation 2.4.

Results

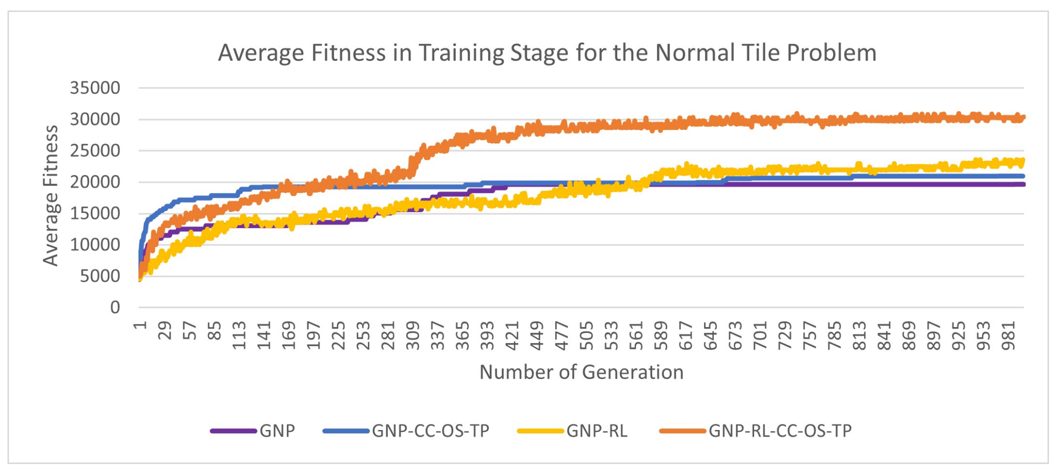


Figure 3.34: Average Fitness in Training Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP.

Table 3.27: GNP-RL-CC-OS-TP results on Tile World Problem.

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-RL-CC-OS-TP	25378	30946	5021	30/30 S1 22/30 S2 25/30 S3	172500	183900	612

Figure 3.34 and Figure 3.35 show the average Fitness and number of dropped tiles on the training results for three experiments when applying GNP-RL-CC-OS-TP on the Normal Tile World Problem for 1000 generations. Figure 3.36 and Figure 3.37 show the average fitness and number of dropped tiles on the testing results for three experiments for 1000 generations. The training results differ from the testing results in GNP-RL,

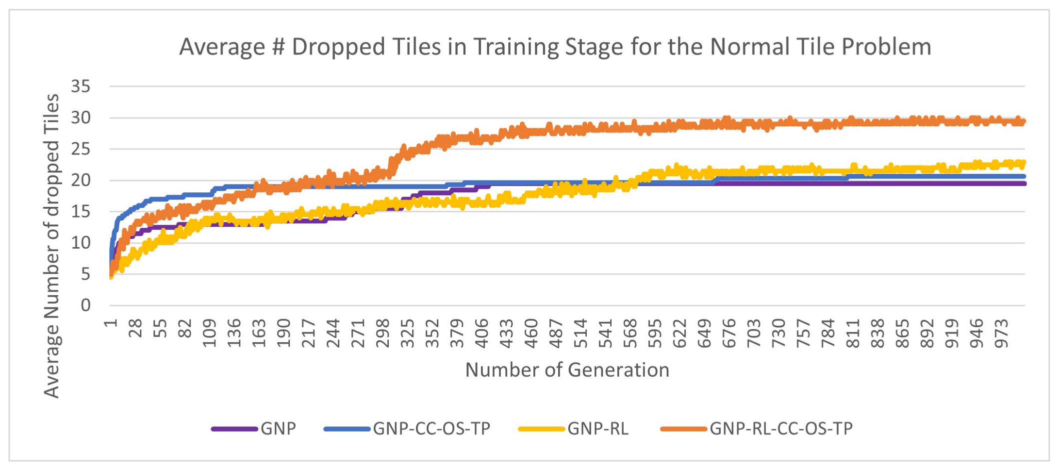


Figure 3.35: Average number of dropped tiles in Training Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP.

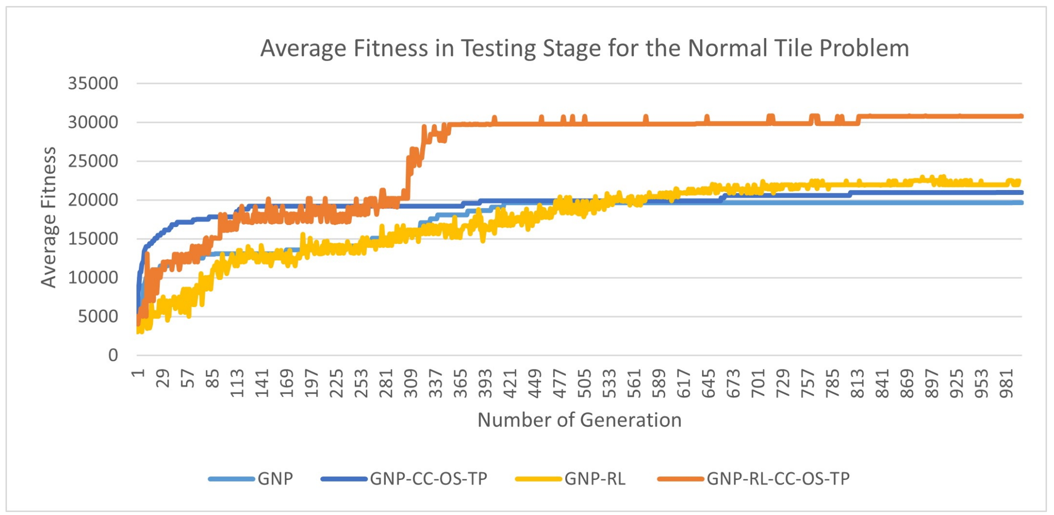


Figure 3.36: Average Fitness in Testing Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP.

especially because of the exploration and exploitation phases in the training stage. GNP-RL-CC-OS-TP had the best results on the training and testing than the other algorithms (GNP, GNP-CC-OS-TP, GNP-RL). Table 3.27 shows that the maximum fitness for the GNP-RL is 30946, and the best chromosome appears on generation number 612 with 30/30, 22/30, and 25/30 dropped tiles in the training set (1) (Figure 1.1 (1)), testing sets (2) (3), (Figure 1.1 (2)(3)) respectively.

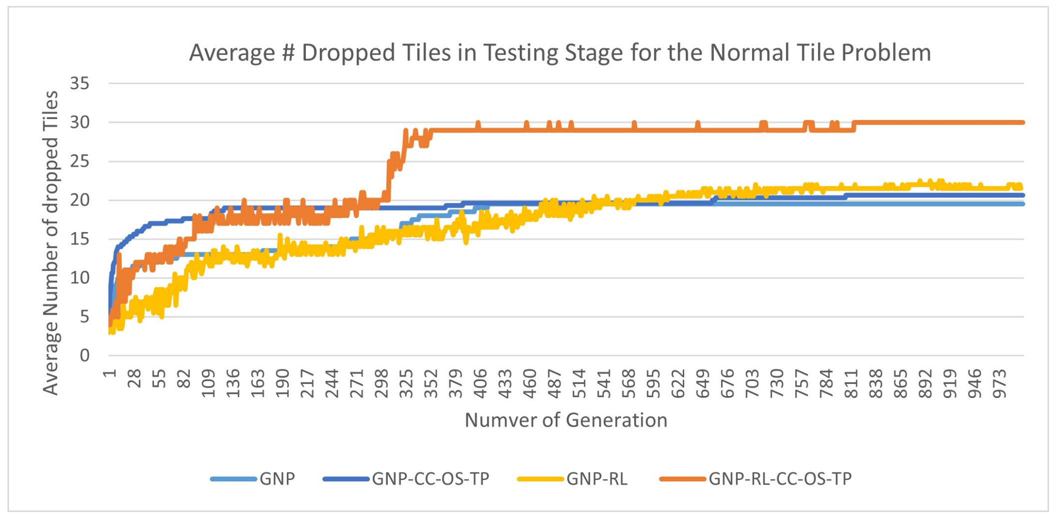


Figure 3.37: Average number of dropped tiles in Testing Stage for the Normal Tile Problem using GNP-RL-CC-OS-TP.

3.5.2 Tile World Problem with Heavy Tile Using one sub-program for the graph (Proposed)

Applying GNP-RL-CC-OS-TP on the Heavy Tile World problem has the same parameters as applying GNP-CC-OS-TP on the same problem, which changes the judgment node (8) to check for the tile type instead of getting the direction to the second nearest tile.

Chromosome Evaluation (Fitness):

The same evaluation procedure that has been used with the Normal Tile World with GNP-RL is used here, too, with the Heavy Tile World as explained below:

Agents begin at the starting node and proceed to the next connected nodes. When the agent visits the node, they must select one of the available sub-nodes. In order to choose a sub-node, the Sarsa algorithm with an ϵ-greedy technique is utilized, which was first implemented by Sutton and Barto [48]. The probability of randomly selecting the sub-node is set at 0.1, called exploration, and 0.9 for the sub-node with the maximum Q-value, called exploitation. Once the node is visited, the Q-value will be updated according to 2.2, where Q_{ip} is the Q value for the visited sub-node, Q_{jq} is the Q-value for the chosen sub-node in the next node, and the chosen (α) learning and (γ) discount rates were both 0.9 in this work.

In the GNP-RL algorithm, each time a tile is pushed into a hole, the reward is equal to 1, but in GNP-RL-CC-OS-TP, to apply the task prioritization technique, the reward has a scalable value as follows:

Table 3.28: Parameters for the Tile World Problem with Heavy Tile Using one sub-program with GNP-RL-CC-OS-TP.

Problem Domain	**Tile World Problem with Heavy Tile Using one sub-program with GNP-RL-CC-OS-TP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile; J5, J6, J7 → Forward, Backward, Left, Right, Nothing; J8 → Normal, Heavy		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

- 8 points → when the agent pushes the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 4 points → when the agent pushes the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 4 points → when the agent pushes the nearest Normal tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 4 points → when the agent pushes not the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes not the nearest heavy tile and the distance

between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 2 points → when the agent pushes the nearest Normal tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 10 points → when the agent pushes a normal tile to a hole.
- 20 points → when the agent pushes a heavy tile to a hole.

Once the ten environments are completed, the fitness function is then calculated using Equation 2.4.

Results

Table 3.29: GNP-RL-CC-OS-TP results on the Heavy Tile World Problem with one sub-program.

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-RL-CC-OS-TP	28965	32423	6031	30/30 S1 17/30 S2 30/30 S3	83700	131100	436

Figure 3.38, Figure 3.39, Figure 3.40, and Figure 3.41 show the average Fitness and number of dropped tiles on the training and testing results for three experiments when applying GNP-RL-CC-OS-TP on the Heavy Tile World Problem with one sub-program

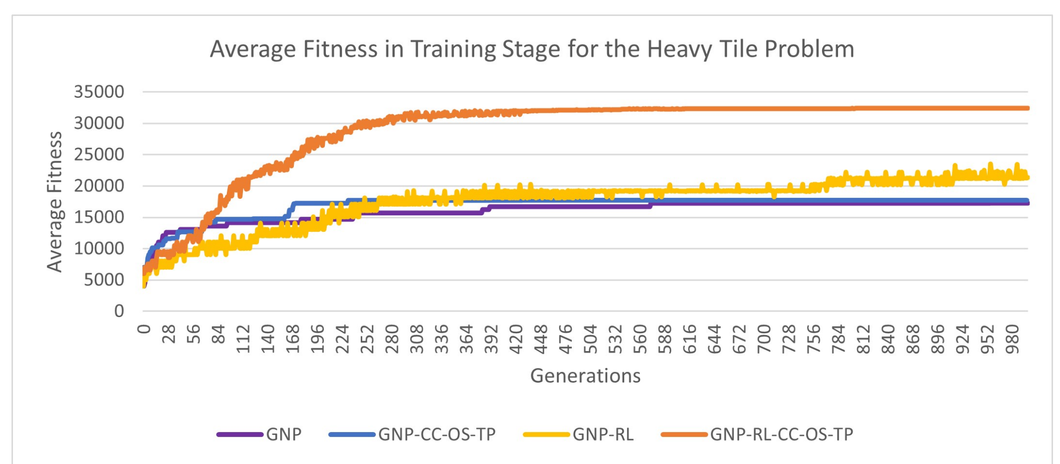


Figure 3.38: Average Fitness in Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP.

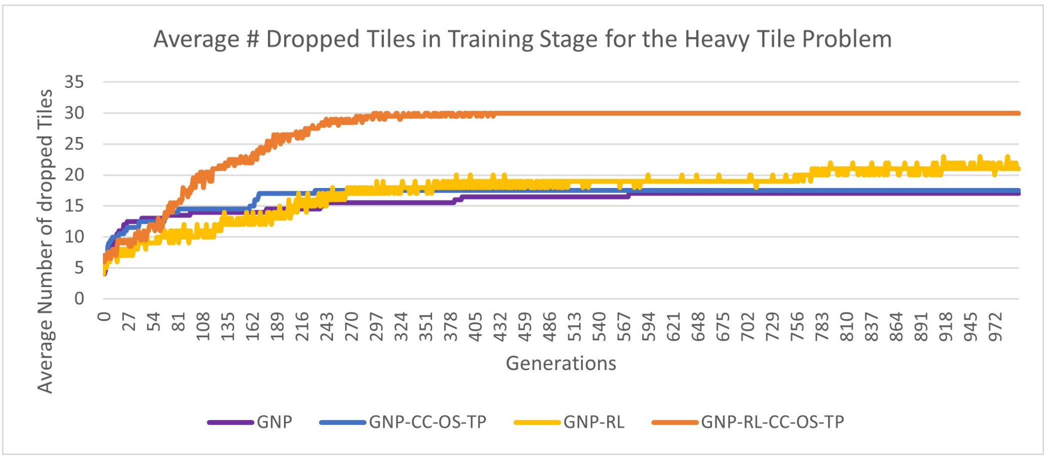


Figure 3.39: Average number of dropped Tiles in Training Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP.

for 1000 generations. The difference between training and testing results in GNP-RL-CC-OS-TP can be attributed to the exploration and exploitation phases during training. GNP-RL-CC-OS-TP performed better on both the training and testing phases than the other algorithms (GNP, GNP-CC-OS-TP, GNP-RL) and shows results superior to those when applied to the Tile World Problem. Table 3.29 shows that the maximum fitness for the GNP-RL-CC-OS-TP is 32423, and the best chromosome appears on generation number 436 with 30/30, 17/30, and 30/30 dropped tiles, in training set (1) and testing sets (2) and (3) from Figure 1.2 respectively.

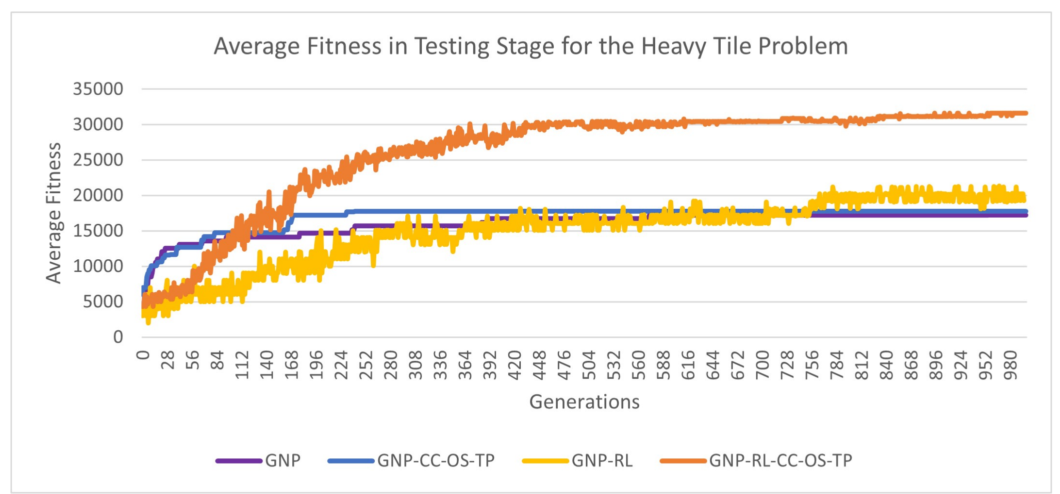


Figure 3.40: Average Fitness in Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP.

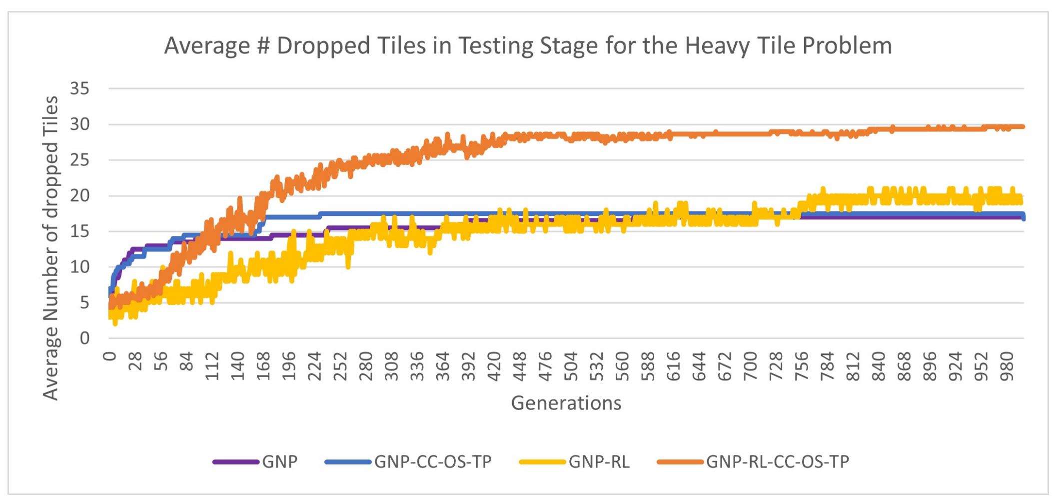


Figure 3.41: Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with one sub-program using GNP-RL-CC-OS-TP.

3.5.3 Tile World Problem with Heavy Tile Using two sub-programs for the graph (Proposed)

Applying GNP-RL-CC-OS-TP on the Heavy Tile World problem has the same parameters as applying GNP-CC-OS-TP on the same problem, which divides the graph into two sub-programs and changes the judgment node (8) to check for the tile type instead of getting the direction to the second nearest tile.

Table 3.30: Parameters for the Tile World Problem with Heavy Tile Using two sub-programs with VSGNP-RL-CC-OS-TP.

Problem Domain	Tile World Problem with Heavy Tile Using two sub-program with VSGNP-RL-CC-OS-TP		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile**
	J2	Judge what is in back position (JB)	- Hole
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left
	J6	Direction to the nearest Hole (HD)?	- Right
	J7	Direction to the second nearest Tile (THD)?	- None
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal → sub-program 0; Heavy → sub-program 1 For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.		MF → With Conflict, Without Conflict TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) below will explain how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

The same evaluation procedure that has been used with Heavy Tile World with GNP-RL-CC-OS-TP with one sub-program is used here, too, with the Heavy Tile World with two sub-programs.

Results

Figure 3.42, Figure 3.43, Figure 3.44, and Figure 3.45 show the average Fitness and number of dropped tiles on the training and testing results for three experiments when

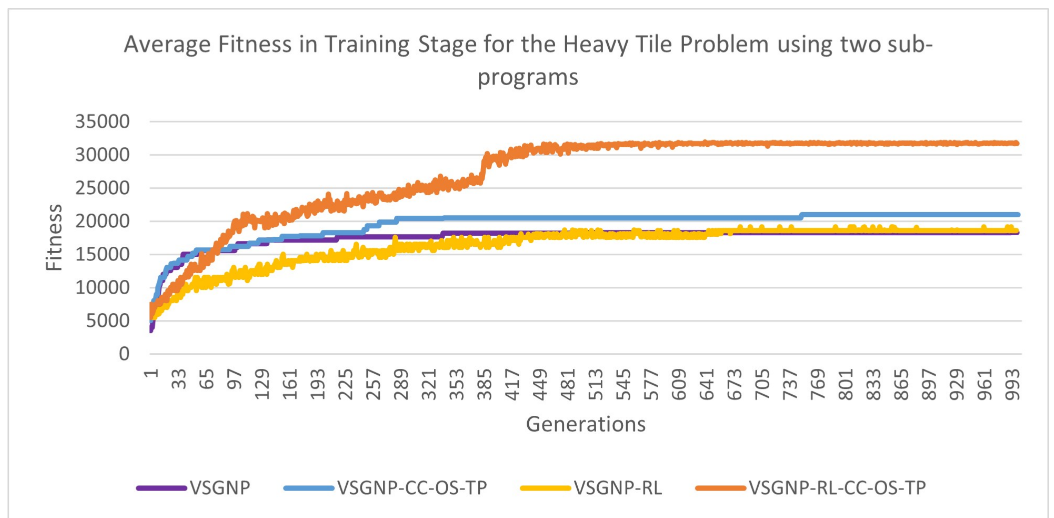


Figure 3.42: Average Fitness in Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP.

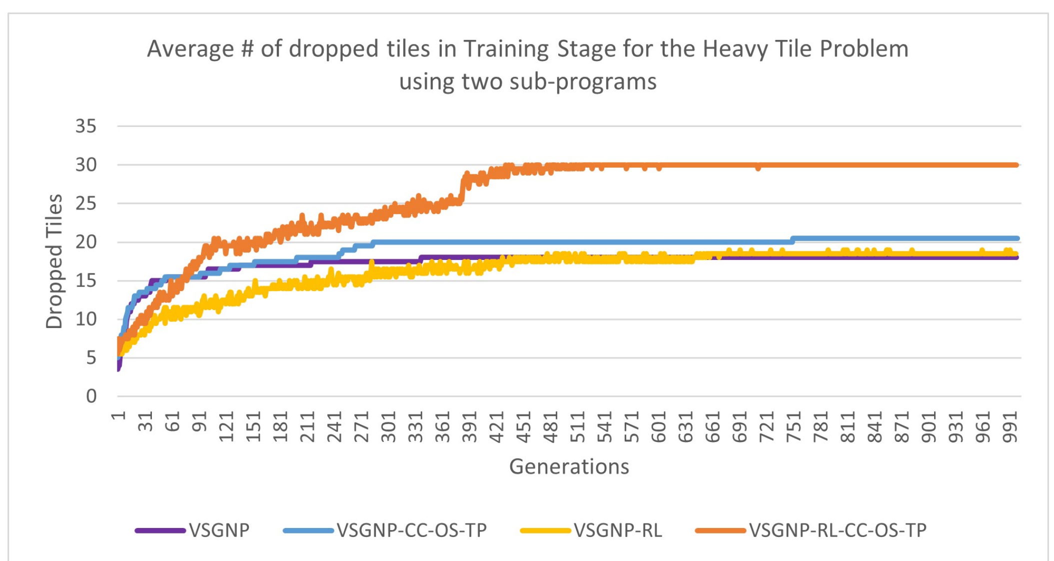


Figure 3.43: Average number of dropped Tiles in Training Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP.

Table 3.31: The VSGNP-RL-CC-OS-TP results on the Heavy Tile World Problem using two sub-programs

Algorithm	**Mean of Fitness**	**Max of Fitness**	**Min of Fitness**	**Best training results and Best Chromosome**	**Average # evaluation until get the top training result**	**# evaluation until get the best chromosome**	**# generation for the best chromosome**
VSGNP-RL-CC-OS-TP	27200	31951	5523	30/30 S1 17/30 S2 30/30 S3	125400	198600	661

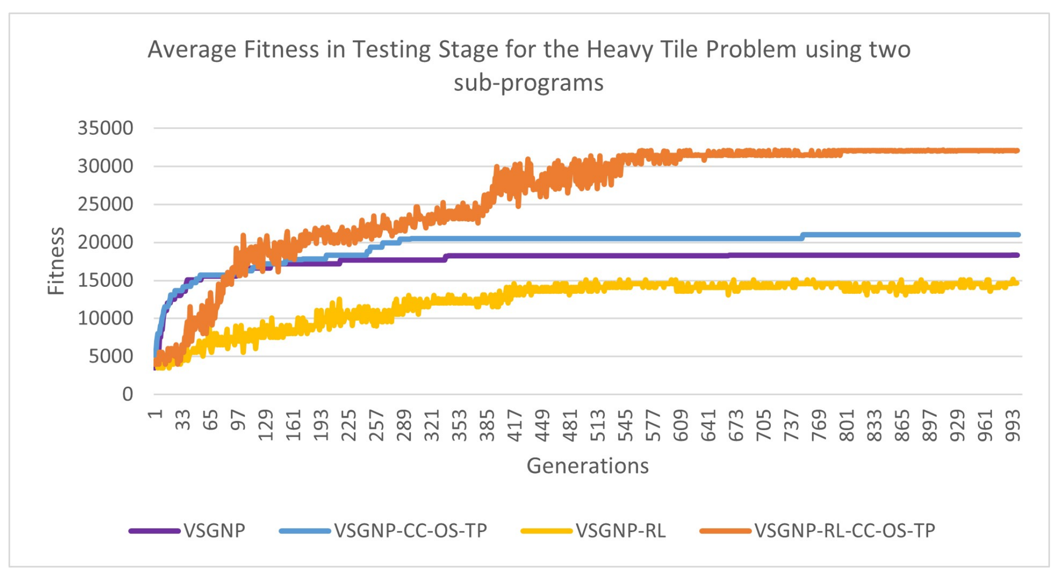


Figure 3.44: Average Fitness in Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP.

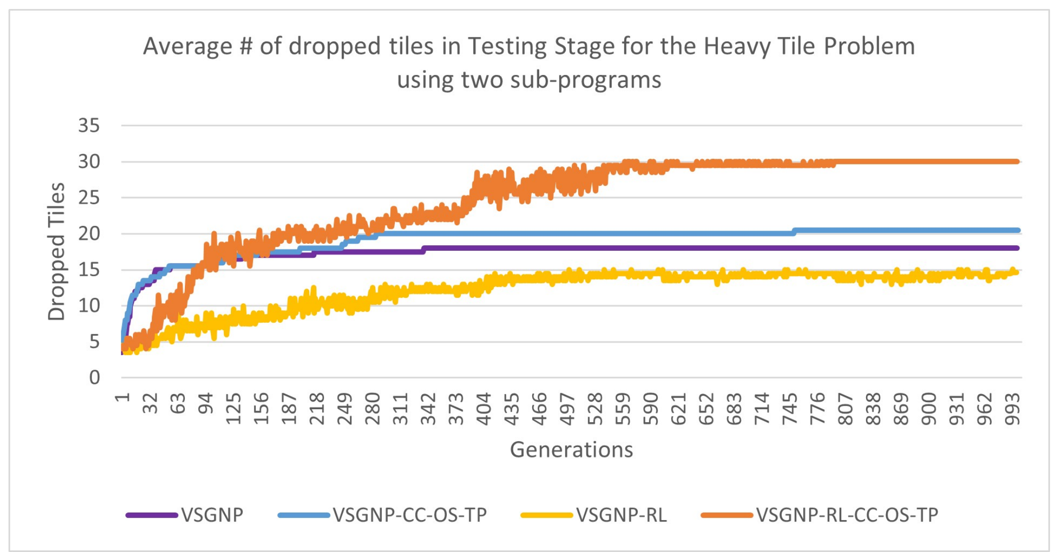


Figure 3.45: Average number of dropped Tiles in the Testing Stage for the Heavy Tile Problem with two sub-programs using VSGNP-RL-CC-OS-TP.

applying VSGNP-RL-CC-OS-TP on the Heavy Tile World Problem with two sub-programs for 1000 generations. VSGNP-RL-CC-OS-TP performed better on both the training and testing phases than the other algorithms (VSGNP, VSGNP-CC-OS-TP, and VSGNP-RL) and shows results that are superior to those when applied to the Heavy Tile World Problem. Table 3.31 shows that the maximum fitness for the VSGNP-RL-CC-OS-TP is 31951, and the best chromosome appears on generation number 661 with 30/30, 17/30, and 30/30 dropped tiles, in training set (1) and testing sets (2) and (3)

(Figure 1.2).

In this comparison, we assessed the differences between using algorithms (GNP, GNP-RL, GNP-CC-OS-TP, and GNP-RL-CC-OS-TP) for one sub-program versus using them with a variable-sized distributed graph (two sub-programs) on (VSGNP, VSGNP-RL, VSGNP-CC-OS-TP, and VSGNP-RL-CC-OS-TP). The results showed that using the variable-sized distributed graph with (GNP and GNP-CC-OS-TP) improved the results slightly, as shown in (a) and (b) (Figure 3.46) but did not show any significant improvement when used with (GNP-RL and GNP-RL-CC-OS-TP), as shown in (c) and (d) (Figure 3.46).

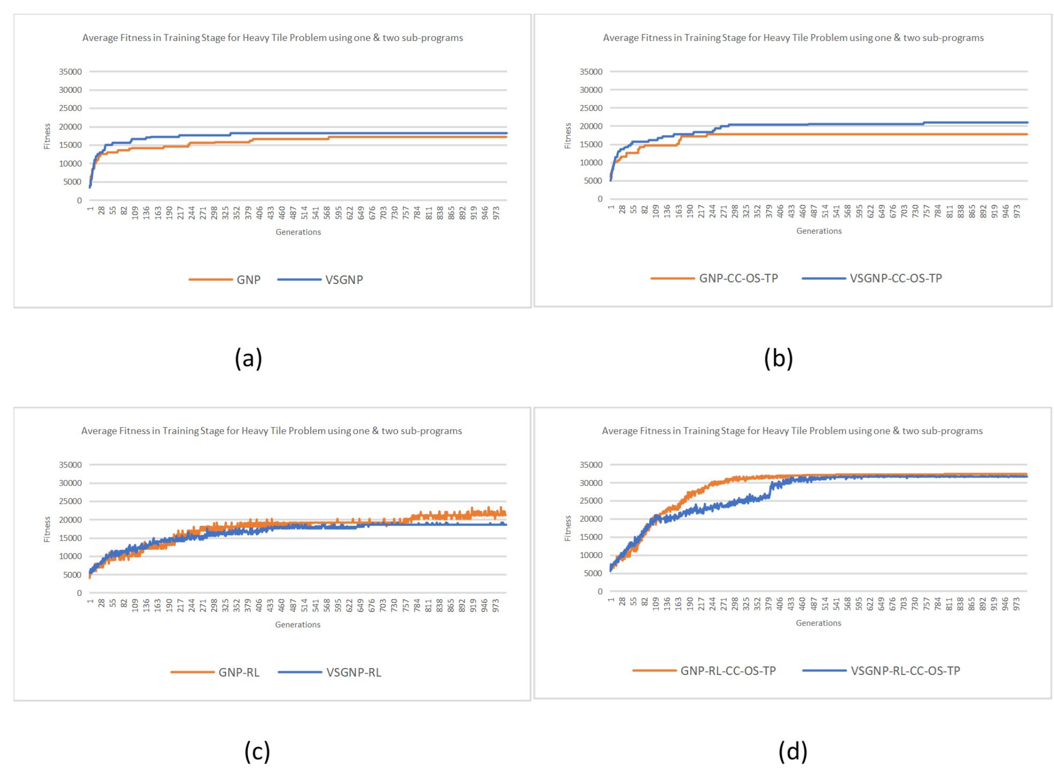


Figure 3.46: Comparing the using of one and two sub-programs with Heavy Tile World Problem for the GNP extensions.

3.5.4 Prey and Predator Problem (Proposed)

For the Prey and Predator problem, we used the same parameters used by [11] and [12].

Table 3.32: Parameters for the Prey and Predator with GNP-RL-CC-OS-TP.

Problem Domain	**Prey and Predator with GNP-RL-CC-OS-TP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent
	J2	Judge what is in back position (JB)	- Prey
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Prey (PD)?	- Forward - Backward - Left - Right - None
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Prey J5, → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node As there is no trapped location for the prey in this problem, we didn't use the constraint conformance technique in this experiment.
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Q-value	Each sub-node is initialized with a Q value of 0. Whenever the agent visits a node, it selects a single sub-node to execute. The section (Chromosome Evaluation (Fitness)) explains how the agent chooses the sub-node.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	In the GNP-RL algorithm, each node contains a number of sub-nodes, and each one of them has its ID and connections. In this experiment, each node has a maximum of four sub-nodes chosen randomly from one to four at the first population.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	50		
Number of Elites	1		
Number of Crossover individuals	20		
Number of Mutation individuals	29		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	2		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the 30 environments one by one (as in Figure 1.6). Agents begin at the starting node and proceed to the next connected nodes. When the agent visits the node, they must select one of the available sub-nodes. In order to choose a sub-node, the Sarsa algorithm with an ϵ-greedy technique is utilized, which was first implemented by Sutton and Barto [48]. The probability of randomly selecting the sub-node is set at 0.1, called exploration, and 0.9 for the sub-node with the maximum Q-value, called exploitation.

Once the node is visited, the Q-value will be updated according to equation 2.2. Q_{ip} is the Q value for the visited sub-node, Q_{jq} is the Q-value for the chosen sub-node in the next node, and the chosen (α) learning and (γ) discount rates were both 0.9 in this work. When the prey is surrounded by the predators, the reward is equal to 1.

The four agents start working from the first node in the graph and following the directions (connections) until the prey is rounded by all four agents or if the available number of steps is finished in which case the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 3.1, and the final fitness is calculated using equation 3.2.

The task prioritization techniques were not used since there was only one prey to

catch.

Results

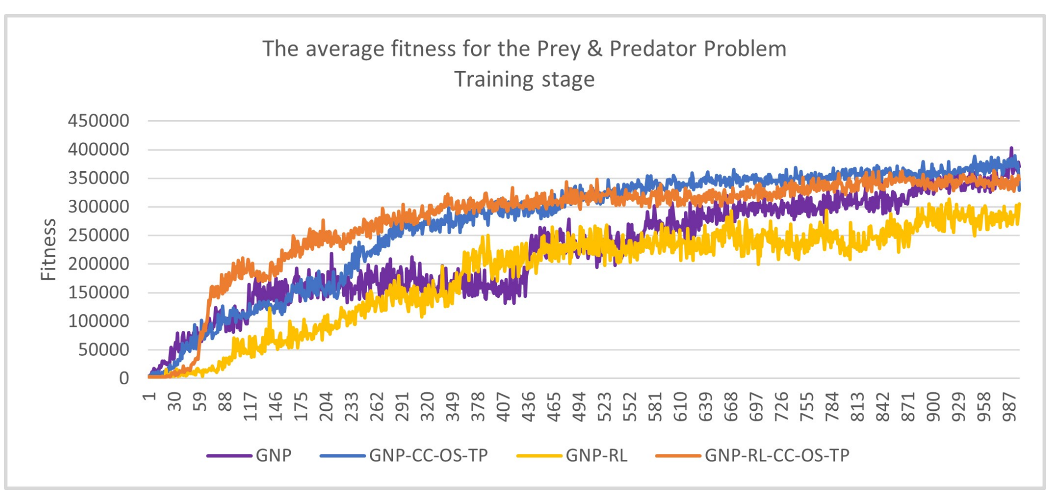


Figure 3.47: The average fitness for the Prey and Predator Problem – Training Stage using GNP-RL-CC-OS-TP.

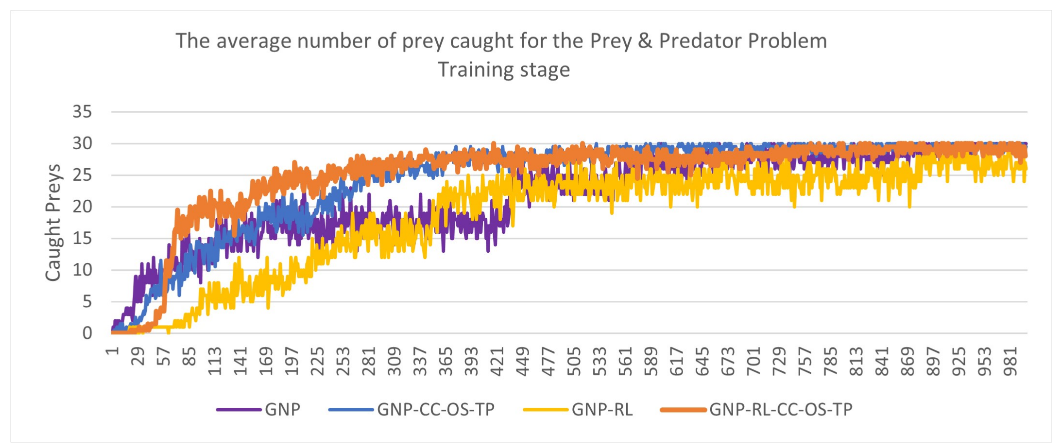


Figure 3.48: The average number of caught prey for the Prey and Predator Problem – Training Stage using GNP-RL-CC-OS-TP.

Table 3.33: GNP-RL-CC-OS-TP results on the Prey and Predator Problem.

Algorithm	Mean of Fitness	Max of Fitness	Min of Fitness	Best training results and Best Chromosome	Average # evaluation until get the top training result	# evaluation until get the best chromosome	# generation for the best chromosome
GNP-RL-CC-OS-TP	279561	363331	2745	829/900 27/30	20900	50000	999

Figure 3.47, Figure 3.48, Figure 3.49, and Figure 3.50 show the average Fitness and

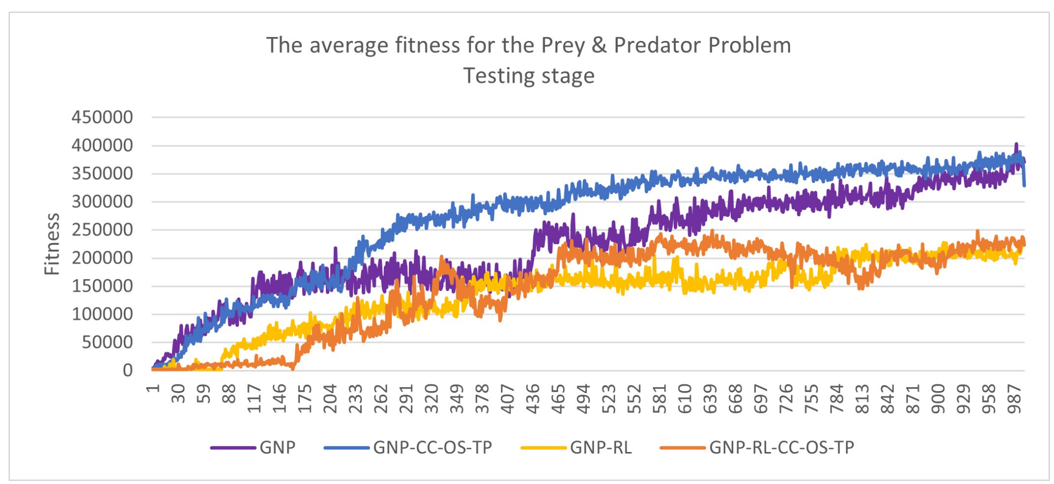


Figure 3.49: The average fitness for the Prey and Predator Problem - Testing stage using GNP-RL-CC-OS-TP.

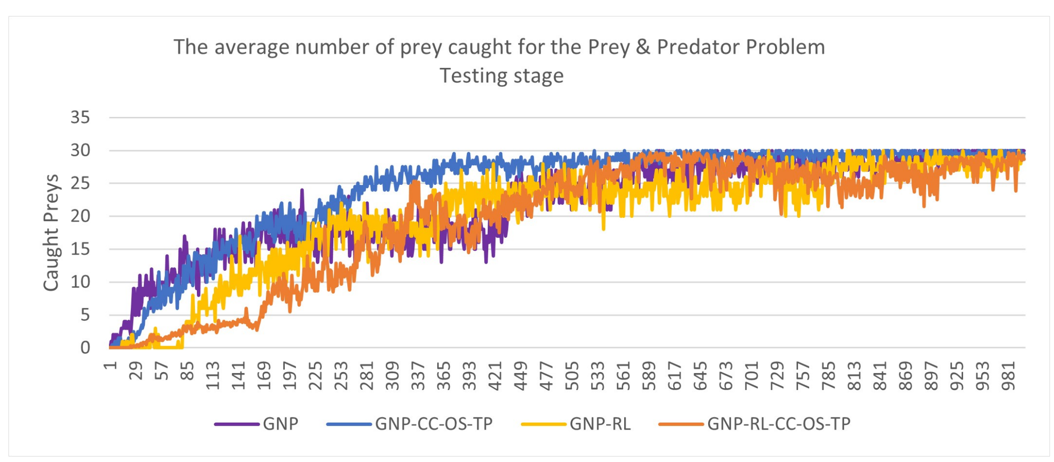


Figure 3.50: The average number of caught prey for the Prey and Predator Problem - Testing stage using GNP-RL-CC-OS-TP.

caught prey on the training and testing results for three experiments with 1000 generations for each when applying GNP-RL-CC-OS-TP on the Prey and Predator Problem. The results in GNP-RL-CC-OS-TP differ between the training and testing stages, primarily due to the exploration and exploitation phases during training. Furthermore, the environment contains prey that moves in a random pattern. The results for the GNP-RL-CC-OS-TP were less than the GNP-CC-OS-TP in the training results and less than the GNP and GNP-CC-OS-TP in the testing results. Table 3.33 shows that the maximum fitness for the GNP-RL is 363331, and the best chromosome appears on generation number 999 with an average of 27/30 caught prey.

3.6 Summary

In this chapter, we discussed the methodology and results of various algorithms including GNP, VSGNP, GNP-CC-OS-TP, VSGNP-CC-OS-TP, GNP-RL, VSGNP-RL, GNP-RL-CC-OS-TP, and VSGNP-RL-CC-OS-TP. These algorithms were applied to solve the Tile World Problem, Heavy Tile World Problem, and Prey and Predator Problem. In the next chapter, we will illustrate the first proposed approach in this thesis.

Chapter 4

Using Private Conflict Kernels Inside GNP

4.1 Motivation

GNP is a stochastic algorithm that relies on biased probabilities in the selection, crossover, and mutation operations to find a solution. In this chapter, we aim to add a systematic strategy to this stochastic approach to make the algorithm faster and better at finding the best solution. The idea is to extract the subsets of nodes that cause a conflict and compile them into conflict kernels. In turn, we use these kernels to improve the evolutionary mechanisms and prevent the conflict-generating structures in the graph from appearing in the next generations of chromosomes. This approach was inspired by the Model-Based Diagnosis (MBD) methodology [59], which was initially introduced in the General Diagnostic Engine (GDE) [60].

4.2 Related work

Constraint satisfaction is a problem-solving technique that Artificial Intelligence utilizes to tackle a diverse range of issues. Many problems can be framed as a Constraint Satisfaction Problem (CSP) and many algorithms have been proposed, some specialising for a subset of these problems. As an example, WALKSAT and GSAT [61] are stochastic-based algorithms with incremental-repair strategies for solving boolean satisfiability problems that are framed as a CSP. Wang et al. were inspired to utilise the same incremental repair principles and employed them in GNP with great success [39]. In addition, diagnostic problems can also be framed as a CSP. Successfully diagnosing faulty components in a device is crucial for diagnostic tasks. This can be achieved by observing the device's symptomatic behaviour. In these diagnostic problems, the goal is often to identify the values or states of a set of variables that are consistent with a

given set of observations and rules, while satisfying certain constraints.

To effectively initiate the GDE process, it is imperative to search for all conflicts in parallel. In other words, these are the smallest partial assignments that result in an inconsistency. Through the merging of conflicts, compact descriptions of feasible states can be generated, which are called kernel diagnoses. This approach will ensure that no potential conflicts are overlooked and will enable a thorough assessment of the situation, which will reduce the size of the active search space. This early approach has a significant limitation, that is, in many cases, only a few optimal solutions are required rather than an exhaustive list. Regarding this matter, the parallel generation of all solutions and conflicts can be significantly wasteful. The situation is made worse by the fact that conflicts and kernel diagnoses increase rapidly in a worst-case scenario. Because of that, during the 1990s, the use of GDEs declined and has been replaced by utilizing methods that concentrate on a small subset of diagnoses by listing the state space in best-first order [62] [63] [64] [65]. Model-Based Diagnosis (MBD) is a diagnosis system that relies on Artificial Intelligence [60] [66].

In model-based diagnosis, the concepts of conflict and kernel diagnosis are utilized to remove inconsistent subspace around each state diagnosis [60] [67]. MBD depends on the diagnosed system model that is used to mimic the way the system behaves. The simulations are used to compare the observed behaviour with the simulated output to find out the conflicts that lead to failure. Then, this model can be used to detect the conflict elements inside the system. In multi-agent system models, the model includes the agents' plans, the interaction between them, and their observation.

With the appearance of using multi-agents in more complex and dynamic environments, it has become very important to find a way to interact with the failures in multi-agent systems. In some systems, agents must reach a consensus on their objectives and strategies. In other systems, agents are required to coordinate their interactions with each other. However, sometimes agents fail to interact because of losing the communication or sensory. In this case, the conflicts should be diagnosed and solved.

In this chapter, we will use the technique of (MBD) to extract the conflicts from the GNP graph and use these conflicts to improve the evolutionary operation (Selection, Mutation, Crossover). Moreover, in this study, we employ similar incremental repair techniques proposed by Wang et al. [39] to repair conflict-generating structures.

4.3 Proposed Algorithms (Architecture)

Our methodology is to find the conflict substructs (part or series of nodes from the graph that lead to a conflict) and then add these extracted substructs to a private-conflict-kernel for each individual. The GNP graph is a networking graph that has

many nodes which are connected and could be visited more than one time, so the extracted conflicts from the graph are a sub-structure that led to a conflict.

We used for each chromosome a private-conflict-kernel that contains all the sub-structures that have conflicts; these kernels will be used to improve the (Selection, Mutation, and Crossover) (see Pseudocode 2). In the Tile World Problem, one of the conflicts that we have noticed is when the agent pushes a tile to a trapped location, as that will lead to the tile not being able to be pushed anywhere else, and that will affect the result for all agents although it was a conflict caused by one agent. In all the experiments in this chapter, we will adopt this situation as the conflict that the algorithm will look for. These conflict kernels are augmented while evaluating the chromosomes.

Algorithm 2 GNP with Private-Conflict-Kernel

```
 1: Identify source of conflicts in problem domain
 2: Population ← Initialize the first population ( )
 3: while (MaxGeneration is not reached) AND (solution is not found) do
 4:     for all the individuals in the Population do
 5:         Evaluate (Individual)
 6:     end for
 7:     Sort(Population)                              ▷ By highest fitness value
 8:     Add Elites (Population) to Next Generation
 9:     Selection (Population)
10:     Crossover (Population)
11:     Mutation (Population)
12:     Population ← Next Generation
13: end while
```

Each sub-structure ends with a processing node. The processing node is the one that is actually getting the reward or punishment after taking an action. Thus, the algorithm explores the graph node by node. If the node is not a processing node, it will add it to a structure to complete the sub-structure. Otherwise, if it is a processing node, it will check if it leads to a trapped tile; it will add this sub-structure to the private-conflict-kernels, considering avoidance of future repetition; then it will start a new sub-structure after that (see Pseudocode3).

In GNP, selecting the best individuals is usually based on the fitness value of each chromosome. However, in this approach, another factor is considered: the number of conflict-generating structures for the individuals. The main benefit of utilizing the private-conflict-kernel is to improve crossover and mutation operations. In the crossover, the two parents can be chosen based on the diversity of their private kernels, and the crossover point (the nodes that are crossed over on) can happen on one of the

nodes on those conflict-generating substructures to develop a new combination substructure that could resolve the conflict. Regarding mutation, there are two ways to use the private kernel: the first one is for choosing the mutation gene (node) from the conflict-generating structures. The second way is to use an incremental repair technique [38] [39] to be applied to these conflict-generating structures.

Algorithm 3 Evaluate(Individual)

```
 1: Private-Conflict-Kernel ← [ ]              ▷ Contains the conflict sub-structure
 2: sub-structure ← [ ]
 3: visited-node ← start node
 4: while (Not Finished) do
 5:     Add visited-node to sub-structure
 6:     if visited-node is processing-node then
 7:         if there is a conflict then
 8:             if sub-structure is not in the Private Conflict Kernel then
 9:                 Add sub-structure to Private Conflict Kernel
10:             end if
11:         end if
12:         sub-structure ← [ ]
13:     end if
14:     visited-node ← next node
15: end while
16: Calculate Fitness()
17: Return Individual with their private-conflict-kernel and Fitness
```

	Node 1			Node 2			Node 3			Node 4			Node 5			Node 6			
Sub struct #	Node type	Node id	answer	Node type	Node id	answer	Node type	Node id	answer	Node type	Node id	answer	Node type	Node id	answer	Node type	Node id	answer	Conflict
1	1	2	0	1	4	0	1	1	0	1	6	1	1	7	0	2	1	2	Trapped location
2	1	4	2	1	3	0	2	1	2										Trapped location
3	1	2	0	1	5	0	1	8	0	2	1	2							Trapped location
4	1	3	0	1	4	0	1	1	2	1	6	4	2	1	2				Trapped location

Figure 4.1: The Structure for the `Private-Conflict_Kernels` The sub-structs 2 and 3 will not be added to the conflict kernel because they contain less than five nodes..

The structure of the `private_conflict_kernel` for each chromosome:

These kernels contain of number of series of (rules) (`node_type, node_id, answer`) for each substruct node in the conflict substructure ended with a processing node that cause a conflict as in Figure 4.1.

1. Nodetype (1→Judgment), (2→Processing).
2. Nodeid is the node function which is for the judgment node:

   1→ Judge Forward

   2→ Judge Backward

   3→ Judge Left

   4→ Judge Right

   5→ the direction to the nearest Tile

   6→the direction to the nearest Hole

   7→ the direction from the nearest Tile to the nearest Hole

   8→ the Tile type (normal or Heavy) this function has been changed from (finding the direction to the second nearest Tile) in the original Tile World problem to (the Tile type) in the Heavy Tile World version.
3. The answer is the NodeID of the node that connects to the answer to any of these judgment nodes.

So, the substruct is a series of judgment nodes with their answers that leads to a conflict (e.g. trapped location).

Before adding any node to the substruct:

1. It checks for repetitions, so there is no repeated node in the extracted substruct.
2. It does not include the turn left and turn right nodes, instead, it converts the answers of the nodes after the turns to be suitable with the agent before the turn. As we need to make each substructure contain of a series of judgment nodes and end with a (Move Forward) processing node, the appearance of (Turn Left and Turn Right) nodes inside the substruct will give a different answer for the same judgment nodes. For example, when there is an agent that has a hole in its Forward and a tile in its Right, and there are these series of nodes (JF, TR, JF). So, for the first Judgment node (JF),the answer is a hole, and then the agent will turn right, which makes the answer for the second (JF) is tile. In this case, we will not add the (TR) to the substructe instead we will change the second (JF) to be (JR) and add it with its answer (Tile) to the substruct. This method will

ensure that there is no repetition in the judgment nodes in the substruct and will ensure that the series of rules are clear and balanced.

3. The last node for the sub-structs is the Go Forward node when there is a trap.

4. The minimum size for the substructs is 5 (Judgment) nodes with a go-forward processing node.

Before adding the substruct to the kernel (`private_conflict_kernel`):

1. It checks for any possible duplicates.

2. It makes sure that the number of the nodes in the substruct is 5 or more; this is to prevent adding a small-sized substruct that is not helpful.

Once the kernels have been created, they are utilized to optimize the performance of GNP by implementing these techniques:

1. Using the Private-Conflict-Kernels on the Selection (PCK-GNP (S)).

2. Using the Private-Conflict-Kernels on the Crossover (PCK-GNP (C)).

3. Using the Private-Conflict-Kernels on the Mutation (PCK-GNP (M)).

4.3.1 Using the Private-Conflict-Kernels on the Selection (PCK-GNP (S))

In the selection stage, the GNP chooses a specific number of elites to pass on to the next generation without changing them. Usually, these elites are selected based on their fitness value ranking. In the Tile World problem, for each generation, the best five individuals with the highest fitness value are passed on to the next generation. In the proposed algorithm, using the conflict kernels, the algorithm will first choose the best five elites depending on their fitness value and then select another (E2) five individuals (from a pool of the best (E1) ten individuals) with the lowest number of conflict sub-structures in their conflict kernels (see Pseudocode 4).

Algorithm 4 Selection (Population)

```
1: Pool ← Best E1 Individuals in Population
2: Sort(Pool)                              ▷ by lowest Private-Conflict-Kernel size
3: Return The first E2 individuals in the sorted Pool
```

4.3.2 Using the Private-Conflict-Kernels on the Crossover (PCK-GNP (C))

The conflict kernels can be used to improve the crossover operation by choosing the parents that have diversity in their private-conflict-kernels to be crossed over together and then choosing the crossover points (from the kernel) to be crossed over between the two parents.

First, choose parent1 using tournament selection. Secondly, from a pool of random individuals, choose the individual that shares the lowest number of conflict substructures with parent1 to be parent2. After choosing the two parents, choose a random node from the conflict kernel of parent1 to be the crossover point with another random node from parent2 (see Pseudocode 5).

Algorithm 5 Crossover (Population)

```
1:  for i=1 to Population_crossover/2 do
2:      Parent1 ← Tournament-Selection( )
3:      Parent2 ← Conflict-Diversity(Parent1)
4:      if Parent1.private-conflict-kernel.size()> 0 then
5:          With probability Pc
6:          Node1 ← random node from Parent1.private-conflict-kernel
7:          Node2 ← random node from Parent2
8:      else
9:          Node1 ← random node from Parent1
10:         Node2 ← random node from Parent2
11:     end if
12:     Offsprings ← Crossover(Parent1.Node1 , Parent2.Node2)
13:     Add Offsprings to Next Generation
14: end for
```

4.3.3 Using the Private-Conflict-Kernels on the Mutation (PCK-GNP (M))

The conflict kernels can be used to improve the mutation operation by choosing a mutation gene randomly with a probability of ($CKm = 0.1$) from the conflict kernels and then applying the incremental repair technique to this gene. The following are the basic steps: firstly, randomly choose one of the nodes from the private conflict kernel. If the node is a judgment node, apply incremental repair on this node ID by choosing a new value from Judgement Node IDs (i.e. 1 to 8) based on the best improved fitness. If the node is a Processing node, apply incremental repair on this node ID by choosing a value from Processing Node IDs(i.e. 1 to 4) based on best fitness improvement. Secondly, choose the node ID for this node that has the maximum fitness (see Pseudocode 6).

Algorithm 6 Mutation (Population)

```
1: for i=1 to Population_mutation do
2:     Parent1 ← Tournament-Selection( )
3:     if randomNum() < probability CKm then
4:         if Parent1.private-conflict-kernel.size()> 0 then
5:             SelectedNode ← random node from Parent1.private-conflict-kernel
6:             if SelectedNode.Type is Judgment Node then
7:                 SelectedNode ← arg max_{k∈JudgementNodeIDs} Fitness(SelectedNode_k)
8:             else if SelectedNode.Type is Processing Node then
9:                 SelectedNode ← arg max_{k∈ProcessingNodeIDs} Fitness(SelectedNode_k)
10:            end if
11:        else
12:            Node1 ← random node from Parent1
13:            With probability Pm, randomly change the connection of SelectedNode
14:            With probability Pm1, randomly change the ID of SelectedNode
15:        end if
16:        Add Parent1 to Next Generation
17:    end if
18: end for
```

4.3.4 How to prevent losing diversity in the PCK-GNP

1. Selection: part of the elite, which are passed to the next generation, are selected depending on their Private-Conflict-Kernel size. That will ensure the diversity of the next generation as not all the passed individuals are ranked on their fitness value, but some will be ranked on both the fitness value and the Private-Conflict-Kernel size as explained in Pseudocode 4.

2. Crossover: the selected parents for the crossover operation are chosen based on the diversity between their Private-Conflict-Kernels. This ensures the impact of crossover in enhancing population diversity (as in Pseudocode 5).

3. Mutation: An incremental repair technique is applied to a selected node within the Private-Conflict-Kernel to ensure that any changes made are useful and improve the overall algorithm (as in Pseudocode 6).

4.4 Testing and Analysis

The proposed algorithms have been tested on the Tile World Problem with its two variants (the normal and heavy tile)

4.4.1 Solving the Tile World Problem, PCK-GNP

For the Tile World problem, we used the parameters utilized in [17] and [14] as they have been empirically derived to perform well on the domain. Table 4.1 shows the complete parameter settings with descriptions.

Table 4.1: Parameters for the Tile World Problem: PCK-GNP.

Problem Domain	**Tile World Problem with PCK-GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole; J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals, $Population_{\text{crossover}}$	120		
Number of Mutation individuals, $Population_{\text{mutation}}$	175		
Mutation rate	Pm: 0.1, Pm1: 0.01, CKm: 0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness): To evaluate any chromosome, the algorithm runs on the ten environments one by one (as in Figure 1.1(1)). The three agents start working from the first node in the graph and follow the directions (connections) until either all three tiles drop into the three holes or if the available number of steps is finished, the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 2.4.

The $D_{distance}$ is calculated using the total of the below points:

- 4 points → when the agent pushes the nearest tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

- 2 points → when the agent pushes the nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.

- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.

- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.

- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 10 points → when the agent pushes a normal tile into a hole.

While the chromosome is evaluated the conflict sub-structs are extracted and added to the private-conflict-kernel for each individual.

Results

Figures 4.2 and 4.3 show the different results between the different variations of using Private-Conflict-Kernels with GNP in the Tile World Problem. There are seven different ways of applying the Private-Conflict-kernels on the GNP. We tested using Private-Conflict-kernels on the selection, crossover, and mutation alone. Then we tested using it with selection & mutation, selection & crossover, crossover & mutation, and selection & crossover & mutation. Each one of the results is an average of three experiments when applying it to the Tile World problem for 1000 generations. The results indicate that (crossover & mutation, selection & mutation, crossover, and mutation) were the algorithms that could reach the top training results (30/30) dropped tiles within the first 1000 generations, while the other algorithms could not. Using the Private-Conflict-Kernels with GNP in the (crossover & mutation) was the fastest algorithm that reached the top. Table 4.2 compares the seven algorithms, illustrates the statistics behind each one, and shows the best chromosome. The crossover & mutation algorithm received the first rank with the maximum mean of fitness (30705). The best chromosome for this algorithm was able to reach (30/30), (19/30), and (30/30) for the set (1), (2), and

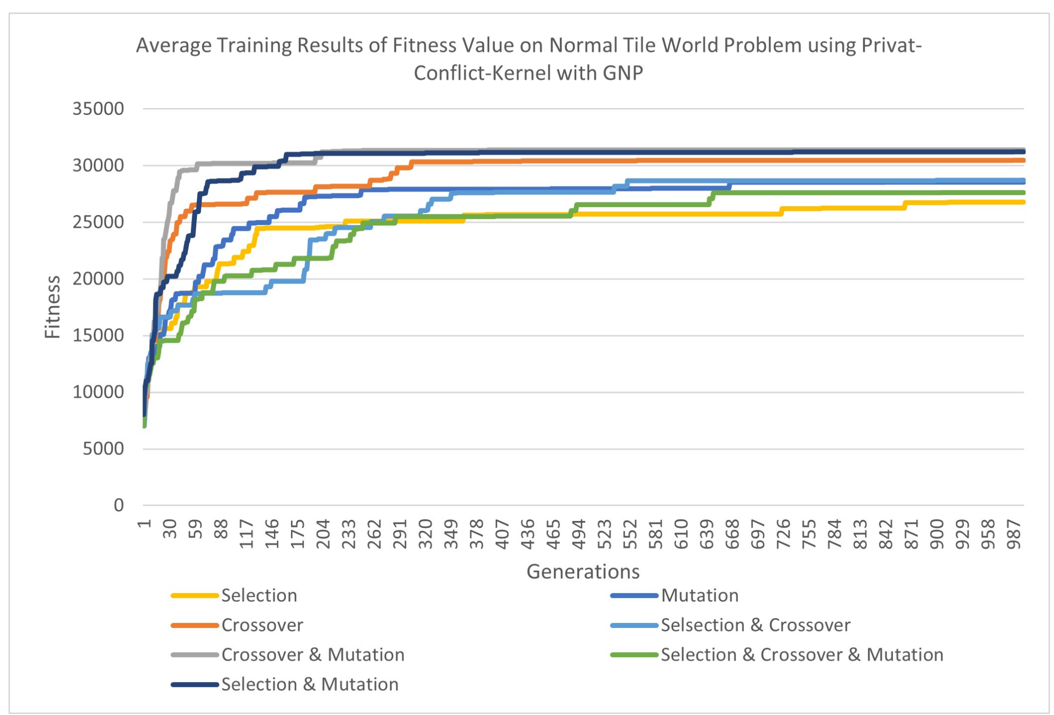


Figure 4.2: Average Fitness During Training on Tile World Problem: PCK-GNP.

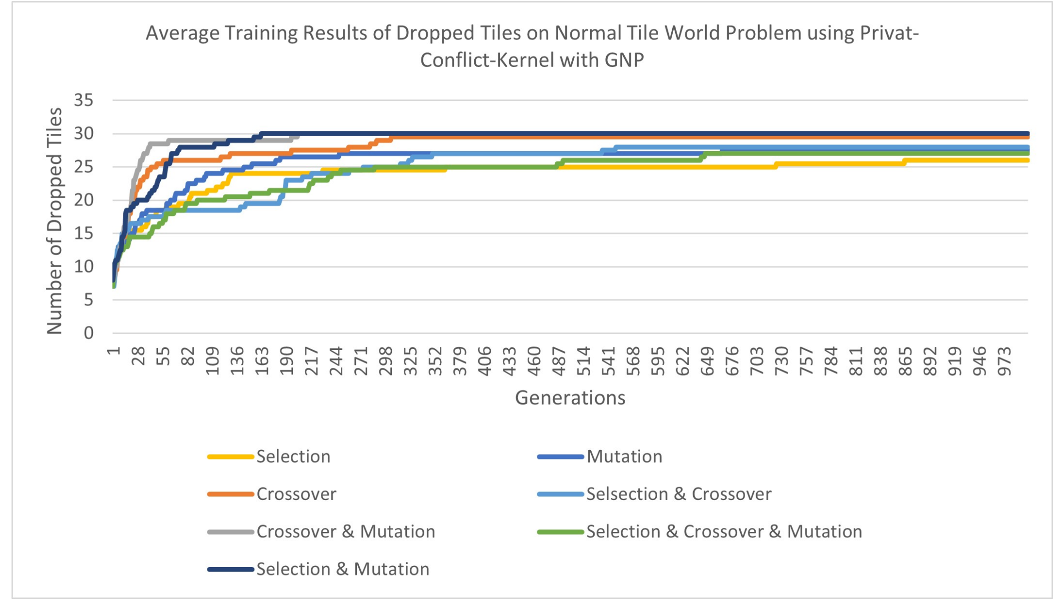


Figure 4.3: Average number of dropped tiles during training on Tile World Problem: PCK-GNP

(3), respectively, within 85 generations that needed 255000 fitness evaluations. After that, selection & mutation with a mean of fitness of 30225 and a p-value compared to (crossover & mutation) equals 0.00E-00, which explains the big difference between the

Table 4.2: Comparison of results on Tile World Problem: PCK-GNP

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Average Number evaluation until get the top training result	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
Selection(S)	24806	7	0.00E-00	26788	8017	27/30 09/30 21/30	-	267900	892
Crossover (C)	29200	3	9.79E-33	30467	8519	30/30 15/30 24/30	91500	186300	620
Mutation (M)	26935	4	3.69E-151	28533	7011	30/30 23/30 30/30	-	221100	736
Selection & Crossover (S&C)	25944	5	3.11E-172	28702	7011	28/30 12/30 13/30	-	191100	636
Selection & Mutation (S&M)	30225	2	0.00E-00	31214	8011	30/30 13/30 25/30	46500	61200	203
Crossover & Mutation (C&M)	30705	1	-	31411	8011	30/30 19/30 30/30	58800	25800	85
Selection & Mutation & Crossover (S&M&C)	24928	6	1.76E-263	27617	7011	28/30 4/30 14/30	-	225000	749

fitness values for their training results. The algorithm was able to find the most effective chromosome in generation number 203 that needs 60900 fitness evaluations that could reach (30/30), (13/30), and (25/30) dropped tiles for set (1), set (2), and set (3) respectively. The third-ranking algorithm is (crossover) with a mean of fitness equal to 29200 and a p-value of 9.79E-33, which illustrates the significant difference between it and the best algorithm (crossover & mutation). This algorithm was able to get the best individuals within 620 generations with succeeded dropped tiles (30/30), (15/30), and (24/30) for the set (1), set (2), and set (3), respectively. The (mutation) algorithm was the fourth-ranked one. However, it received the best individual compared to the other variations of the algorithm with (30/30), (23/30), and (30/30) dropped tiles for the set (1), (2), and (3), respectively. Although the average training results for this algorithm do not reach the top, the best individual produced by this algorithm is the best solution among all algorithms. The reason for this is that while some experiments of this algorithm are able to reach the top, others could not succeed within the first 1000 generations due to the algorithm's randomization. The mentioned factor has a crucial impact on the performance of the algorithm. The remaining algorithms were not able to succeed in dropping all the tiles during the training stage within the first 1000 generations.

4.4.2 Solving the Heavy Tile World Problem, PCK-GNP with one subprogram

The parameters for the Normal Tile World problem remain the same, but with a few modifications. Specifically, the Judgment node number 8 has been updated from providing the direction to the second nearest tile to checking whether the nearest tile is a normal or heavy tile.

Table 4.3: Parameters for the Heavy Tile World Problem: PCK-GNP with one subprogram.

Problem Domain	**Heavy Tile World Problem, PCK-GNP wiith one subprogram**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile; J5, J6, J7 → Forward, Backward, Left, Right, Nothing; J8 → Normal, Heavy		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01, CKm:0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the ten environments one by one (as Figure 1.2(1)). The three agents start working from the first node in the graph and follow the directions (connections) until all three tiles drop into the three holes or if the available number of steps is finished, the algorithm will close the current environment and start the next one. Then, the fitness value is calculated by equation 2.4.

The $D_{distance}$will be calculated using the total of the below points:

- 8 points $\rightarrow$ when the agent pushes the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 4 points $\rightarrow$ when the agent pushes the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 4 points $\rightarrow$ when the agent pushes the nearest Normal tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 4 points $\rightarrow$ when the agent pushes not the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 2 points $\rightarrow$ when the agent pushes not the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 2 points $\rightarrow$ when the agent pushes the nearest Normal tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 2 points $\rightarrow$ when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point $\rightarrow$ when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point $\rightarrow$ when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point $\rightarrow$ when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 10 points $\rightarrow$ when the agent pushes a normal tile to a hole.
- 20 points $\rightarrow$ when the agent pushes a heavy tile to a hole.

While the chromosome is evaluated, the conflict sub-structs are extracted and added to the private-conflict-kernel for each individual.

Results

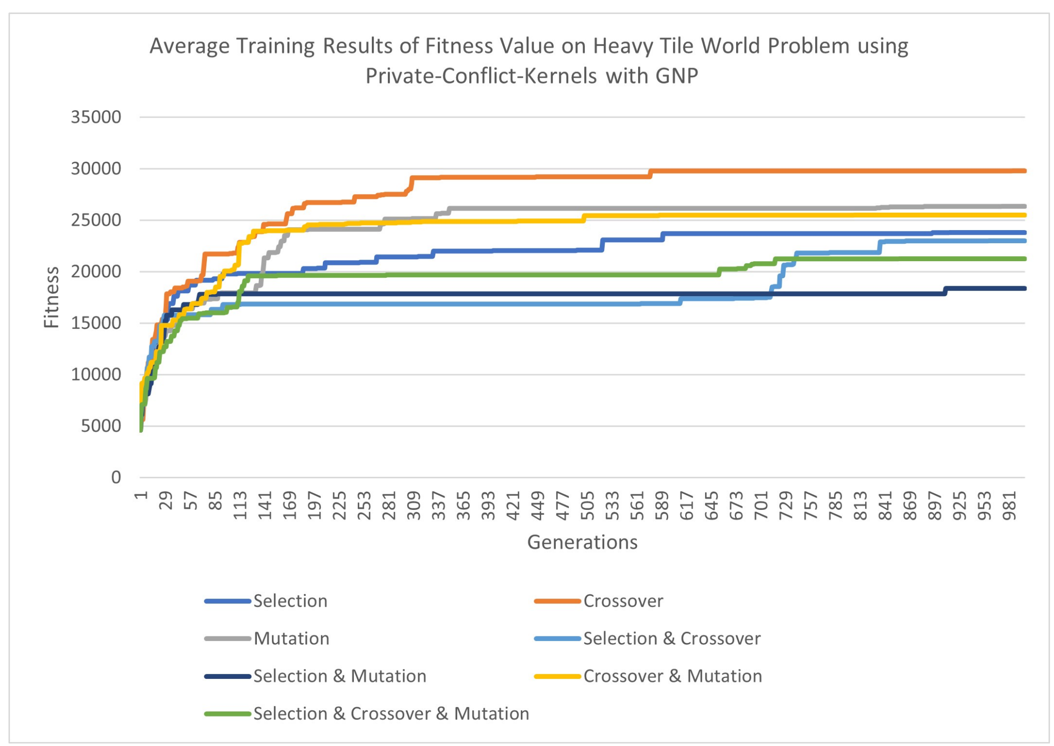


Figure 4.4: Average fitness during training on Heavy Tile World Problem, PCK-GNP with one subprogram.

Figures 4.4 and 4.5 illustrate the varying outcomes among the different versions of using Private-Conflict-Kernels with GNP in the Heavy Tile World Problem with one sub-program. The seven different algorithms have been applied to the heavy Tile World problem with one sub-program (selection, crossover, mutation, selection & crossover, selection & mutation, crossover & mutation, and selection & crossover & mutation). Each result averages three experiments for the Heavy Tile World problem over 1000 generations. The results indicate that (selection, crossover, and mutation) were the algorithms that could reach the top training results (30/30) dropped tiles within the first 1000 generations, while the other algorithms could not. Using the Private-Conflict-Kernels with GNP in the (crossover) was the fastest algorithm that reached the top. Table 4.4 compares seven algorithms and illustrates the statistics behind each one, highlighting the best chromosome. The (crossover) algorithm received the first rank with the mean of fitness (27603). The best chromosome for this algorithm was able to reach (30/30), (10/30), and (26/30) for the set (1), (2), and (3), respectively, within 770

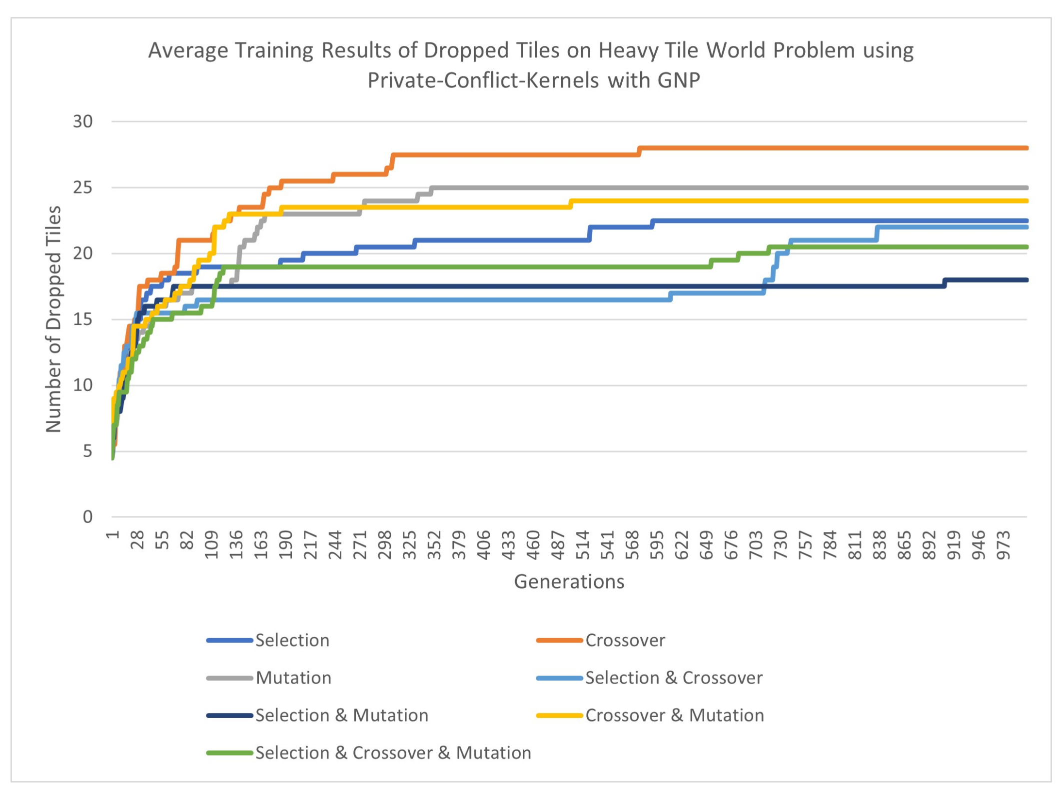


Figure 4.5: Average Number of Dropped tiles on Heavy Tile World, PCK-GNP with one subprogram.

generations that needed 231300 fitness evaluations. After that, (mutation) with a mean of fitness of 24351 and a p-value compared to (crossover) equals 6.67E-72, resulting in significant differences in training results. The algorithm was able to find the most effective chromosome in generation number 851 that needs 255300 fitness evaluations that could reach (30/30), (13/30), and (30/30) dropped tiles for set (1), set (2), and set (3), respectively. The algorithm that has the third ranking is (crossover & mutation) with a mean of fitness equal to 24086, and a p-value of 6.77E-95 clearly shows that there is a significant difference between it and the best algorithm (crossover). This algorithm was able to get the best individual within 330 generations with succeeded dropped tiles (29/30), (14/30), and (27/30) for the sets: set (1), (2), and (3), respectively. The (selection) algorithm was the fourth-ranked one. However, it received the top training result in set (1) with (30/30) dropped tiles, (9/30), and (14/30) for set (2) and set (3) respectively. The remaining algorithms were not able to succeed in dropping all the dropped tiles for the remaining stage within the first 1000 populations.

From the results, we noticed that some of the algorithms, such as (selection and mutation) could not reach the top training results when averaging their experiences. At the same time, their best experiment received 30/30 dropped tiles in the training environments. This proves that even the `private_conflict_kernels` can increase the

Table 4.4: Comparison of Results on Heavy Tile World Problem: PCK-GNP with one subprogram.

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
Selection(S)	21918	4	1.67E-236	23800	4584	30/30 09/30 14/30	243000	809
Crossover (C)	27603	1	-	29817	4596	30/30 10/30 26/30	231300	770
Mutation (M)	24351	2	6.67E-72	26347	5596	30/30 13/30 30/30	255600	851
Selection & Crossover (S&C)	18270	6	0.00E+00	23028	5606	28/30 09/30 16/30	285000	949
Selection & Mutation (S&M)	17635	7	0.00E+00	18380	4586	19/30 05/30 13/30	278100	926
Crossover & Mutation (C&M)	24086	3	6.77E-95	25508	6120	29/30 14/30 27/30	99300	330
Selection & Mutation & Crossover (S&M&C)	19539	5	0.00E+00	21261	4593	21/30 07/30 13/30	227100	756

results and improve the GNP algorithm. However, randomization is still playing a huge part in the algorithm, and we can notice that from the different results for each experiment for the algorithms.

4.4.3 Solving the Heavy Tile World Problem: PCK-VSGNP with two subprograms

Using variable-sized Genetic Networking Programming (VSGNP), we divided the graph into two sub-programs. One sub-program solves normal tiles, and the other sub-program solves the heavy tiles.

Table 4.5: Parameters for the Heavy Tile World Problem: PCK-VSGNP with two subprograms .

Problem Domain	**Tile World Problem with Heavy Tile Using two sub-program with PCK-GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile** - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal → sub-program 0; Heavy → sub-program 1 For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.		MF → With Conflict, Without Conflict TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
	Chromosome Structure		
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.		
	Algorithm Parameters		
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01, CKm:0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

The same process that is used to evaluate the chromosome with one sub-program is used with two sub-programs.

Results

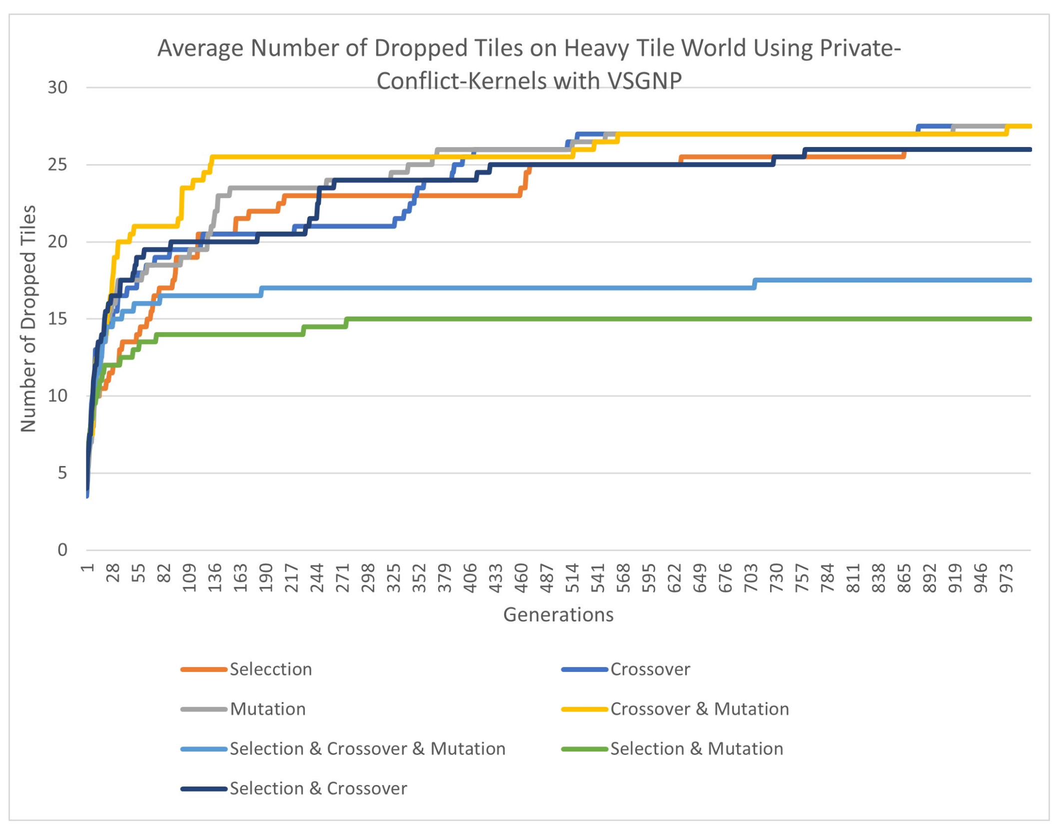


Figure 4.6: Average Number of Dropped Tiles on Heavy Tile World, PCK-VSGNP with two subprograms.

Figures 4.6 and 4.7 illustrate the varying outcomes among the different versions of using Private-Conflict-Kernels with GNP in the Heavy Tile World Problem with two sub-programs. The seven different algorithms have been applied to the heavy Tile work problem with two sub-programs (selection, crossover, mutation, selection & crossover, selection & mutation, crossover & mutation, and selection & crossover & mutation). Each result averages three experiments for the Heavy Tile World problem over 1000 generations. The results indicate that (selection, crossover, mutation, and crossover & mutation) were the algorithms that could reach the top training results (30/30) dropped tiles within the first 1000 generations, while the other algorithms could not. Using the Private-Conflict-Kernels with GNP in the (crossover & mutation) was the algorithm with the highest mean fitness. Table 4.6 illustrates the statistics behind

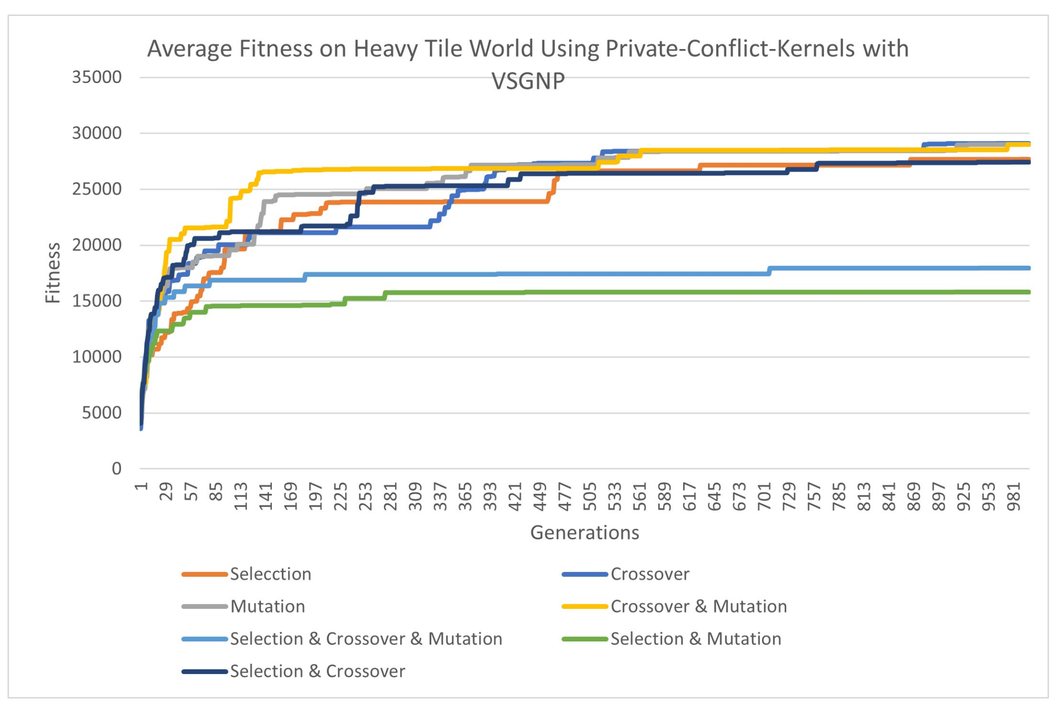


Figure 4.7: Average Fitness on Heavy Tile World, PCK-VSGNP with two subprograms.

each one, highlighting the best chromosome. The (crossover & mutation) algorithm received the first rank with the mean of fitness (26722). The best chromosome for this algorithm was able to reach (30/30), (12/30), and (29/30) for the set (1), (2), and (3), respectively, within 281 generations that needed 84300 fitness evaluations. After that, (mutation) with a mean of fitness of 25954 and a p-value compared to (crossover) equals 1.68E-06, resulting in significant differences in training results. The algorithm was able to find the most effective chromosome in generation number 998 that needs 299700 fitness evaluations that could reach (30/30), (8/30), and (23/30) dropped tiles for the sets: set (1), set (2), and set (3) respectively. The algorithm that has the third ranking is (crossover), with a mean of fitness equal to 25281 and a p-value of 2.77E-17, which clearly shows that there is a significant difference between it and the best algorithm (crossover & mutation). This algorithm was able to get the best individuals within 902 generations with succeeded dropped tiles (30/30), (18/30), and (24/30) for the sets: set (1), (2), and (3), respectively. The (selection & crossover) algorithm was the fourth-ranked one. It could not get the top training results, it received the results (27/30) dropped tiles in the set (1), (6/30), and (18/30) for set (2) and set (3), respectively. While the fifth-ranked algorithm (selection) was able to reach the top training results (30/30), (19/30), and (27/30) in sets (1), (2), and (3), respectively. The remaining algorithms were unable to drop all tiles in the remaining stage within the first 1000 populations. Overall, and based on our thorough analysis, we cannot

Table 4.6: Comparison of Results on Heavy Tile World Problem: PCK-VSGNP with two subprograms.

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Average Number evaluation until get the top training result	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
Selection(S)	24386	5	1.60E-43	27685	4583	30/30 19/30 27/30	-	158100	526
Crossover (C)	25281	3	2.77E-17	29080	3579	30/30 18/30 24/30	-	270900	902
Mutation (M)	25954	2	1.68E-06	29016	4076	30/30 08/30 23/30	-	299700	998
Selection & Crossover (S&C)	24877	4	3.36E-36	27399	4069	27/30 06/30 18/30	-	241800	805
Selection & Mutation (S&M)	15321	7	0.00E+00	15804	5077	15/30 01/30 08/30	-	120600	401
Crossover & Mutation (C&M)	26722	1	-	29020	5091	30/30 12/30 29/30	-	84600	281
Selection & Mutation & Crossover (S&M&C)	17272	6	0.00E+00	17936	4594	19/30 03/30 14/30	-	78300	260

confidently ascertain whether using one or two sub-programs produces better outcomes using the `private_conflict_kernels` with the GNP algorithm in the Heavy Tile World Problem.

4.5 Summary

Using private-conflict-kernels with GNP was able to get 100% training and testing accuracy in some of the experiments when applying it to the Tile World problem and Heavy Tile World problem using one-sub-program. However, there are three limitations of this approach: firstly, the complexity of performing the algorithm. Secondly, the technique does not generalise, as the extracted conflict is related to the environment. Thirdly, due to the randomization element involved, the algorithm was not successful in dropping all the tiles into the holes. So, in the next chapter, we will produce a new algorithm called Conflict-directed-A* with GNP that can strike a balance between applying stochastic techniques and optimal search, which has a systematic increment, adding to this, it will generalize the extracted conflicts to be from the chromosome (graph) instead of the environment.

Chapter 5

Using Conflict Directed A* Inside GNP Graph

5.1 Motivation

Genetic algorithms make use of stochastic techniques in its core. The benefit of stochastic optimization is that it is able to explore interesting areas in the search space faster than using brute force search. However, using an element of randomness depends on luck and could sometimes lead to losing a grasp of good solutions. The selection, Crossover, and Mutation still have some randomization in their work. Hence, we need to reduce the randomization in this algorithm by using the Optimal Search A*. We have incorporated the optimal search into the GNP technique, enabling us to determine the most effective combination of processing nodes within the graph to achieve optimal results. The goal of using optimal search with GNP is to ensure that we implement effective changes and enhance the chromosome to the highest attainable level prior to passing it on to the next generation.

We chose CDA* because it was designed with a search tree, making it suitable to work in a networking graph. However, adapting CDA* from a tree structure to a networking graph was a challenge. In addition, our findings from the PCK-GNP showed that using extracted conflicts was effective in improving the performance of the GNP algorithm. This led us to search for general conflicts that could be applied to any problem. As a result, we decided to extract conflicts from the networking graph. This approach makes the CDA* the best optimal search algorithm for this mission, as it uses conflicts to find the best solution more quickly. We used the CDA* to thoroughly explore all potential functions for the processing nodes in the GNP graph. By excluding conflicting functions and finding the best combination of functions for the processing nodes, we can maximize the graph's fitness value and significantly reduce evolution time compared to randomly changing the functions for the nodes.

This strategy aims to minimize exploration time by eliminating conflict-generating paths from consideration. This chapter proposes a novel algorithm named Conflict-Directed A* with Genetic Network Programming (CDA*-GNP).

5.2 Related work

Optimal search algorithms are designed to find the shortest path between a start state and a goal state and typically use heuristic methods to estimate the path. In 1971, Nilsson developed the A* search algorithm that uses heuristics to take advantage of environmental feedback to make the process of finding the shortest path with an A* algorithm faster [68] by estimating the path cost between an object and the goal, then finding the shortest path depending on this estimation. The heuristics could be found in many ways, such as Manhattan distances or Euclidean distances [69]. A* works to explore all the nodes that connect the start node with the goal node by calculating the cost for the paths; the path with the minimum cost is the winner, and it will be chosen as the optimal path. The A* algorithm has been improved to many other algorithms such as D* [70], Lifelong Planning A* (LPA*) [71], D* Lite [72] ...etc, and has been used in many artificial intelligence applications, and with many types of Problems.

Multi-Agent Path Finding (MAPF)

In a shared multi-agent environment, finding collision-free paths for each agent is challenging. Multi-Agent Path Finding (MAPF) is a highly effective solution for finding optimal and collision-free paths for multiple agents in a shared environment [73]. Minimizing travel time ensures efficient movement of agents towards their respective goals. It has been used in many applications, such as unmanned aerial vehicles [74], autonomous vehicles [75], and autonomous warehouses [76]. Solving Multi-Agent Pathfinding (MAPF) optimally is a difficult problem that falls under the category of NP-hard [77], which means finding an optimal solution may require significant computational resources and time [78] [79] [80]. The researchers have investigated algorithms that can efficiently solve MAPF, albeit sub-optimally.

These are the top prioritized algorithms: Prioritized Planning (PP) [81] and Priority-Based Search (PBS) [82]. Each agent is assigned a unique priority by PP. PBS is assigning priorities to colliding agents lazily. PBS plans paths for agents one at a time, resolving collisions by assigning priorities and replanning during its search. Ma et al. [82] proposed Priority-Based Search (PBS), a two-level algorithm based on Conflict-Based Search (CBS). At a high level, it conducts a search on the Priority Tree (PT). When a conflict exists between the lowest cost paths for two agents, PBS performs a low-level search to replan the minimum-cost paths of the child PT nodes to meet the

new priority order. When it comes to solving multi-agent pathfinding (MAPF) problems, PBS may not be the most effective approach for instances with a high density of both agents and obstacles. PP and PBS have been proven to be effective, but they do not provide the highest level of efficiency and completeness.

Constraint optimization problems

In 2007 [83], the A* algorithm was improved by Williams and Ragno to solve the constraint satisfaction problems (CSP) and thus called Constraint-Based A*. This kind of problem has three factors: a set of variables, a set of domains for these variables, and a set of constraints on these variables [84]. The solution is to find the best variables for each state that do not conflict with the constraints. There are many examples of CSP, such as the N-queens problem [85], Sudoku [86], the map colouring problem [87], etc. To solve these types of problems, Williams and Ragno used A* to search only over the decision variables, so each search was assigned to a partial goal, which is the decision variable. After finding this partial search, the algorithm searches over the non-decision variables to check for constraints; if this search does not conflict with the constraint, it will be chosen as a solution. Otherwise, it will check for other paths.

Constraint-based A* is designed to search for the optimal solution with minimum cost or maximum reward that is represented by an extracted path from a tree node structure. It starts by adding the root node to a Q and then looking for its best child with the best f cost. The algorithm starts with an empty solution list . Firstly, it checks if it reaches the goal. If the goal is found, it will return a solution, but if the Q is empty, it will terminate. If the Q is not empty, it will pop the best node from the Q with the best f-value and will add the best expanded node from the last node to the Q. After that, the algorithm will check the popped node to see if it is complete and consistent, and if it reaches a decision state, it will be added to the solution. These steps should be repeated until the goal is found or all the nodes have been explored without finding a solution that does not conflict with the constraint. The limitation of this approach is that it will look for all the possible paths even if it has detected a conflict with the constraint before. Thus, Brian Williams et al. developed an enhancement of this algorithm called Conflict-Directed A* (CDA*) that overcame this issue.

Conflict-Directed A* adds all the explored paths that conflict with the constraint to a kernel to avoid expanding any state that has the same conflict. This will reduce the searching time [83]. Conflict-Directed A* is similar to A* in terms of having a Queue that contains all the explored paths and also having the expanded-list that contains the explored nodes to prevent expanding the same node again. Unlike A* Conflict-Directed A* has a conflict Kernel which contains the paths that conflict with one or more of the constraints. The algorithm starts with exploring the first node, adding it to the

expanded list, and adding all the neighbours' nodes, each with a separate path to the Q. After that, the algorithm pops the best path with the minimum cost or maximum reward from the Q, adds the explored node to the expanded list, and checks this path; if it reaches the goal, it checks that it does not contain any conflict so it will be chosen as the solution. Otherwise, it will be added to the conflict kernel, and the algorithm removes any path from the Q that contains this conflict. If the path does not reach the goal, it will be checked to see if it does not conflict with any constraint, so it adds all the neighbours that are not in the expanded list - each in a separate path - to the Q. Otherwise, the path will be split from the conflict node and the other neighbours for this conflict node will be added to the Q if it has not been explored before (see pseudocode 7).

Algorithm 7 Conflict-Directed A*

```
1:  Q = []
2:  Add { } to Q
3:  Γ = []
4:  expanded = []
5:  while (Q isn't empty) do
6:      assignment ← pop best from Q
7:      Add assignment to expanded
8:      if assignment is full assignment to decision variables then
9:          is_consistent, conflict = consistent? (assignment)
10:         if is_consistent then
11:             Return assignment
12:         else
13:             Add conflict to Γ
14:             Remove anything from Q that manifests conflict
15:         end if
16:     else
17:         if assignment resolves all conflicts in Γ then
18:             Xi = some decision variable not assigned in assignment
19:             neighbors = split_on_variable(assignment, xi)
20:         else if assignment does not resolve some conflict γ in Γ then
21:             neighbors = split_on_conflict(assignment, γ)
22:         end if
23:         Add each xk in neighbors to Q if not in expanded
24:     end if
25: end while
26: Return NoSolution
```

There is another way for using the conflicts, which converts the conflict kernel to a constituent kernel which contains all the assignments that resolve all the conflicts. By connecting the assignments from the constituent with the extracted state, avoiding the conflict becomes guaranteed [88]. Conflict-Directed A* has been applied to space,

naval and automotive systems, as well as space systems in flight [83] and SAT solver [89], among many others. Figure 5.1 shows the Boolean Polycell example that has been used in [83], which consists of three OR gates and two AND gates. The inputs and output are shown in the figure. Each component could be good (G) or broken (U). It consists of a set of decision variables y, each one of the decision variables has a domain of G, U, and a set of constraint Cy. For example, one of the constraints in this model "IF O1=G Then (X = 1 IFF (B=1 or C=1))"

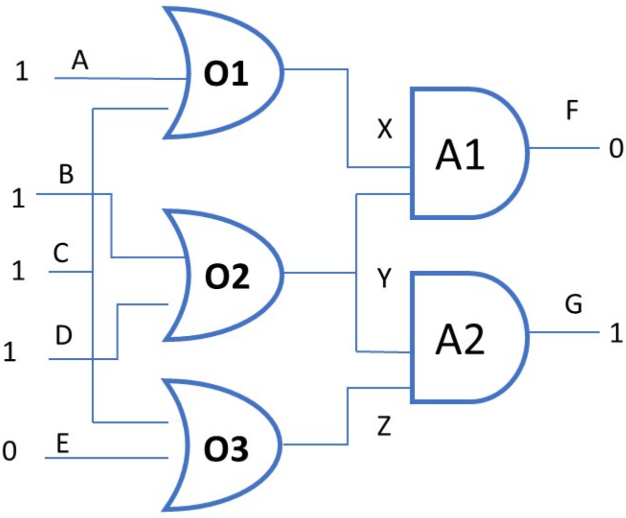


Figure 5.1: Boolean Polycell with Conflict-Directed A* example.

In 2006, Sachenbacher et al. improved the algorithm to solve soft constraints by dividing the constraints into two parts: soft and hard constraints. The hard constraint should be satisfied and the soft is used in the evaluation [90]. In 2013, Yu and Williams reformulated conflicted-directed A* to Controllable Conditional Temporal Problems, which learns the conflicts between the constraint and the assignments and then uses the resolved conflicts to direct the search away from the infeasible region [91].

Another type of algorithm that is used to find the optimal path called Ant Colony Optimization (ACO) which is a population-based optimization method inspired by observing a real ant colony. It is based on the collective behaviour of ants to share information [92] [93]. The ACO algorithm utilizes positive feedback and implements a constructive greedy heuristic. Positive feedback helps find a public solution quickly, and constructive greedy heuristics find acceptable solutions faster [92].

ACO [92] is a method that is based on the behaviour of ants when they search for

a path from their colony to food sources. As they move along, ants leave a chemical trail known as pheromone, which guides other ants towards the food. The higher the concentration of pheromone on a trail, the more likely it is that other ants will follow it. However, over time, the pheromone evaporates, causing the trail to disappear. This creates a feedback loop in the system, ensuring that the ants are always finding the most efficient path towards the food. Ants navigate their environment by following the scent of pheromones. Initially, a group of ants will explore a surface in a random manner. Once food is discovered, the ants will return to their nest and leave a trail of pheromones. Over time, the pheromone deposits build up and create distinct paths. New ants will tend to choose the path with the strongest scent of pheromones. The old trails that are not reinforced by new ants will eventually disappear due to evaporation. This feedback system promotes the use of the most effective path since any trail with a strong pheromone scent will become the preferred route. When applied to the travelling salesman problem [94], ACO utilizes two functions to guide the search towards the optimal solution:

$$H^n(r,a) = (1-p)H^{n-1}(r,a) + \sum_{m \in M} h_m^n(r,a) \tag{5.1}$$

Where $H^n(r,a)$ is the pheromone intensity on edge (r,a) at time n. p is the evaporated parameter $\in$ (0,1). M is the set of the ants on the colony, and $h_m^n(r,a)$ is the pheromone intensity added by ant m at time n. The probability of ant m choosing vertex a next, when it's at vertex r at time n, is given by this formula:

$$p_m^n(r,a) = \left\{ \begin{array}{ll} \frac{H^n \ (r,a)^\alpha \ \eta(r,a)^\beta}{\sum_{s \in M_m} H^n \ (r,s)^\alpha \ \eta(r,s)^\beta} & if \ \ a \ \in \ M_{m'} \\ 0 & otherwise \end{array} \right\} \tag{5.2}$$

Where $\eta(r,a)$ is the visibility between the vertex r and a. M_m is the set of non-visited vertex by ant m. α and β are parameters that control the visibility and importance of the trail.

Using the optimal search with GNP algorithms

In 2007 [95], Yu et al. combined the ACO with the GNP. This hybrid algorithm uses the positive feedback exploration mechanism in the GNP. They proved that this algorithm was often faster than the GNP. The similarity between ACO and GNP is that each individual is represented by an ant. The tour was made by the ant represented by the transition of GNP. The connections between the nodes in the GNP represent the trail edge. The individual that has the best fitness is equivalent to the shortest path that is found by the ant. The algorithm starts with initializing the first population that has a

number of individuals which are identical in the number of each kind of node but with connections that are randomly set. Then the algorithm evaluates the individuals and calculates the fitness value for each of them. In this stage, the pheromone intensity is updated too. After that, the evolutionary operations (crossover and mutation) are implemented on the individuals to produce the new offspring. After each 10 generations, a special generation is used to produce the new individuals by using the pheromone information instead of mutation. The higher the concentration of pheromones on a branch, the more likely it is for that branch to appear in the new GNP individual (see Figure 5.2).

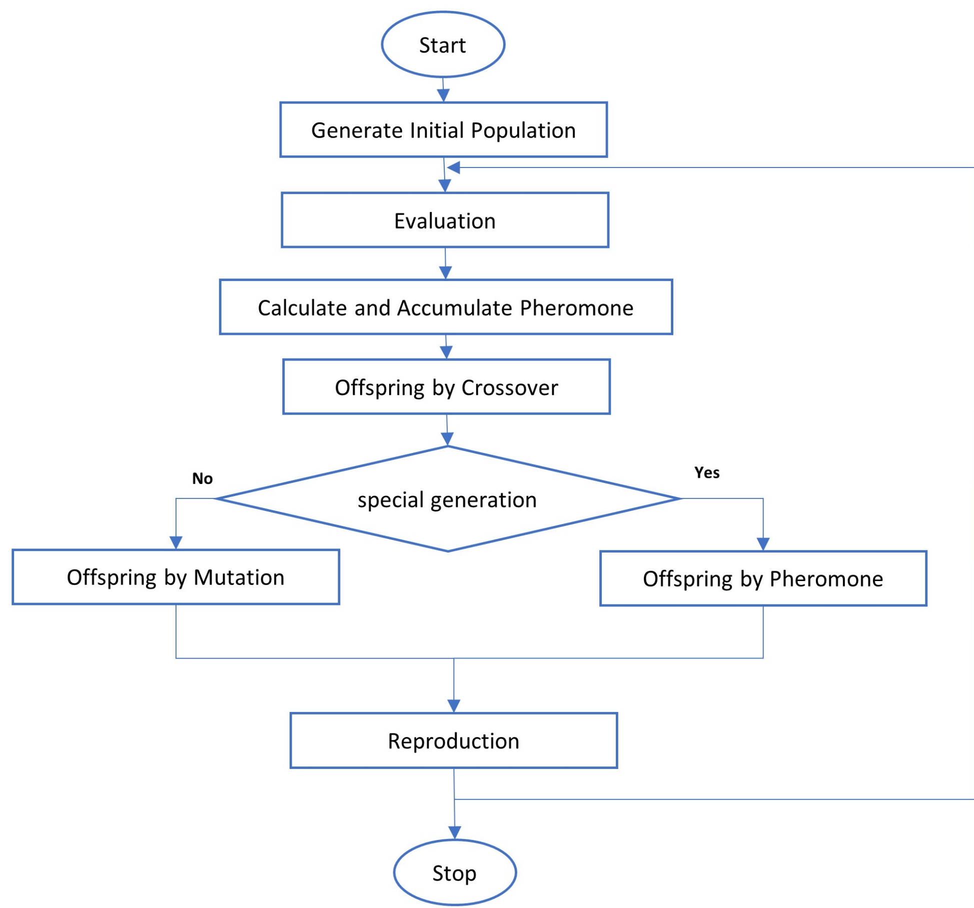


Figure 5.2: Flow chart GNP with ACO [95].

The pheromone is calculated as:

$$h_m^n(i,k,a) = \begin{cases} \frac{F^n - f_m^n}{F^n} & , \quad x(i,k) = a \\ 0 & \quad x(i,k) \neq a \end{cases} \tag{5.3}$$

Where $h_m^n(i,k,a)$ is the pheromone of the kth branch that connects node i with node a from individual m in the nth generation. f_m^n is the fitness of individual m from the n_{th} generation. F_n is the worst fitness of the individuals in the nth generation. $x(i,k)$ is the node number that is connected to node i from the kth branch.

The pheromone is updated by calculating the total pheromone of the kth branch of node i which connect to node a in the nth generation as below:

$$H^n(i,k,a) = (1-p)\,H^{n-1}(i,k,a) + \sum_{m \in M} h_m^n(i,k,a) \tag{5.4}$$

Where $H^n(i,k,a)$ is the pheromone of the kth branch that connects node i with node a in the nth generation. p is the evaporation parameter. M is the set of the suffix's individuals.

The probability of connection is calculated as:

$$P^n(i,k,a) = \frac{H^n(i,k,a)}{\sum_{a \epsilon A(i.k)} H^n(i,k,a)} \tag{5.5}$$

Where $P^n(i,k,a)$ is the probability of connection the kth branch from node i to node a in the nth generation. $A(i,k)$ is the set of the node numbers that connected from the kth branch of node i. In the special generation the offspring is generated by $P^n(i,k,a)$.

In [96], the use of ACO with GNP has been improved when the algorithm updates pheromone information dynamically based on both fitness and transition frequency. When all the individuals evaluated the fitness function and the frequency of transition are calculated, the fitness value and the frequency of transition are used to calculate the pheromone intensity as follows:

$$h_m^n(i,k,a) = \frac{F^n - f_m^n}{F^n} \,.\, \gamma_m^n(i,k,a) \tag{5.6}$$

Where $h_m^n(i,k,a)$ is the pheromone of the kth branch that connects node i with node a from individual m in the nth generation. f_m^n is the fitness of individual m from the nth generation. F_n is the worst fitness of the individuals in the nth generation. $\gamma_m^n(i,k,a)$ is the frequency of the transition in the kth branch that connect node i with node a in individual m at n^{th} generation. This algorithm has been applied to Elevator Group Supervisory Control Systems (EGSCS) and it verified its efficiency.

In 2019 [42], a new combination of GNP with ACO was implemented by Roshanzamir et al. called Graph Structure Optimization of Genetic Network Programming with Ant Colony Mechanism ACNP. This algorithm works to extract useful information from the individuals in a generation and uses it to generate a new generation. Each individual in the population represents a scenario that guides agents' behaviors in the environment. ACNP reinforces certain parts in individuals with high fitness to produce the next generation. This ant colony-inspired mechanism is adopted instead of the crossover and mutation to generate new individuals for the next generation. This algorithm was applied to the Tile World problem and the Prey and Predator problem when the environment is deterministic or stochastic. The algorithm proved its effectiveness, particularly in stochastic environments, when comparing its results to the standard GNP and some of its extensions.

To the best of our knowledge, using the genetic algorithm to improve the optimal search has been done a few times; in 2014, Zhou et al. used a genetic algorithm to calculate minimal hitting sets by encoding the vectors, and then feeding them to the genetic algorithm to find the best minimal hitting set [97]. In 2018, Yiu et al. used a Genetic Algorithm to minimize the effort on heuristic function design [98]. While using the optimal search as a component inside GNP has been done in [17] [14].

In this chapter we add a systematic strategy into this stochastic approach, to make the algorithm faster and better at finding the best solution. Using elements of Conflict-Directed A* with GNP is designed to improve the evolutionary mechanisms. We combine the optimized approach of Conflict-Directed A* with the evolutionary techniques, taking into consideration the conflicts that are extracted from the graph to speed up the process.

5.3 Proposed Algorithms (Architecture)

To incorporate optimality into the GNP, we started by adding A* to the GNP to explore all the possible functions for the (processing and judgment nodes). We found that while the algorithm explored all the possible functions for the nodes that were increasing the performance for the first generations but then it enters in a diversity loss early. That is because the algorithm focuses on exploring all the possible functions and forgets that the individual could also be improved during the evolutionary operations. So, we decided to make the optimal search explore only the processing nodes, as they have the most effect on the results, and let the judgment nodes change through the evolutionary operation (we called it Decision-Based A*-GNP). This algorithm speeds up the improvement process in the individual, but it also loses diversity before it reaches an optimal solution. In the end, we incorporated the CDA* into the GNP, as this algorithm explores all the possible functions for the processing nodes, taking into consideration

excluding the functions that cause a conflict in the graph, which makes this incorporate the fastest one in improving the results for the individuals and get the best solution. In this section, we will explain the three methods and test them on the Tile World problem.

5.3.1 A*-GNP

Suppose we need to find the best graph using A*; if the graph has 120 nodes and each node has a probability of being one of 12 different node types, each node has several connections, each connection could be one of 120 possibilities, the number of explorations for the A* to find the solution is huge, especially when the node can be used more than once. In this approach, the A* will be used in some parts of the GNP graph, and the GNP will be used on the other parts. We decided to make the A* work to find the best function for the nodes (processing and judgment nodes) and the GNP work to change the connections randomly. The algorithm starts as the normal GNP starts with randomly initializing the first population and then evaluating each individual using the fitness function. After that, the algorithm applies evolutionary operations to create new individuals to pass them to the next generation. The A* algorithm will be applied only on the best chromosome in each generation to try to find the best combination of node functions for this chromosome. It will apply for a number of trials, then it will stop and pass the individual to the next generation to make the GNP randomly change the graph component and then try to increase its fitness value in the next generation. The way in how the algorithm has been applied to the Tile World Problem is explained below, (see Pseudocode 8 and Figure 5.3):

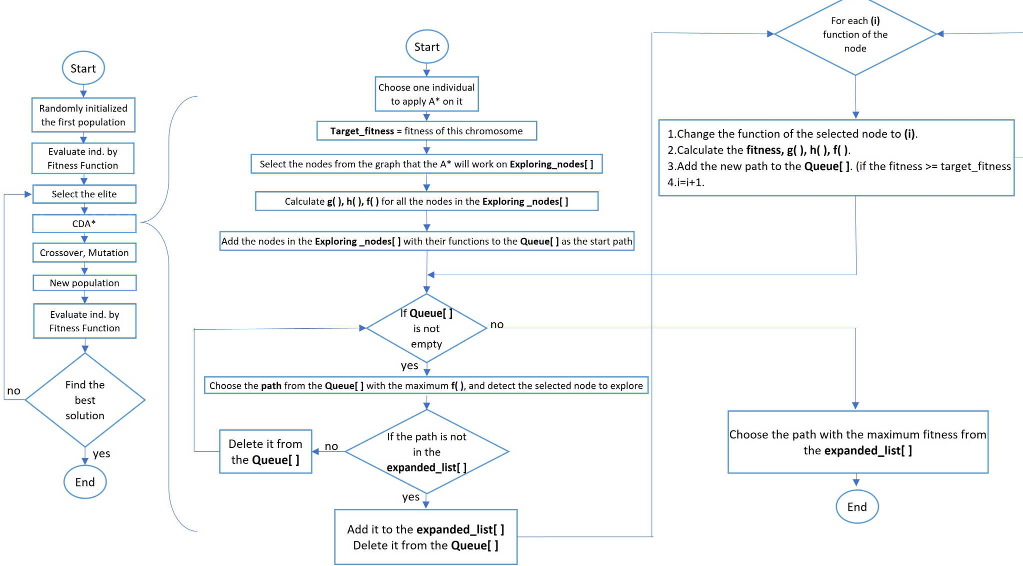


Figure 5.3: Flow chart for A*-GNP.

Algorithm 8 A* with Genetic Networking Programming (A*-GNP)

1: Choose the best individual.
2: **Target-fitness** ← fitness of this chromosome
3: **Exploring-nodes**[ ]← All the used nodes in the Graph (processing and Judgments)
4: Calculate **g( ), h( ), f( )** for all the nodes in the **Exploring-nodes**[ ]
5: **Start-path** ← nodes in the **Exploring-nodes**[ ] with their functions and **g,h,f** value.
6: Add **start-path** to the **Queue**[ ]
7: **while** (**Queue**[ ] is not empty) **do**
8: Choose the path from the **Queue**[ ] with the maximum **f( )**.
9: **if** the path is not in the **Expanded-list**[ ] **then**
10: Add the path to the **Expanded-list**[ ]
11: Delete the path from the **Queue**[ ]
12: **for** each (i) function of the nodes **do**
13: Change the function of the selected node to (i).
14: Calculate the fitness, g( ), h( ), f( ).
15: Add it to **Queue**[ ] if the $fitness >= target_fitness$.
16: i=i+1.
17: **end for**
18: **else**
19: Delete the path from the **Queue**[ ]
20: **end if**
21: **end while**
22: Choose the path with the maximum fitness from the **Expanded-list**[ ]

Solving the Tile World Problem

For the Tile World problem, we used the same parameters that have been used by Mabu et al. in [6] as in Table 5.2.

Table 5.1: Node structure.

Property	Description	
1. Node ID	Unique node ID	
2. Node Type	One of the three possible node types:	
	Start Node (SN)	
	Judgement Node (JN)	
	Processing Node (PN)	
3. Number of sub-nodes	In the GNP, the node is not divided into sub-nodes. So, there are no sub-nodes in the GNP graph's nodes.	
4. Function ID	each node has a function definition that is indexed in the library of functions using its Function ID.	
5. Set of connections	each node has 1 or more connections that directs the agent to the next node.	
6. Delay time	Used to represent the number of steps taken by the agent after executing a node. An agent is stopped once the available number of steps for the agent is already used. Each agent is allowed to take a maximum of predefined steps to solve the problem. In the literature, the number of steps is also referred to as the delay time, as each step incurs some delay. Moreover, each step is counted as being equivalent to a number of micro steps. Depending on the node type executed, the number of micro steps taken varies.	
	Unit	**Description**
	Single step	Each step is equivalent to 8 micro steps
	Judgement node execution	Each judgement node execution is equivalent to 1 micro step.
	Processing node execution	Each processing node execution is equivalent to 5 micro steps.

Chromosome Evaluation (Fitness):
To evaluate any chromosome, the algorithm runs on the 10 environments one by one (as in Figures 1.1).

The three agents start working from the first node in the graph and follow the directions (connections) until all the three tiles drop into the three holes or if the available number of steps is finished in which case the algorithm will close the current environment and start the next one. Then the fitness value is calculated by equation 2.4 [14].

Where D_{tile} is the number of dropped tiles in holes, T_{remain} is the remaining steps

Table 5.2: Parameters for the Tile World Problem: A*-GNP.

Problem Domain	**Parameters for the Tile World Problem with A*-GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole; J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
A* Parameters			
Number of Exploration	150		
Type of exploring nodes	All visited nodes (Judgment & Processing nodes)		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

for the agents after the simulation finished, The $D_{distance}$ is the total distance that the agents spent in the simulation, and it is calculated as below:

- 4 points → when the agent pushes the nearest tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes the nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.

- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 10 points → when the agent pushes a normal tile to a hole.

A* in GNP:

In this part, the algorithm chooses one individual from the population to apply A* on it. The individual is always the best elite from the population except if the best one has been explored in the previous generation and it has not been changed by the evolutionary operation, so the algorithm will choose the next best one. A* is applied only on one chromosome not all the chromosomes.

a - The Fitness value for the selected individual will be chosen to be a Target-fitness that the algorithm needs exceed.

b - The algorithm chooses a vector of Exploring-nodes from the graph to work on. These nodes are all the (processing & judgment) nodes that have been used by the agent from the graph while the simulation was happening.

c - The algorithm calculates **g( )**, **h( )**, and **f( )** for all the nodes from the Exploring-nodes: a. g( ) = Fitness value for the graph b. h( )=(number of Exploring-nodes - Level) x 10 where level is the series number of selected node from the Exploring-nodes it starts with zero and increases by one each time. f( ) = g( ) − h( )

d - The algorithm creates the start path which contains all the Exploring-nodes with their function ID and the values for f( ), g( ) and h( ) and the level.

e - The algorithm adds the start path to the Queue.

f - If **Queue** is not empty:

   a. Choose the **path** from the **Queue** with the maximum **f( )**, based on the level for the selected path the node on the level index will be chosen to explore.

   b. If the selected path and its level value is not in the **expanded-list**:

i. Add it to the **expanded-list**.
ii. Delete it from the **Queue**.
iii. For each (i) function of the nodes (Processing → GF, TL, TR, ST / Judgment → JF, JB, JL, JR, TD, HD, THD, STD):
- Change the function of the selected node to (i).
- Calculate the **fitness, g( ), h( ), f( )**.
- If the fitness >= target-fitness → Add the new path to the **Queue**.

c. Else → Delete it from the Queue[ ].

d. Repeat from step (f).

g - If Queue is empty → Choose the path with the maximum fitness from the **expanded-list** to pass to the next generation as elite.

Calculating the Heuristic in this algorithm

Figure 5.4 shows an example of applying A*-GNP on a graph from a Tile World problem, how the heuristic is calculated, and how to choose the path that will be explored by A*.

Firstly, while evaluating the graph, each agent records all the visited nodes. It records the node-id, node-type, and node-function for all the used nodes. The A* algorithm chooses the first path, which is all the visited nodes from all the agents, considering not duplicating the nodes as each node could be visited many times by the agents. After that, the algorithm will add the first path to the Queue with its **g( )**, **h( )**, and **f( )**. As explained before that **g( )** is the fitness value for the graph, **h( )** is calculated by the following equation:

$$h\,(\,) = (number\ of\ ExploringNodes - Level) \times 10 \tag{5.7}$$

and f( ) is g( ) – h( ).

The reason for choosing this formula to calculate the heuristic is to make sure that the algorithm will always choose the path from the Queue with the maximum f value, which ensures that it is the best one so far. The heuristic here is based on two factors: the number of the exploring-nodes and the path level. When the algorithm first starts, the level is zero, which means no nodes have been explored by the A* yet, which will give the h( ) the maximum value with the number of exploring-nodes. Whereas, when the level is high, the number of the explored nodes is more, which will make the h( ) have a lower value. The relationship between the h( ) and f( ) is an inverse relationship that to maximize the f value, the h value should be the lowest, and the lowest h value means a path with a high level which has a high number of explored nodes in it. We

used ten as a weight for the h value to balance its weight with the fitness value. So, if we have many paths with the same fitness, the path with the maximum level has the priority to be explored by the A* first, as it has already explored many nodes.

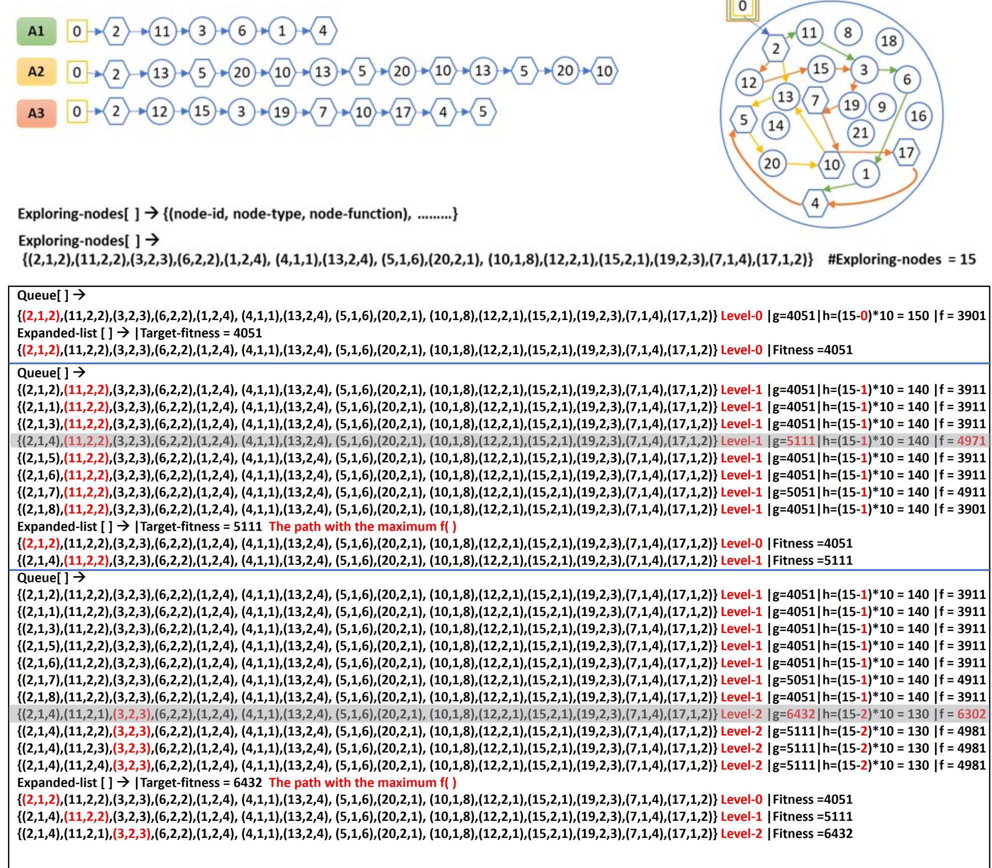


Figure 5.4: An example of applying A*-GNP on a graph (Tile World problem).

In the example (Figure 5.4), the number of the exploring-nodes is 15. The algorithm starts with level 0 and chooses a node (2,1,2) that has id = 2, and it is a judgment node (node-type = 1) with a function (2) which is (JB → What is in the Back position). The fitness value of the start path is 4051 which will be the target-fitness. In this case, g = 4051, h = (15-0)*10 = 150, f = (4051 – 150) = 3901. The algorithm should choose the path with the maximum f value, in this case, there is just one path that is chosen by the algorithm to be explored. When the path is explored, it will be deleted from the Queue and added to the expanded-list, and another (8) new paths will be added to the queue. These paths contain the different function types of the judgment node (as the judgment nodes have different types of functions from 1 – 8). The new 8 paths will be added to the Queue with a level equal to 1 as it increases by one, the g, h, and f values will be calculated for each one of them. The h in this case = (15-1)*10 = 140. After calculating f value for all the paths, the algorithm will choose the path with the

maximum f which is in this case (f = 4971), this path will be deleted from the Queue and added to the expanded-list, and another 4 new paths will be added to the Queue (the new 4 paths have different 4 functions for the node in level 1 which is a processing node). Considering that there are no paths that will be added to the Queue except if its fitness is equal to or greater than the target-fitness, and each time the algorithm finds a path with greater fitness, it will update the target-fitness.

The algorithm will continue till the Queue is empty, or the number of explored paths is equal to 150 (we chose this number because it is half the needed number of evaluated fitness for each generation).

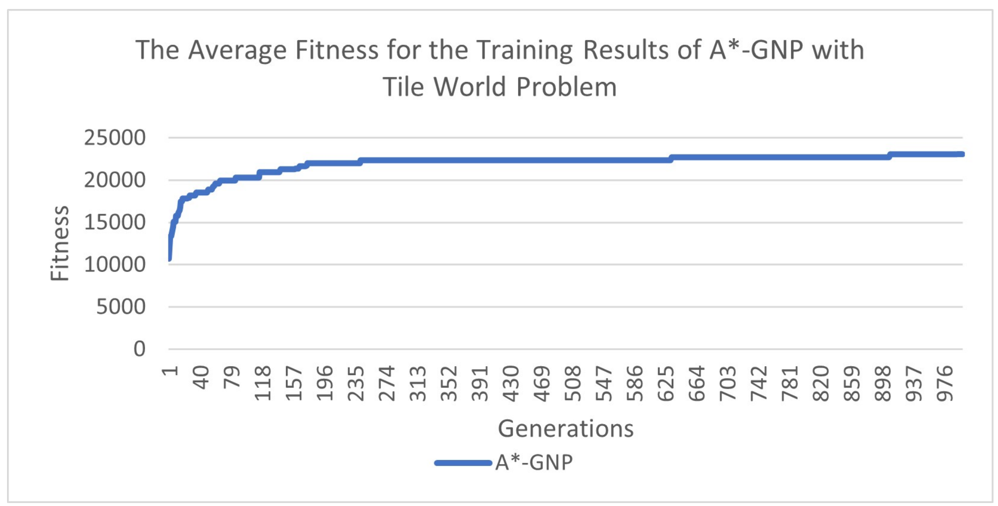


Figure 5.5: Average fitness during training on the Tile World Problem: A*-GNP.

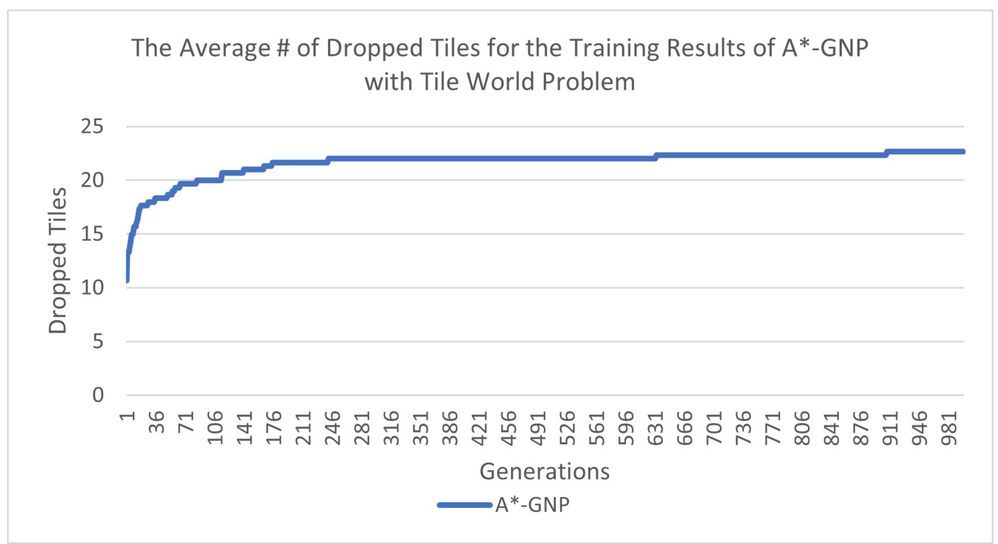


Figure 5.6: Average number of dropped tiles during training on the Tile World Problem: A*-GNP.

To examine this algorithm, we applied the A*-GNP to the Normal Tile World Problem. Figures 5.5 and 5.6 show the average fitness and number of dropped tiles for three experiments when training the algorithm for 1000 generations. We found that the algorithm's performance significantly increased in the early generations, as it could reach 20/30 dropped tiles within the first 100 generations, but then it entered a phase with diversity loss, causing its performance to plateau and wouldn't reach the top result within 1000 generations.

So, to increase the algorithm performance, we decided to use Decision-Based A* with GNP, as in the next section.

5.3.2 Decision-Based A*-GNP (DBA*-GNP)

In this algorithm, we incorporate Decision-Based A* with GNP (DBA*-GNP). The difference between using A* and DBA* with GNP is that A* explores all the nodes (processing and judgment) nodes, while DBA* explores only the decision nodes which are the Processing nodes here, as they are the nodes that make the most impact on the results. We aim from this algorithm to solve the diversity loss problem and make the algorithm quickly increase before it enters the diversity loss issue. Pseudocode 9 shows the algorithm instructions for the DBA*-GNP. It uses all the techniques used by the A*-GNP except the Exploring-nodes [ ] that have been changed to Decision-nodes [ ], which include only the visited processing nodes by the agents. Moreover, the different functions that the algorithm will explore for each processing node are from 1 to 4 (GF, TR, TL, ST).

We have applied the DBA*-GNP algorithm to the Tile World Problem to test this algorithm and check its results. Figures 5.7 and 5.8 show the average fitness and number of dropped tiles for three experiments for 1000 generations. The results of this algorithm also significantly increased within the first generations as it could reach 23/30 dropped tiles within the first 100 generations but then slowly increased and could not reach the top training results. So, it also suffers from the diversity loss problem.

In the following section, we combine Conflict-Directed A* with GNP by avoiding paths that contain conflicts during exploration. This approach aims to improve the algorithm's performance and prevent it from experiencing diversity loss.

Algorithm 9 Decision-Based A* with Genetic Networking Programming (DBA*-GNP)

1: Choose the best individual.
2: **Target-fitness** ← fitness of this chromosome
3: **Decision-nodes**[ ]← All the used Processing nodes in the Graph (Processing only)
4: Calculate **g( ), h( ), f( )** for all the nodes in the **Decision-nodes**[ ]
5: **Start-path** ← nodes in the **Decision-nodes**[ ] with their functions and **g,h,f** value.
6: Add **start-path** to the **Queue**[ ]
7: **while** (**Queue**[ ] is not empty) **do**
8:     Choose the path from the **Queue**[ ] with the maximum **f( )**.
9:     **if** the path is not in the **Expanded-list**[ ] **then**
10:         Add the path to the **Expanded-list**[ ]
11:         Delete the path from the **Queue**[ ]
12:         **for** each (i) function of the processing nodes (1, 2, 3, 4 **do**
13:             Change the function of the selected node to (i).
14:             Calculate the fitness, g( ), h( ), f( ).
15:             Add it to **Queue**[ ] if the $fitness >= target_fitness$.
16:             i=i+1.
17:         **end for**
18:     **else**
19:         Delete the path from the **Queue**[ ]
20:     **end if**
21: **end while**
22: Choose the path with the maximum fitness from the **Expanded-list**[ ]

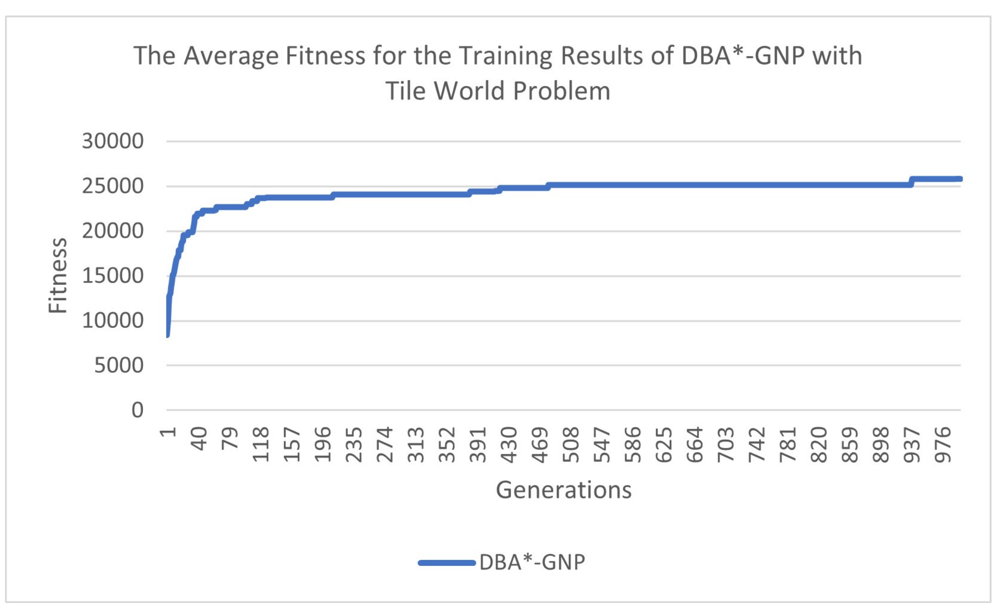


Figure 5.7: Average fitness during training on Tile World: DBA*-GNP.

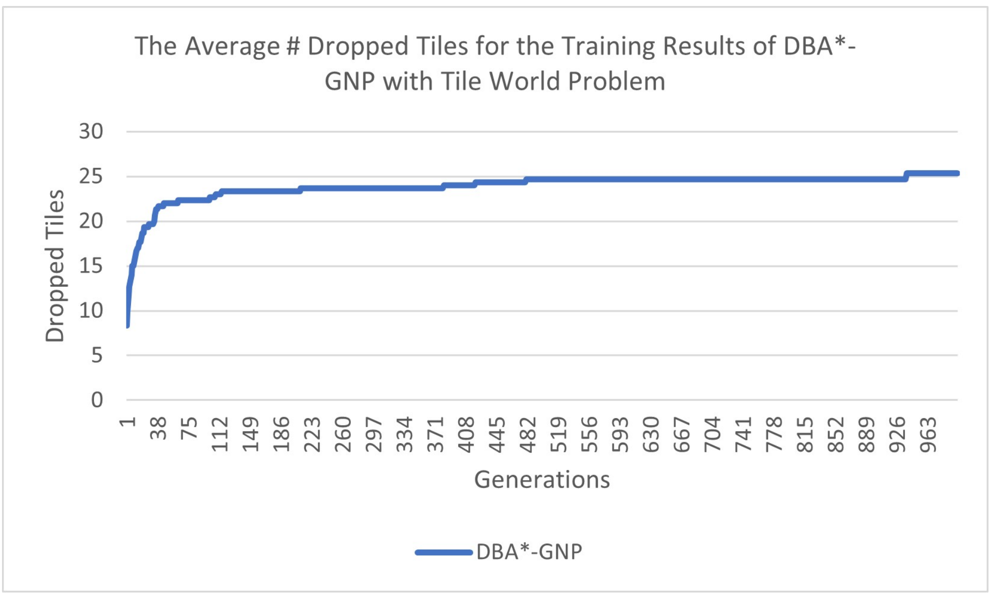


Figure 5.8: Average number of dropped tiles during training on Tile World: DBA*-GNP.

5.3.3 Conflict-Directed A*-GNP (CDA*-GNP)

Conflict-Directed A* operates on a tree structure for the explored rules that has a root node, and the leaves are the actions, whereas GNP has a networking graph structure that has one start node without an end node. Moreover, each node could be visited more than once within the same experiment. Hence, our approach is to make GNP generate the graphs (chromosomes) randomly and then Conflict-Directed A* is applied to the graph to extract the conflicts from it and save them on a conflict list. At the same time, GNP could use these conflicts to improve evolutionary operations. Using elements of Conflict-Directed A* with GNP will help in terms of validating the generated structures from GNP instead of exploring all the possible paths (see Pseudocode 10 and Figure 5.9).

The algorithm combines two techniques: the Conflict-directed A* (CDA*) techniques with the Genetic Networking Programming (GNP) techniques. It combines the optimal search from CDA* with the randomization from GNP. The algorithm begins by randomly initializing the first population and evaluating each individual using the fitness function, just like the normal GNP. Afterwards, the algorithm employs evolutionary operations to generate new individuals for the next generation. The CDA* algorithm will be applied to only the best chromosome in each generation to find the best combination of node functions. It will apply for several trials, and then it will stop and transfer the individual to the next generation. The next generation will randomly modify the graph component of the individual and attempt to increase its fitness value in the following generation. The following is an explanation of how the algorithm was applied to the Tile World Problem.

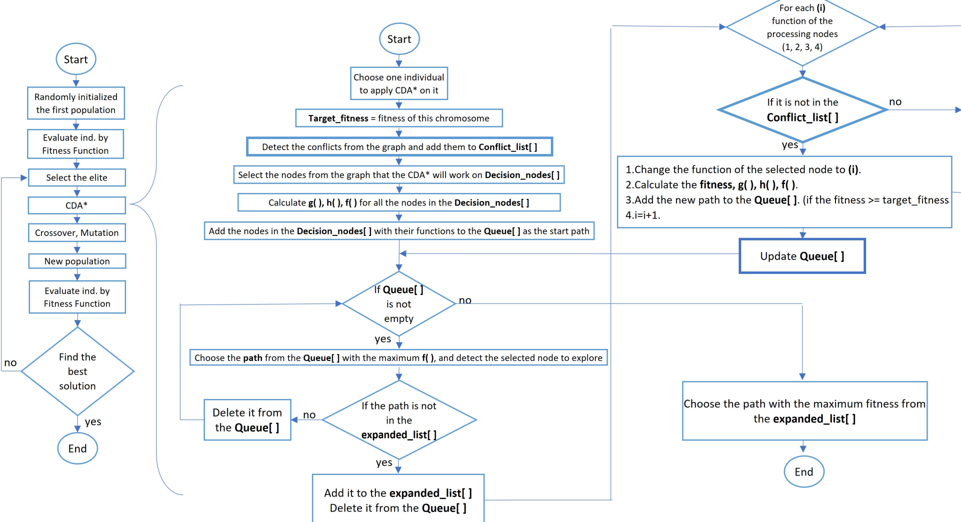


Figure 5.9: Flow chart for CDA*-GNP.

Algorithm 10 Conflict Directed A* (CDA*) with GNP

1: Choose the best individual.
2: **Target-fitness** ← fitness of this chromosome
3: **Conflict-list**[ ]← conflicts from the Graph
4: **Decision-nodes**[ ]← All the used processing nodes in the Graph
5: Calculate **g( ), h( ), f( )** for all the nodes in the **Decision-nodes**[ ]
6: **Start-path** ← nodes in the **Decision-nodes**[ ] with their functions and **g,h,f** value.
7: Add **start-path** to the **Queue**[ ]
8: **while** (**Queue**[ ] is not empty) **do**
9:     Choose the path from the **Queue**[ ] with the maximum **f( )**.
10:     **if** the path is not in the **Expanded-list**[ ] **then**
11:         Add the path to the **expanded-list**[ ]
12:         Delete the path from the **Queue**[ ]
13:         **for** each (i) function of the processing nodes (1, 2, 3, 4) **do**
14:             **if** it is not in the **Conflict-list**[ ] **then**
15:                 Change the function of the selected node to (i).
16:                 Calculate the fitness, g( ), h( ), f( ).
17:                 Add it to **Queue**[ ] if the $fitness >= target_fitness$.
18:                 i=i+1.
19:                 Update **Queue**[ ]
20:             **end if**
21:         **end for**
22:     **else**
23:         Delete the path from the **Queue**[ ]
24:     **end if**
25: **end while**
26: Choose the path with the maximum fitness from the **Expanded-list**[ ]

Tile World problem

For the Tile World problem, we used the same parameters that have been used by Mabu et al. in [6] (see Table 5.3).

Table 5.3: Parameters for the Tile World Problem: CDA*-GNP.

Problem Domain	**Parameters for the Tile World Problem with CDA*-GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - Hole - Floor - Obstacle
	J2	Judge what is in back position (JB)	
	J3	Judge what is in right position (JR)	
	J4	Judge what is in left position (JL)	
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left - Right - None
	J6	Direction to the nearest Hole (HD)?	
	J7	Direction to the second nearest Tile (THD)?	
	J8	Direction from the nearest Tile to the nearest Hole (STD)?	
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole; J5, J6, J7, J8 → Forward, Backward, Left, Right, Nothing		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
CDA* Parameters			
Number of Exploration	150		
Type of exploring nodes	Processing nodes		
Type of extracting conflicts	Loop - Repetition in turns - Repetition in Go Forward - Repetition in Judgment nodes		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.1, Pm1: 0.01, P_{ind}: 0.5, P_{cn}: 0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness):

To evaluate any chromosome, the algorithm runs on the 10 environments one by one (as in Figure 1.1. The three agents start working from the first node in the graph and follow the directions (connections) until all the three tiles drop into the three holes or if the available number of steps is finished in which case the algorithm will close the current environment and start the next one. Then the fitness value is calculated by equation 2.4,

Where D_{tile} is the number of dropped tiles in holes, T_{remain} is the remaining steps for the agents after the simulation finished, $D_{distance}$ is the total distance that the agents

spend in the simulation, and it is calculated as below:

- 4 points → when the agent pushes the nearest tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.
- 2 points → when the agent pushes the nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.
- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.
- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.
- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.
- 10 points → when the agent pushes a normal tile to a hole.

Conflict-Directed A*:
In this part, the algorithm chooses one individual from the population to apply CDA* on it. The individual is always the best elite from the population except if the best one has been explored in the previous generation and it has not been changed by the evolutionary operation, so the algorithm will choose the next best one. CDA* is applied only on one chromosome, not all the chromosomes.

a - The Fitness value for the selected individual will be chosen to be a `Target_fitness` that the algorithm needs to exceed.

b - The algorithm identifies any potential conflicts that may be present on the selected graph. To detect the conflicts, the algorithm will record the nodes in the order that each agent visited while the simulation was happening. Then it extracts the conflict nodes from these records. The conflicts can be one of these situations:

- When there are three or more (Turn Left / Turn Right) nodes after each other without any other nodes between them.
- When there is a loop of nodes (from 2 to 4 nodes) that repeats after each other and it does not contain a processing node.
- When there are two or more (Go Forward) nodes after each other without any other nodes between them.
- When there is a repetition of the Judgment nodes in the records without any processing nodes between them.

c - The algorithm chooses a vector of Decision-nodes from the graph to work on. These nodes are all the processing nodes that have been chosen by the agent from the graph while the simulation was happening.

d - The algorithm calculates **g( ), h( ), f( )** for all the nodes from the Decision-nodes:

i. g( ) = Fitness value for the graph

ii. h( ) = (number of `Decision_nodes` - Level) x 10 where the level is a series number of selected node from the Decision-nodes it starts with one and increases by one each time.

iii. f( ) = g( ) − h( )

e - The algorithm creates the start path which contains of all the Decision-nodes with their function ID and the values for f( ), g( ) and h( ) and the level as in Figure 5.10.

f - The algorithm adds the start path to the **Queue**.

g - If **Queue** is not empty:

(a) Choose the **path** from the **Queue** with the maximum **f( )**, based on the level for the selected path the node on the level index will be chosen to explore.

(b) If the selected path and its level value are not in the **expanded-list**:

i. Add it to the **expanded-list**.

ii. Delete it from the **Queue**.

iii. For each (i) function of the processing nodes (GF, TL, TR, ST):
If (i) is not in the conflict nodes:

- Change the function of the selected node to (i).
- Calculate the **fitness, g( ), h( ), f( )**.
- If the fitness >= target-fitness → Add the new path to the **Queue**.

iv. Update **Queue** by deleting all the paths that contain more than four conflicts from the conflict list and its fitness less than the target-fitness.

v. Repeat from step (g).

(c) Else → Delete it from the Queue[ ].

(d) Update **Queue** by deleting all the paths that contain more than four conflicts from the conflict list and its fitness less than the target-fitness.

(e) Repeat from step (g).

h - If **Queue** is empty → Choose the path with the maximum fitness from the **expanded-list** to pass to the next generation as elite.

Evolutionary Operators:

Crossover: Tournament selection is used to choose two parents for crossover, by exchanging some nodes from parent1 with parent 2 using crossover probability (Pc = 0.1).

Mutation: Tournament selection is used to choose one parent, this time to change the function of a selected node using mutation probability (Pm = 0.01) and to change the connection of a selected node using mutation probability (Pm = 0.1). In CDA*-GNP the Mutation was also done on a probability ($P_{ind} = 0.5$) of the individuals to change the node function or connection of a chosen node from the extracted conflict node-list with a probability ($P_{cn} = 0.1$).

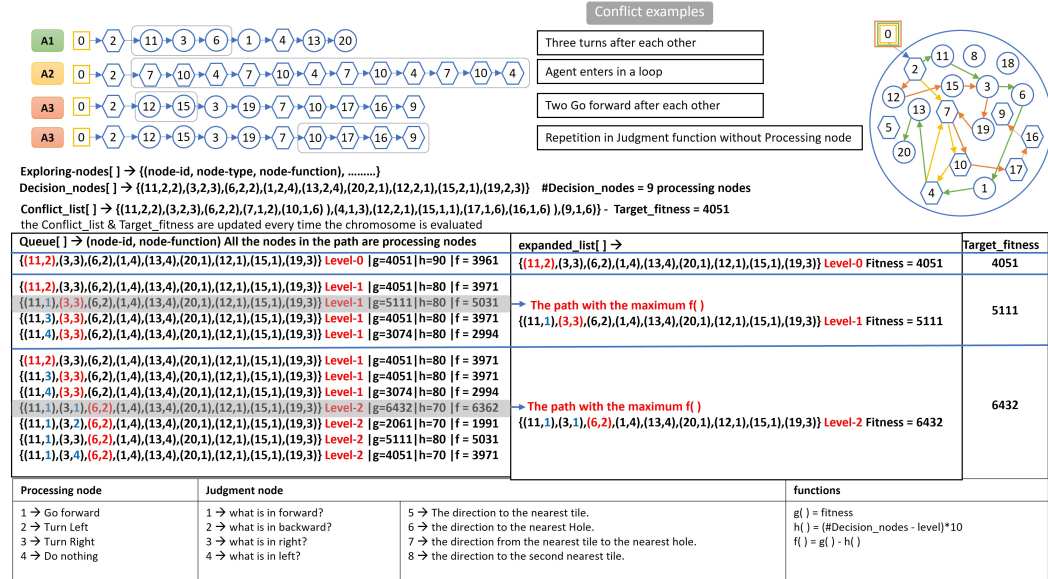


Figure 5.10: Example: Solving the Tile World problem with CDA*-GNP.

Example on CDA*-GNP

In Figure 5.10 there is an example of the CDA*-GNP algorithm. Suppose that we have three agents that use the same chromosome, they start from the start node number 0 that directs each of them to node 2. The algorithm will record the used nodes by each agent as shown in the figure:

A1: 0 → 2 → 11 → 3 → 6 → 4 →13→20→...

A2: 0 → 2 → 7 → 10 → 4 →7 → 10 → 4→ 7 → 10 → 4→ 7 → 10 → 4 →...

A3: 0 → 2 → 12 → 15 → 3 → 19 → 7 → 10 → 17 → 16 → 19 → ...

After recording the paths for each agent, it will extract the conflicts from each agent. In this example there is a conflict on {`(11), (3), (6)`} for Agent 1 as they are three turns after each other without any go-forward node between them, so this series of nodes will be added to the Conflict-list[ ]. The same method will applied to all agents' records and all the conflicts will be added to the same Conflict-list[ ]. Noting that, the Conflict-list[ ] contains all the conflict nodes (processing and judgment) nodes.

The Decision-nodes[ ] list will contain all the processing nodes that all the agents have visited as in the example. The number of the nodes in the Decision-nodes[ ] will be the maximum number of levels the algorithm will explore which is in this example = 9. The algorithm will use the fitness value for the graph to be the initial value for the Target-fitness which is in this example = 4051.

First Step:

The Decision-nodes[ ] will be chosen to be the first path to be added to the Queue[ ], the level for the first path is 0, and the g( ), h( ), and f( ) are calculated as below: g( ) = fitness – 4051. h( ) = (number of Decision-nodes[ ] – level) * 10 = (9-0)*10 = 90. f( ) = g( ) – h( ) = 4051 – 90 = 3961. So, the Queue now has the first path with its level, g, h, f, and fitness values.

Second Step:

The algorithm will choose the path with the maximum f value and add it to the expanded-list[ ]. Then, because the level of this path is 0 so the algorithm will explore the node number 0 from the path which is in this example (11,2) → (node id, node function). So, the node with id = 11 and function 2 which is (Turn Left) will be the chosen node.

The algorithm will delete the chosen path from the Queue and add 4 new paths instead. Each one of these paths has a different function for the selected node (node 0) as in the figure. The fitness, g, h, f, and level values will be calculated for all the paths before adding them. The level for the paths is 1 as the level is always calculated → new level = level +1.

Third Step:

Now we have four paths in the Queue[ ], the path with the maximum f value will be the next chosen path, which is here in the example the path with the f = 5031. This path will be deleted from the Queue and added to the expanded-list[ ]. And the Target-fitness will be updated to be the new highest fitness value = 5111.

Then four new paths will be added to the Queue[ ] after calculating fitness, g, h, f, and level. Each one of them has a different function value for node number 1 on the path as the level here is 1.

The algorithm will continue the steps till the Queue is empty or till a specific number of explorations. At the end, the path with the maximum fitness values from expanded_list[ ] will be chosen as the best combination of node functions for the selected graph. To reduce the number of evaluations we added three techniques to the algorithm:

1. The algorithm will continue the steps till the Queue is empty or till the algorithm explores 150 paths from the Queue. We chose 150 because it's half the number of evaluations per generation, and we want to stop the algorithm from exploring the graph as the graph could be improved too through the evolution operation via generation.

2. When exploring the nodes only the paths with the fitness value equal to or more than the Target-fitness will be added to the Queue[ ].

3. Each time the algorithm starts it will randomly rearrange the nodes in the Decision-nodes[ ] to increase the chance to explore more different nodes and to avoid losing the diversity.

Additional detailed explanations and justifications for the decisions made in designing the algorithm

1. We chose to apply CDA* only to the best chromosome because, typically, after a few generations, the elite chromosomes are similar to each other. Applying CDA* to more than one individual would increase the similarity between them, leading to a loss of diversity.

2. The conflicts identified from the graph are selected by observing the behaviour of the agents in previous algorithms' individuals. This is done by recording videos and determining the existence of these conflicts.

3. We decided to apply the CDA* only on the processing nodes because they produce the most effective changes on the graph.

4. One of the challenges we faced in the proposed algorithm was controlling the size of the Queue. Therefore, we implemented various techniques to overcome this issue:

   (a) We created a variable called "Target-Fitness" and initially assigned it the fitness value for the first path. Then, each time the algorithm finds a path with a fitness value greater than the "Target-Fitness," it will update the "Target-Fitness" to the highest fitness value found. Therefore, before adding any path to the queue, we ensure that the fitness value of the new path is equal to or greater than the "Target-Fitness.

   (b) After each evaluation for the individual, the Queue will be updated. This update will make sure to delete all the paths that have a fitness value that is less than the Target-fitness and has more than four conflicts in its path.

   (c) We decided to stop the CDA* from completing the exploration after a number of evaluations, which was in the Tile World Problem (150) half the total number of evaluations per generation.

5. We designed the proposed heuristic function because it estimates the number of nodes the algorithm explored along the path. We utilized a weight of 10 to balance its outcomes with the fitness value.

6. We rearrange the order of the decision nodes before each time the CDA* starts exploring the individuals to increase the chance of exploring all the decision nodes.

7. In the mutation process, we utilize the conflict list by selecting a node from it and applying the mutation to increase the chances of resolving the conflict.

How to prevent losing diversity in the CDA*-GNP?

1. We keep the mutation rate at (0.01) when changing the node function as the CDA* will apply only to one individual, so we make sure to change the node function for the rest of the individuals to increase the diversity. Choosing (0.01) for the mutation rate on the node function will ensure the minimum changes in the individuals because in most cases, the individual needs to change just one function to increase its performance. Making many changes could cause individuals to lose their stability in improvement.

2. Since the CDA* ensures choosing the best function for the decision nodes, we increase the mutation rate when changing the connections between the nodes to (0.1). Because not all the nodes are visited by the agents when running the simulation, so changing the connection on the unused node, will not make any improvement. Because of that, we increase the mutation rate for the connection.

3. CDA* is applied to only one individual from each generation except the initial population (first population), the CDA* is applied to the best five individuals from this population to increase the diversity in the upcoming generations.

4. Each time the CDA* start working on an individual, it will make a random change in the order of the decision nodes that have been visited by the agents. So, the CDA* will start exploring a different node each time it works.

5. CDA* has a methodology for choosing the individual that will work on it. It will not always choose the best individual, but it will check two points: if the best individual has been explored by CDA* in the previous generation and it has not been changed by the crossover and mutation, CDA* will go to the next best individual and check the two-points again till find the suited individual.

Why CDA* has not been applied on GNP-RL instead of GNP?

In GNP-RL each node has more than one sub-node and each sub-node has a different function. Within a simulation, each time an agent visits a node it will choose a different sub-node based on the exploration and exploitation and because the node could be visited many times by an agent or by another agent that will lead to a non-stable function for the node. So, if we add a CDA* to GNP-RL, the CDA* needs to explore and test all the different function types for the explored node, if the GNP-RL chooses a different function type each time it visits a node that will confuse the CDA* and will

not make it able to find the best function type. Table 5.1 shows the differences between the CDA*-GNP and GNP-RL.

The differences between CDA*-GNP and GNP-RL

Table 5.4 shows the differences between CDA*-GNP and GNP-RL:

Table 5.4: The differences between CDA*-GNP and GNP-RL.

<table>
<tr><th>CDA*-GNP</th><th>GNP-RL</th></tr>
<tr><td>In this algorithm the A* will apply only on the decision nodes (Processing nodes), while the Judgment nodes will be sited and changed through the evolutionary operation on the GNP.</td><td>The Reinforcement Learning is working on all the nodes (judgment and Processing)</td></tr>
<tr><td>It is applied just on the best chromosome (individual)</td><td>It is applied on all the chromosomes.</td></tr>
<tr><td>The aim is to find the best function for the processing nodes in a graph that suited with the established judgment nodes and connections that has been chosen for this chromosome through the GNP process.</td><td>The aim is to explore more different functions on the same graph without increase the graph size</td></tr>
<tr><td>Within the same simulation on the same chromosome all the agents will choose the same function for the nodes every time they visit it.</td><td>Within the same simulation on the same chromosome every time any agent visited a node it could use different function type. So, within the same evaluation the node could be used with different type function and that makes the testing result for the chromosome is different than the training result on the same training environment, because of the randomization on the exploration in the Reinforcement Learning.</td></tr>
<tr><td>All chromosomes will be simulated once in each generation except the best one will be simulated many more times (it could be till 150 more evaluation) because of the running of the CDA* on it so there are more than 300. It could be from 300 to 450 evaluation for each generation</td><td>All chromosomes will be simulated once in each generation so there are 300 evaluations within each generation.</td></tr>
<tr><td colspan="2">The size of the GNP-RL graph is up to 4 times the size of the CDA-GNP graph.</td></tr>
<tr><td colspan="2">Because of that we will use the number of evaluations instead of the number of generations to compare both algorithm</td></tr>
</table>

To test the generalization of this algorithm we implemented this algorithm on another two problems (Heavy Tile World Problem and Prey and Predator Problem). Below is the explanation on how the algorithm is working with these two problems and then the results of all the implementations are explained in section (Testing and Analysis) below.

Heavy Tile World Problem

To increase the difficulty of the Tile World problem, we change one of the tiles in each environment to be a heavy tile that needs two agents to push it at the same time as explained in Chapter 1.

To apply CDA*-GNP to the Heavy Tile World problem we have done some changes to it:

1. We used variable-sized Genetic Networking Programming (VSGNP) on the chromosome structure to divide the graph into two sub-programs, one to solve the normal tiles and the other to solve the heavy tile.

2. The Judgment node number 8 has been changed from (giving the direction to the second nearest tile) to (check the type of the nearest tile is normal or heavy).

Chromosome structure:
Tables 5.5 and 5.6 show the chromosome structure and the algorithm parameters for the CDA*-GNP and CDA*-VSGNP when applying it on Heavy Tile World Problem.

First Population Initialization: For this experiment, there are 300 individuals (chromosomes) with 120 nodes for each; each is randomly initialized as the first population using the same technique as in [9] when using two sub-programs and the parameters from [6] when using one sub-program. When using the two sub-program, the initial population consider the two sub-programs as below: The graph contains two sub-programs:

1- To solve the Normal tiles: 72 nodes → 48 Judgment nodes, and 24 Processing nodes.

2- To solve the Heavy tiles: 48 nodes → 32 Judgment nodes, and 16 Processing nodes.

Only the Judgment node number (8) can have a connection to the other sub-program. All the other nodes should have a connection to the same sub-program, and they can't connect to the other sub-program. The Judgment node number (8) checks for the tile type if it is normal or heavy.

1- If the answer of JN(8) is (Normal) the connection should be to the sub-program1.

2- If the answer of JN(8) is (Heavy) the connection should be to the sub-program2.

Table 5.5: Parameters for the Tile World Problem with Heavy Tile Using one sub-program with CDA*-GNP.

Problem Domain	**Tile World Problem with Heavy Tile Using one sub-program with CDA*-GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile**
	J2	Judge what is in back position (JB)	- Hole
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left
	J6	Direction to the nearest Hole (HD)?	- Right
	J7	Direction to the second nearest Tile (THD)?	- None
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile; J5, J6, J7 → Forward, Backward, Left, Right, Nothing; J8 → Normal, Heavy		MF → With Conflict, Without Conflict; TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
CDA* Parameters			
Number of Exploration	150		
Type of exploring nodes	Processing nodes		
Type of extracting conflicts	Repetition in turns - Repetition in Judgment nodes		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.01, Pm1: 0.01, P_{ind}: 0.5, P_{cn}: 0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

Chromosome Evaluation (Fitness): To evaluate any chromosome, the algorithm runs on the 10 environments one by one (Figure 1.1. The three agents start working from the first node in the graph and follow the directions (connections) until all the three tiles drop into the three holes or if the available number of steps is finished in which case the algorithm will close the current environment and start the next one. Then the fitness value is calculated by equation 2.4.

Where D_{tile} is the number of dropped tiles in holes, T_{remain} is the remaining steps for the agents after the simulation finished, $D_{distance}$ is the total distance that the agents spend in the simulation, and it is calculated as below:

- 8 points → when the agent pushes the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

Table 5.6: Parameters for the Tile World with Heavy Tile with two sub-programs with CDA*-VSGNP.

Problem Domain	**Tile World Problem with Heavy Tile Using two sub-programs with CDA*-VSGNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent - Tile - **Htile**
	J2	Judge what is in back position (JB)	- Hole
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Tile (TD)?	- Forward - Backward - Left
	J6	Direction to the nearest Hole (HD)?	- Right
	J7	Direction to the second nearest Tile (THD)?	- None
	J8	**Type of the nearest Tile (TT)?**	- **Normal** - **Heavy**
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Tile, Hole, HTile J5, J6, J7 → Forward, Backward, Left, Right, Nothing J8 → Normal → sub-program 0; Heavy → sub-program 1 For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.		MF → With Conflict, Without Conflict TL, TR, ST → Next Node
Number of Steps	Each agent is allowed to take a maximum of 100 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.		
CDA* Parameters			
Number of Exploration	150		
Type of exploring nodes	Processing nodes		
Type of extracting conflicts	Repetition in turns - Repetition in Judgment nodes		
Algorithm Parameters			
Number of individuals	300		
Number of Elites	5		
Number of Crossover individuals	120		
Number of Mutation individuals	175		
Mutation rate	Pm: 0.01, Pm1: 0.01, P_{ind}: 0.5, P_{cn}: 0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	7		

- 4 points → when the agent pushes the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 4 points → when the agent pushes the nearest Normal tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

- 4 points → when the agent pushes not the nearest heavy tile and the distance between this tile and the nearest hole after pushing the tile is less than the distance between them before the push.

- 2 points → when the agent pushes not the nearest heavy tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 2 points → when the agent pushes the nearest Normal tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 2 points → when the agent pushes a not nearest tile and the distance between this tile and its nearest hole after pushing the tile is less than the distance between them before the push.

- 1 point → when the agent pushes a not nearest tile and the distance between this tile and any hole after pushing the tile is less than the distance between them before the push.

- 0 point → when the agent pushes any tile and the distance between this tile and any hole after pushing the tile is equal to the distance between them before the push.

- -1 point → when the agent pushes any tile and the distance between this tile and all holes after pushing the tile is more than the distance between them before the push.

- 10 points → when the agent pushes a normal tile to a hole.

- 20 points → when the agent pushes a heavy tile to a hole.

<u>Conflict-Directed A*:</u> The same technique of (CDA*) used with the Normal Tile World problem will be used in the Heavy Tile World Problem, except for the conflicts that the algorithm will exclude. In the Normal Tile World Problem, we have four types of conflicts: loop nodes, repeated Going forward nodes, repeated turn right and left nodes, and repeated judgment nodes. In Heavy Tile World Problem, we use only repeated turn right and left nodes and repeated judgment nodes as a conflict. We exclude the loop and the repeated Going Forward nodes, as to push the heavy tile, the agents need to repeat the Going Forward nodes and make some loop nodes so the two agents can push the same heavy tile together.

Evolutionary Operators:
The same Crossover and Mutation process are used too with the Heavy Tile World Problem considering this point when using two sub-programs :
On the evolutionary operation, the algorithm should make sure that the connection between the sub-programs (the connection for the Judgment node 8) is stable after the changing. So, the normal connection should direct to sub-program 0 and the heavy connection should direct to sub-program 1 as shown in Figure 5.11.

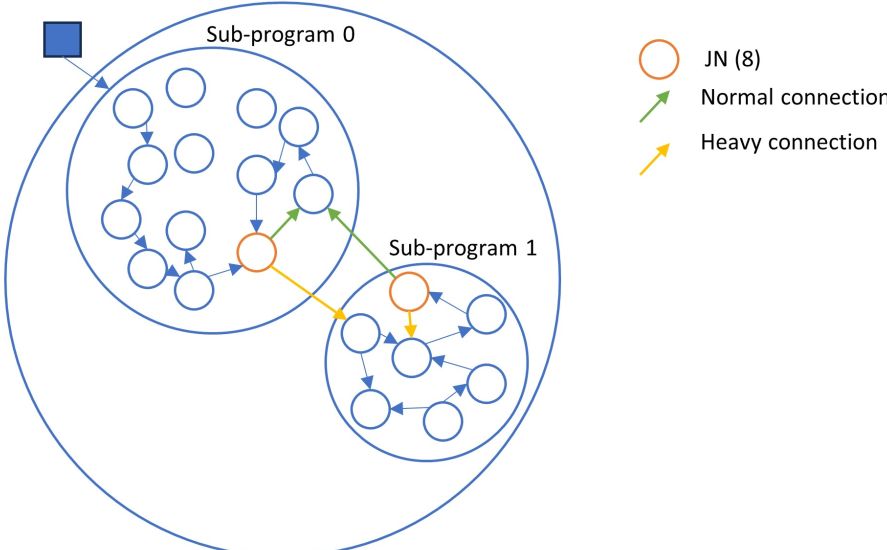


Figure 5.11: Heavy Tile Program Graph Structure.

Prey and Predator Problem

For the Prey and Predator problem, we used the same parameters that have been used by Mabu et al. in [6]. Table 5.7 shows the chromosome structure and algorithm parameters for the CDA*-GNP when applying to Prey and Predator Problem.

First Population Initialization: For this experiment, there are 50 individuals (chromosomes) with 120 nodes for each; each is randomly initialized as the first population using the same technique as in [9].

Chromosome Evaluation (Fitness): To evaluate any chromosome, the algorithm runs on the 30 environments one by one (as in Figure 1.6). The four agents start working from the first node in the graph and following the directions (connections) until the prey is rounded by all four agents or if the available number of steps is finished in which case the algorithm will close the current environment and start the next one. Then

Table 5.7: Parameters for the Prey and Predator with CDA*-GNP.

Problem Domain	**Prey and Predator with CDA*-GNP**		
Judgement Nodes	ID	Query Definition	Possible response to query
Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14].	J1	Judge what is in front position (JF)	- Agent
	J2	Judge what is in back position (JB)	- Prey
	J3	Judge what is in right position (JR)	- Floor
	J4	Judge what is in left position (JL)	- Obstacle
	J5	Direction to the nearest Prey (PD)?	- Forward - Backward - Left - Right - None
Processing Nodes	ID	Process definition	
	P1	Move forward (MF)	
	P2	Turn Left (TL)	
	P3	Turn Right (TR)	
	P4	Stay (ST)	
Connections	Judgement Node Connections		Processing Node Connections
	J1, J2, J3, J4 → Floor, Obstacle, Agent, Prey; J5, → Forward, Backward, Left, Right, Nothing		MF, TL, TR, ST → Next Node. As there is no trapped location for the prey in this problem, we didn't use the constraint conformance technique in this experiment.
Number of Steps	Each agent is allowed to take a maximum of 60 steps to solve the problem.		
Chromosome Structure			
Number of Nodes	A total of 120 number of nodes is defined for the problem. 40 → processing nodes and 80 → judgment nodes.		
Number of sub-nodes	Only one sub-node in this experiment.		
Number of Sub-programs	Only 1 sub program with index 0.		
CDA* Parameters			
Number of Exploration	150		
Type of exploring nodes	Processing nodes		
Type of extracting conflicts	Loop - Repetition in turns - Repetition in Go Forward - Repetition in Judgment nodes		
Algorithm Parameters			
Number of individuals	50		
Number of Elites	1		
Number of Crossover individuals	20		
Number of Mutation individuals	29		
Mutation rate	Pm: 0.01, Pm1: 0.01, P_{ind}: 0.5, P_{cn}: 0.1		
Crossover rate	Pc: 0.1		
Tournament selection size	2		

the fitness value is calculated by equation 3.1, and the final fitness is calculated using equation 3.2.

Conflict-Directed A*: The same technique of (CDA*) that has been used with Normal Tile World problem will be used in the Prey and Predator Problem, with considering all the conflicts.

Evolutionary Operators: The same technique of (Crossover and Mutation) that has been used with Normal Tile World problem will be used in the Prey and Predator Problem.

5.4 Testing and Analysis

In this section, we tested the best proposed algorithm, which is CDA*-GNP, and compared its results with the extensions of the GNP algorithms (GNP, GNP-CC-OS-TP, GNP-RL, and GNP-RL-CC-OS-TP). CDA*-GNP has been implemented on three problems: Normal Tile World Problem, Tile World problem with a heavy tile with two experiments (one sub-program & two sub-programs), and Prey and Predator problem. Each result is an average of three experiments. The experiments were done in two

stages: training and testing stages. In both stages, the algorithm has been applied to the training set (1). Although the training and testing results on the same environments are the same for the algorithms (GNP, GNP-CC-OS-TP, CDA*-GNP), the training results are different from the testing results for the algorithms (GNP-RL and GNP-RL-CC-OS-TP) because of the exploration and exploitation face on the training stage. The results on the charts below presented the fitness value and the number of dropped tiles on holes for the best chromosome that every generation has generated.

As mentioned before CDA*-GNP spend more evaluations in each generation than the other algorithms spend, The Table 5.8, Table 5.9, Table 5.10, and Table 5.11 below show the number of evaluations that are used to reach the top training results and to reach the best chromosome for every algorithm. These tables have been added to give a fair comparison between the algorithms.

5.4.1 Normal Tile World Problem Results

Figures 5.12 and 5.13 show the training results for five algorithms (GNP, GNP-CC-OS-TP, GNP-RL, GNP-RL-CC-OS-TP, CDA*-GNP) within the first 1000 generations. The charts illustrate the fitness value and the number of dropped tiles for the best chromosome in each generation. CDA*-GNP was able to reach the top performance within 44626 evaluations compared with the second-ranking algorithm GNP-RL-CC-OS-TP which reached the top within 172500 evaluations as shown in Table 5.8. The other algorithms (GNP, GNP-CC-OS-TP, and GNP-RL) could not reach the top performance for the Tile World problem within the first 1000 generations.

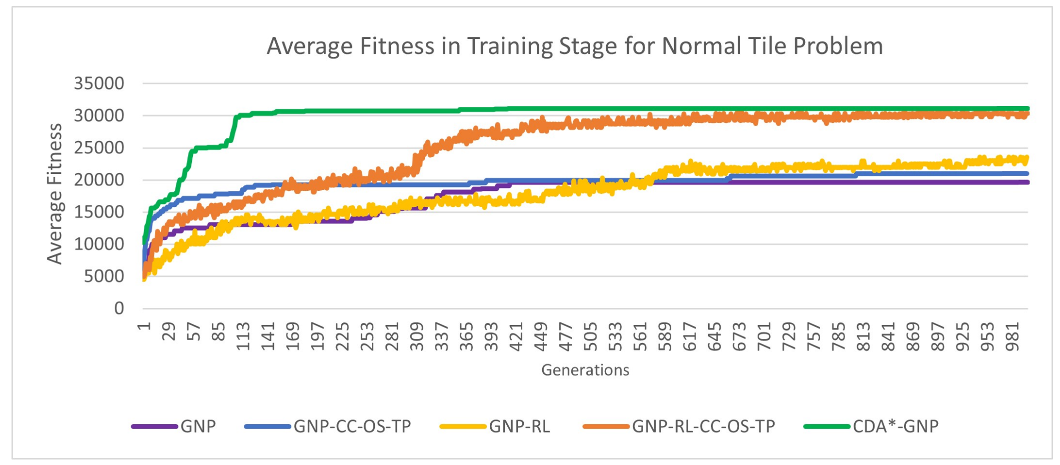


Figure 5.12: Average fitness during Training (Normal Tile World Problem).

Figures 5.14 and 5.15 show the testing results for the five algorithms. The ranks of the algorithms are the same in the testing stage, as the testing results for (GNP, GNP-CC-OS-TP, and CDA*-GNP) have not changed from the training, the testing

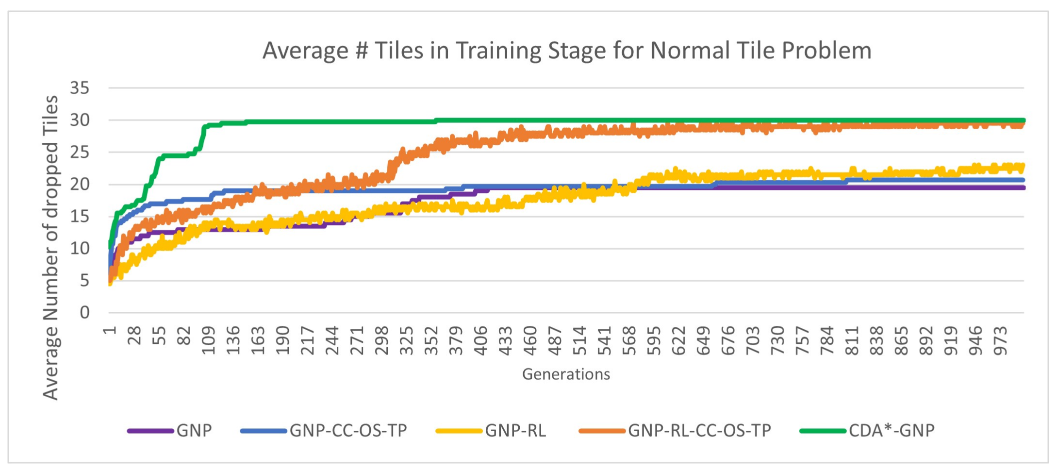


Figure 5.13: Average number of dropped tiles during Training (Normal Tile Problem).

results for (GNP-RL, and GNP-RL-CC-OS-TP) have changed, because the exploration and exploitation operations for the Reinforcement learning in the training stage that is not used in the testing stage. Table 5.8 shows the best chromosome for each algorithm in this problem. The best chromosome has been chosen depending on many factors. Firstly, we looked for an individual that could reach 30/30 dropped tiles on the Training Set (1). Secondly, the same chromosome can have the maximum number of dropped tiles for the Testing Set (2) and Testing Set (3). The best individual for the CDA*-GNP algorithm was able to reach 30/30, 29/30, and 30/30 dropped tiles for the Sets (1), (2), and (3) respectively, which was the best result from the five algorithms. Comes next is the GNP-RL-CC-OS-TP which was able to reach 30/30, 22/30, and 25/30 dropped tiles for the Sets (1), (2), and (3) respectively. The other algorithms were not able to succeed in getting the top results on the training Set (1), As they received 21/30, 24/30, and 17/30 for GNP, GNP-CC-OS-TP, and GNP-RL respectively.

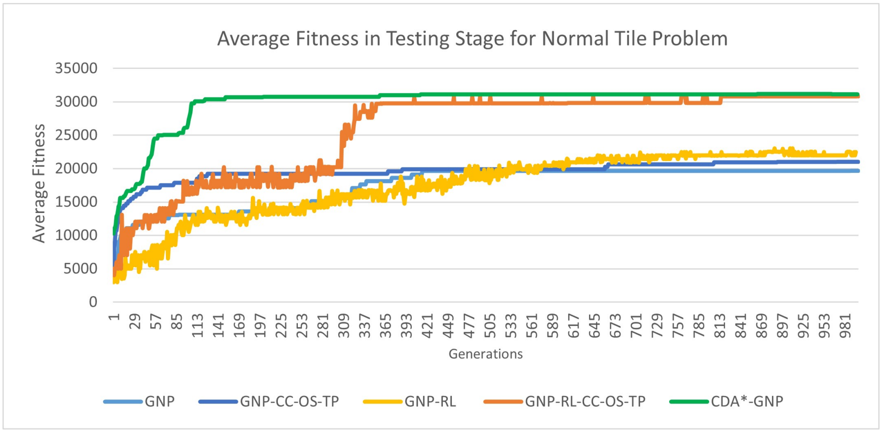


Figure 5.14: Average Fitness (Testing Stage), Normal Tile World Problem.

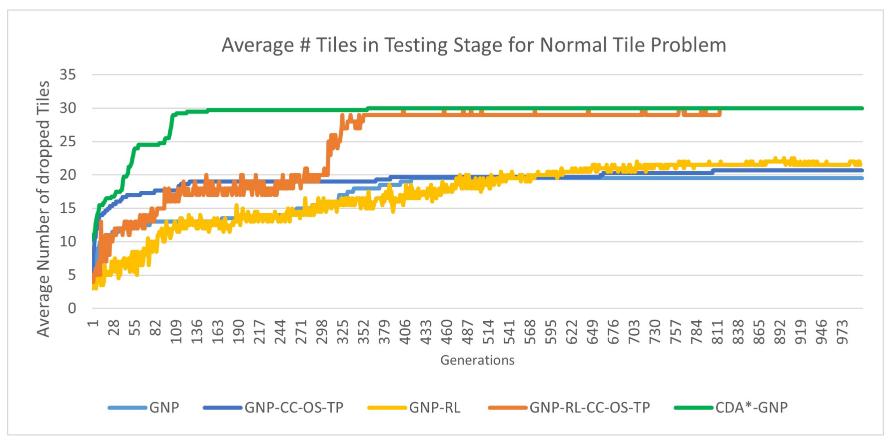


Figure 5.15: Average number of dropped tiles (Testing Stage), Normal Tile World Problem.

Table 5.8: Performance comparison on the Tile World Problem.

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Average Number evaluation until get the top training result	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
GNP	17475	5	0.00E+00	19672	4515	21/30 S1 02/30 S2 08/30 S3	-	282300	940
GNP-CC-OS-TP	19642	3	0.00E+00	21001	5010	24/30 S1 06/30 S2 15/30 S3	-	266700	888
GNP-RL	18098	4	0.00E+00	23587	4520	17/30 S1 02/30 S2 13/30 S3	-	273300	910
GNP-RL-CC-OS-TP	25378	2	3.63E-91	30946	5021	30/30 S1 22/30 S2 25/30 S3	172500	183900	612
CDA*-GNP	29969	1	-	31157	10275	30/30 S1 29/30 S2 30/30 S3	44626	251325	611

5.4.2 Tile World Problem with Heavy Tile using one sub-program for the Graph

The same algorithms have been applied to the heavy version of the Tile World problem using one sub-program. Figure 5.16 and 5.17 show the average fitness value and number of dropped tiles for three experiments for each algorithm on 1000 generations. CDA*-GNP achieved the best training and testing results on the heavy tile problem; within an average of 289 generations (124548 evaluations), the algorithm was able to reach the top result of 30/30 dropped tile on the Training Set (1), whereas the GNP-RL-CC-OS-TP which received the second-ranked algorithm was able to reach the top training result on an average of generation number 279 within 84000 evaluations. The other algorithms (GNP, GNP-CC-OS-TP, and GNP-RL) were not able to reach the top results within the first 1000 generations.

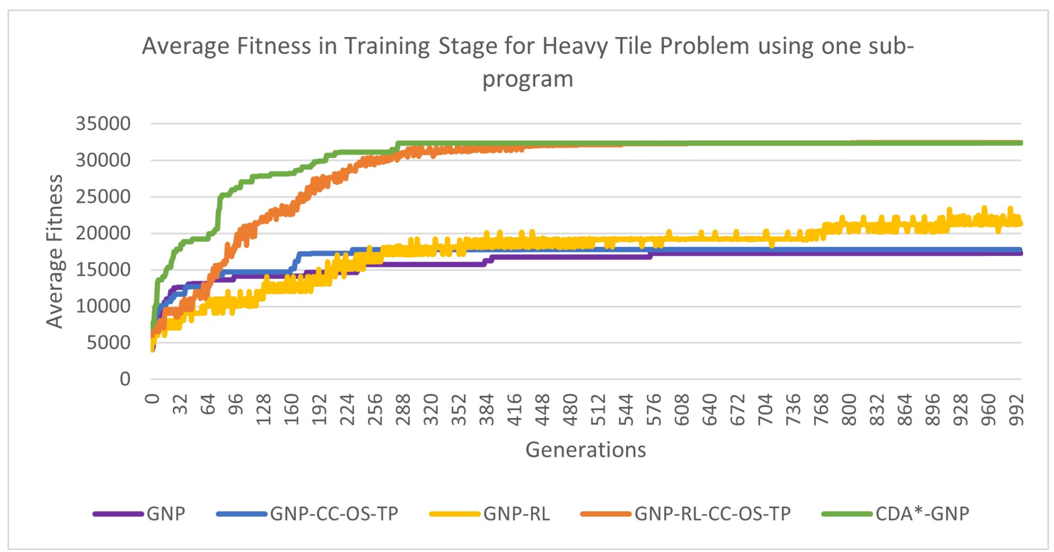


Figure 5.16: Average fitness during training (Heavy Tile World Problem) using one sub-program.

Figure 5.18 and 5.19 illustrate the testing results for the five algorithms on the Heavy Tile World. It is clear to notice that the CDA*-GNP received the best testing results that can successes in figuring out a chromosome that could get 30/30, 22/30, and 30/30 dropped tiles on the Training Set (1), Testing Set (2), and Testing Set (3) respectively. The algorithm reached this result on generation number 999 with 451368 evaluations. While GNP-RL-CC-OS-TP received 30/30, 17/30, and 30/30 on the Training Set (1), Testing Set (2), and Testing Set (3) respectively, within 131100 evaluations in generation number 436 (see Table 5.9). Although the number of evaluations and the generation where we found the best chromosome for GNP-RL-CC-OS-TP was less than the ones for CDA*-GNP, the performance of the best chromosome in CDA*-GNP

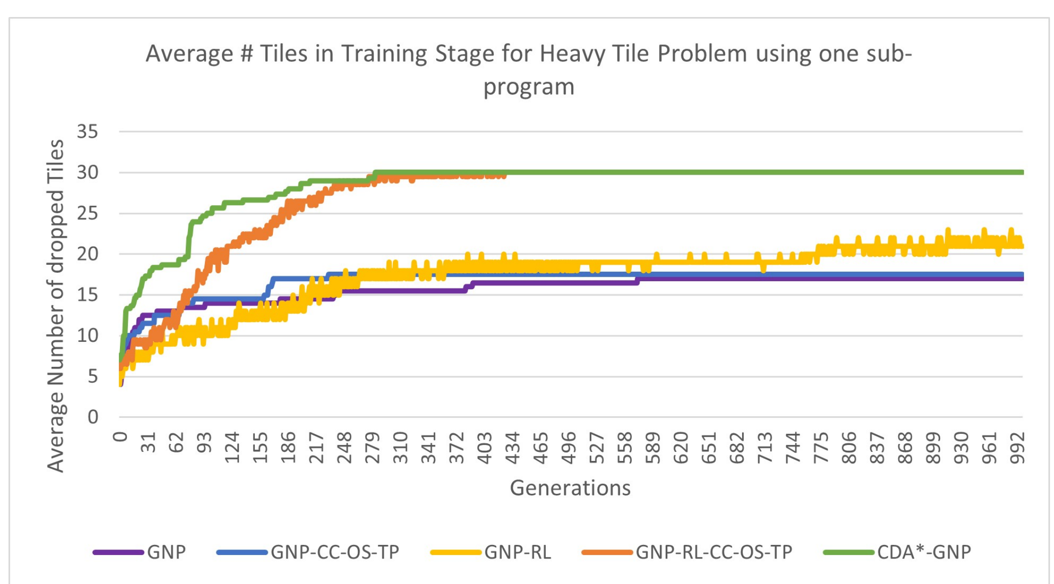


Figure 5.17: Average number of dropped tiles during training (Heavy Tile World) using one sub-program.

is higher than the one on the GNP-RL-CC-OS-TP. However, we also trained the GNP-RL-CC-OS-TP for 5,000 generations for 1,500,000 evaluations, but we could not get a chromosome that beat the one from CDA*-GNP. In [17], we proved that the GNP-RL-CC-OS-TP was an over-trained algorithm, which means the results are increased in the training results but not in the testing results.

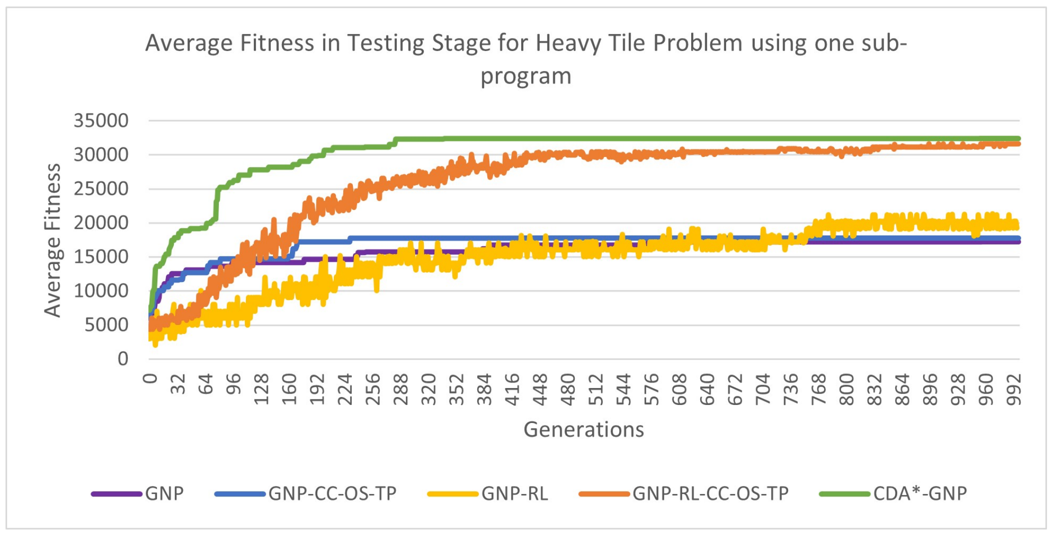


Figure 5.18: Average Fitness (Testing Stage, Heavy Tile World) using one sub-program.

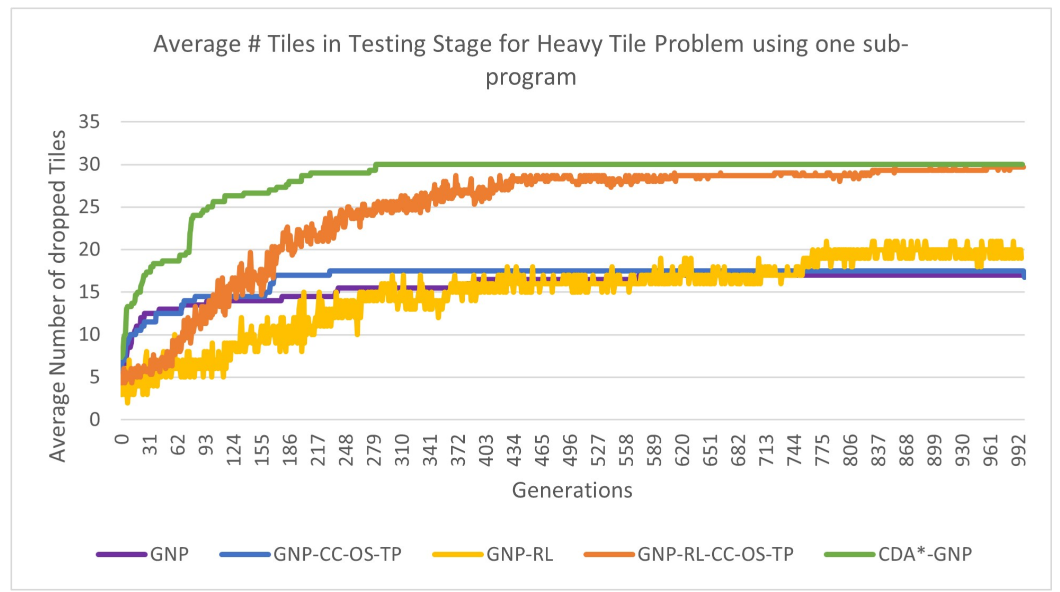


Figure 5.19: Average number of dropped tiles (Testing Stage, Heavy Tile World) using one sub-program.

Table 5.9: Performance of Algorithms (using one sub-program) on the Heavy Tile World.

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Average Number evaluation until get the top training result	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
GNP	16018	5	0	17247	4029	19/30 S1 02/30 S2 08/30 S3	-	260700	868
GNP-CC-OS-TP	16971	4	0	17749	6022	24/30 S1 03/30 S2 08/30 S3	-	300000	999
GNP-RL	17519	3	0	23499	4037	21/30 S1 07/30 S2 15/30 S3	-	291300	970
GNP-RL-CC-OS-TP	28965	2	3.16E-10	32423	6031	30/30 S1 17/30 S2 30/30 S3	83700	131100	436
CDA*-GNP	30522	1	-	32395	7148	30/30 S1 22/30 S2 30/30 S3	124548	451368	999

5.4.3 Solving the Heavy Tile World Problem using two sub-programs for the Graph

The heavy version of the Tile World problem was tested using two sub-programs. Figure 5.20 and 5.21 display the average fitness value and number of dropped tiles for each algorithm for three experiments over 1000 generations. CDA*-VSGNP received the best training and testing results on the heavy tile problem; within an average of 60 generations (27494 evaluations), the algorithm was able to reach the top score of 30/30 dropped tile on the Training Set (1), whereas the VSGNP-RL-CC-OS-TP which received the second-ranked algorithm was able to reach the top training result on an average of generation number 418 within 125400 evaluations. The other algorithms (VSGNP, VSGNP-CC-OS-TP, and VSGNP-RL) were unable to achieve the top results within the first 1000 generations.

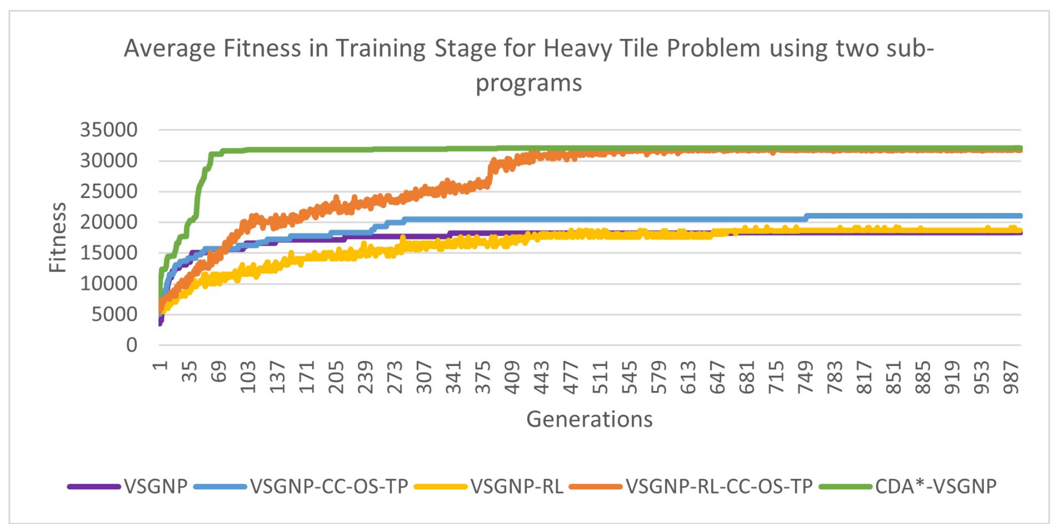


Figure 5.20: Average fitness during training on the Heavy Tile World: using two sub-programs.

Figures 5.22 and 5.23 illustrate the testing results for the five algorithms on the Heavy Tile World. It is clear to notice that the CDA*-VSGNP received the best testing results, taking into consideration the number of evaluations and the number of generations that can succeed in figuring out a chromosome that could get 30/30, 23/30, and 30/30 dropped tiles on the Training Set (1), Testing Set (2), and Testing Set (3) respectively. The algorithm reached this result on the 860th generation after conducting 377448 evaluations. While VSGNP-RL-CC-OS-TP received 30/30, 17/30, and 30/30 on the Training Set (1), Testing Set (2), and Testing Set (3) respectively, within 198600 evaluations on the 661st generation (see Table 5.10). Although the number of evaluations and the generation where we found the best chromosome for VSGNP-RL-CC-OS-TP was less than the ones for CDA*-VSGNP, the performance of

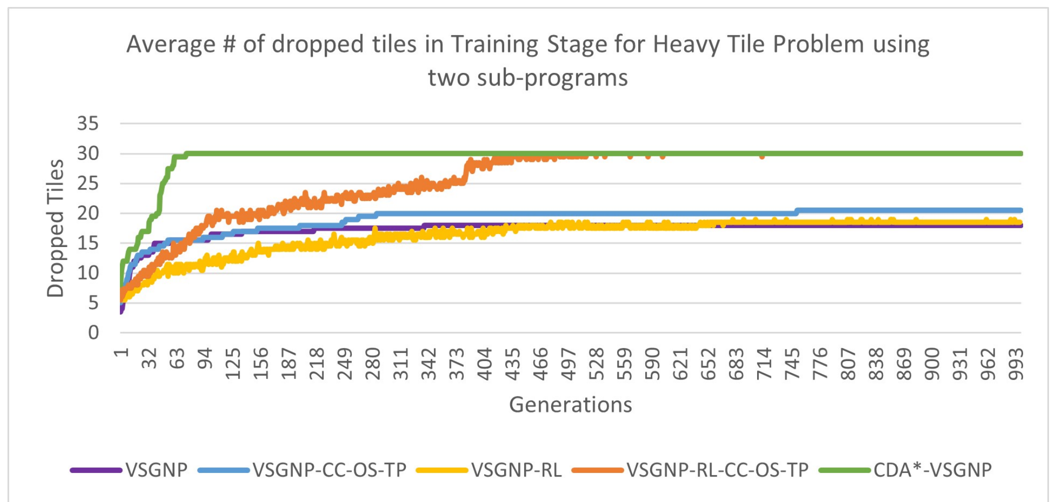


Figure 5.21: Average number of dropped tiles duing training on the Heavy Tile World: using two subprograms.

the best chromosome in CDA*-VSGNP is higher than the one on the VSGNP-RL-CC-OS-TP. Overall, using the one sub-program with the GNP-RL and its variance was better than using the variable-sized graph, while with the CDA*-VSGNP, using the variable-sized graph achieved the best results on all algorithms.

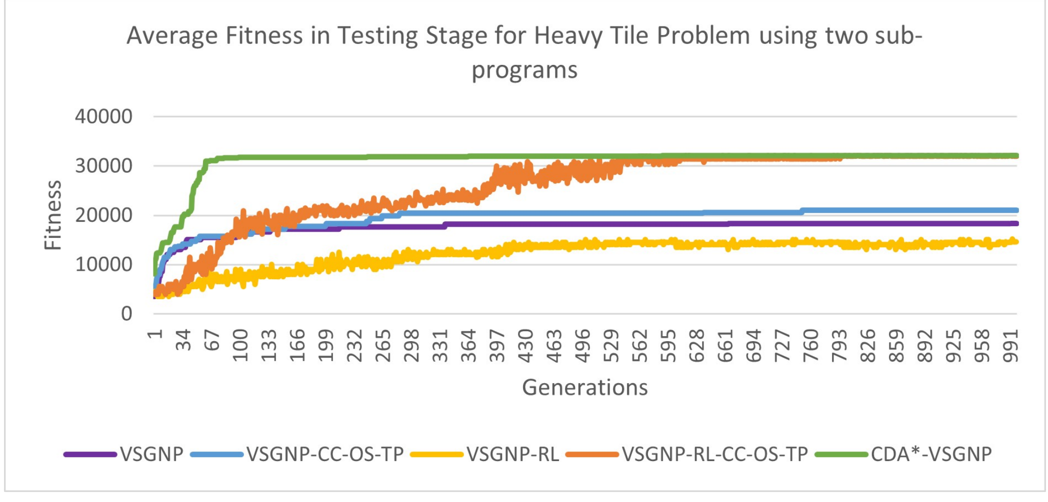


Figure 5.22: Average fitness (Testing Stage, Heavy Tile World): using two subprograms.

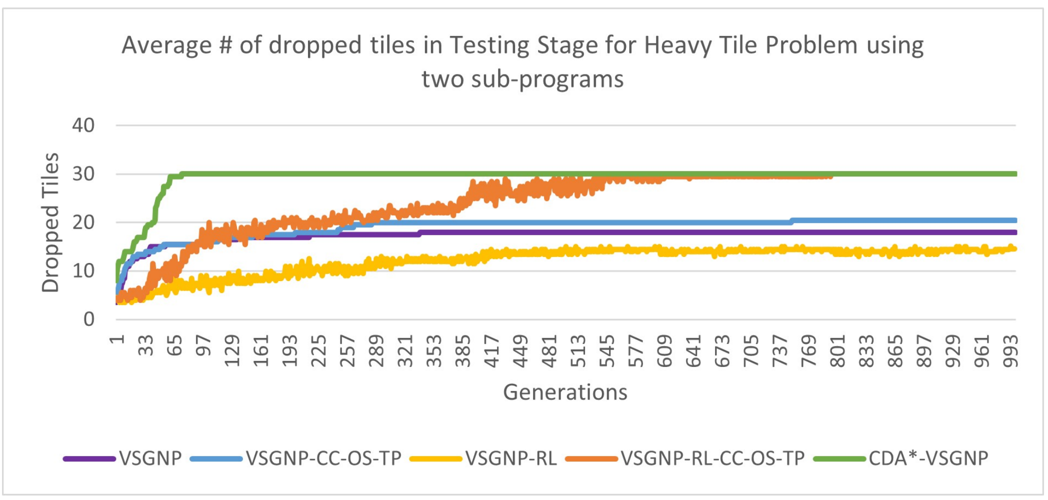


Figure 5.23: Average number of dropped tiles (Testing Stage, Heavy Tile World): using two subprograms.

Table 5.10: Performance of Algorithms (using two sub-programs) on the Heavy Tile World.

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Average Number evaluation until get the top training result	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
VSGNP	17620	4	0	18297	3520	19/30 S1 02/30 S2 09/30 S3	-	229800	765
VSGNP-CC-OS-TP	19533	3	0	21022	5016	21/30 S1 06/30 S2 13/30 S3	-	163800	545
VSGNP-RL	16480	5	0	19198	5525	16/30 S1 07/30 S2 13/30 S3	-	173700	578
VSGNP-RL-CC-OS-TP	27200	2	5.98E-67	31951	5523	30/30 S1 17/30 S2 30/30 S3	125400	198600	661
CDA*-VSGNP	31231	1	-	32124	8126	30/30 S1 23/30 S2 30/30 S3	27494	377448	860

5.4.4 Prey and Predator Problem Results

Prey and Predator problem is much easier than the Tile World Problem; we decided to apply the algorithms on this problem to test the generalization of the algorithms. As the Prey and Predator Problem is an easy problem, all the algorithms could reach 30/30 caught preys on the training results within the first 1000 generations as in Figure 5.25. The strength of the algorithms can be compared by the fitness values as in Figure 5.24. When the fitness value for the algorithm is high, that means the algorithm was able to catch the prey within less time than the other, and that clearly appeared on the CDA*-GNP algorithm. In the testing stage, because the Prey is moving randomly, we received different results each time we tested the best chromosome. In order to get a fair result when testing the best chromosome, we decided to test the best chromosome 30 times and then get the average. As the prey and predator environment has 30 environments and we run the test 30 times, the optimal solution should be 900/900 caught Prey.

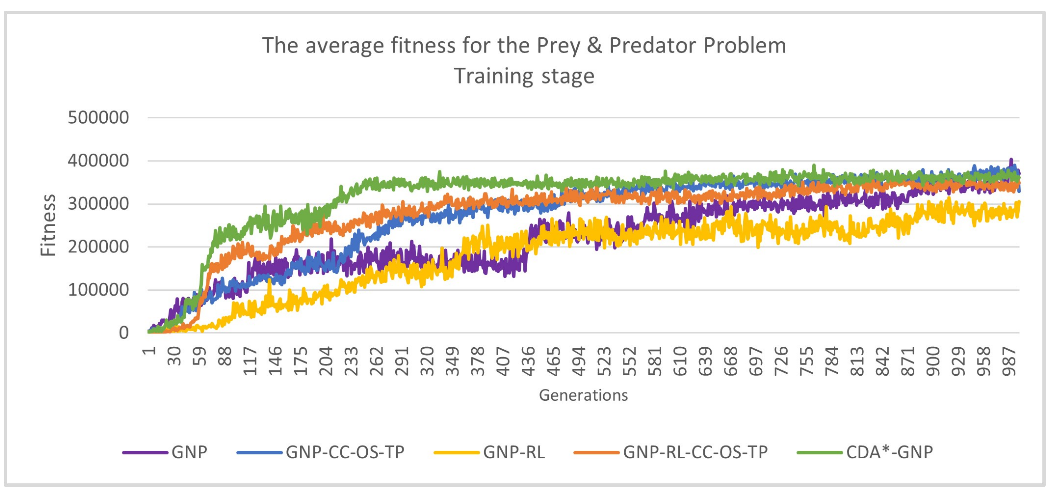


Figure 5.24: The average fitness for the Prey & Predator Problem – Training Stage.

In Figures 5.26 and 5.26, CDA*-GNP received the first ranked algorithm on the average fitness value and average hunted preys with finding the best chromosome on generation number 600 within 32449 evaluations which get 847/900 (average of 28/30) hunted preys. While GNP-RL-CC-OS-TP, GNP-CC-OS-TP and GNP received an average of 27/30 hunted prey, at the end, GNP-RL received 26/30 hunted prey (see Table 5.11).

GNP-RL-CC-OS-TP and GNP-RL results have decreased in the testing stage especially in the fitness value. The reason for the low results for the GNP-RL algorithms here is that the prey and predator is a simple problem, and the size of the GNP-RL chromosome is larger than the other algorithms because each node has a maximum of four sub-nodes. That makes the size of the problem search (the available possibilities)

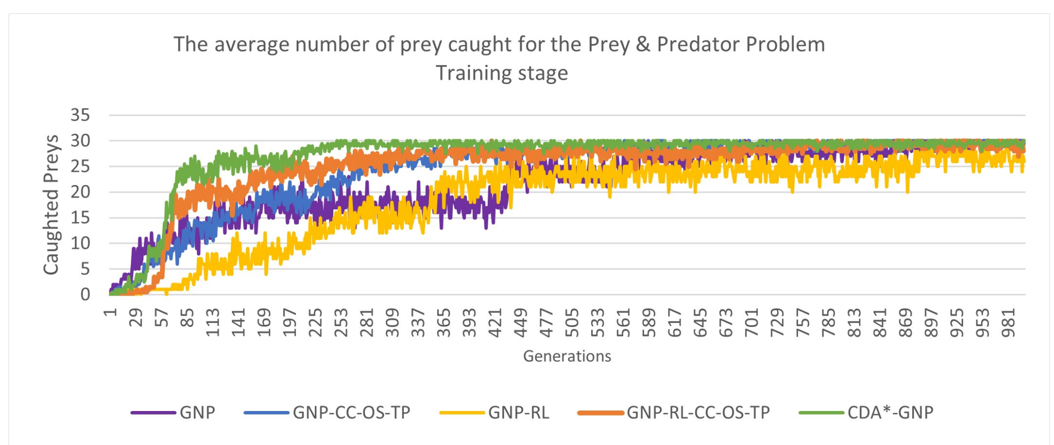


Figure 5.25: The average number of Prey caught for the Prey & Predator Problem – Training Stage.

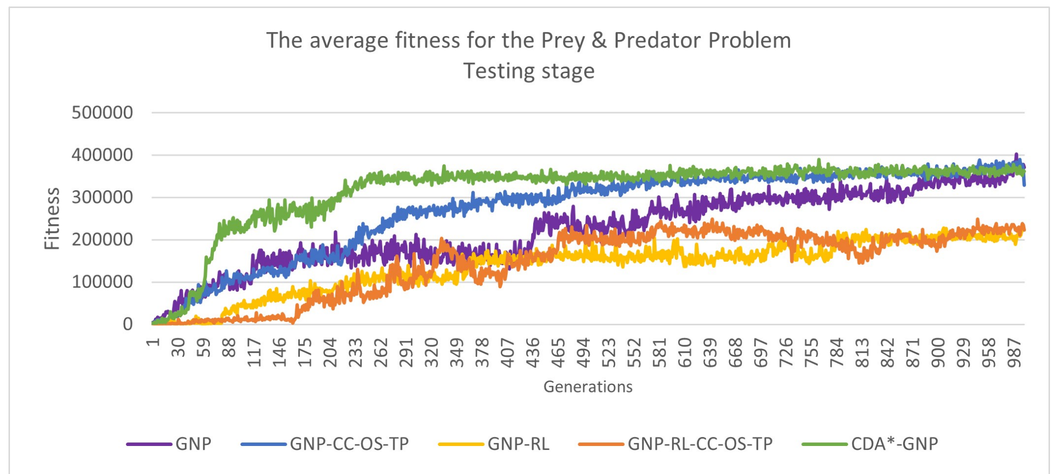


Figure 5.26: The average fitness for the Prey & Predator Problem - Testing stage.

is bigger than the others, while the prey and predator problem does not need this giant chromosome to be solved, and that what makes the other algorithms increase the results faster than the GNP-RL algorithms.

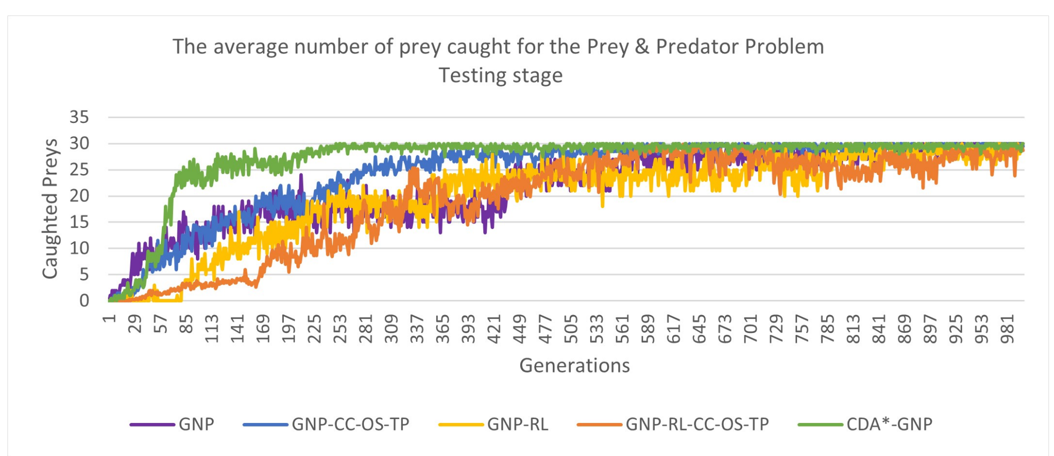


Figure 5.27: The average number of Prey caught for the Prey & Predator Problem - Testing stage.

Table 5.11: Performance of algorithms on the Prey and Predator Problem.

Algorithm	Mean of Fitness	Rank	P-value	Max of Fitness	Min of Fitness	Best training results Best Chromosome	Average Number evaluation until get the top training result	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
GNP	224856	4	7E-114	402885	3662	831/900 27/30	30050	50000	999
GNP-CC-OS-TP	273117	3	5.9E-26	389224	999	826/900 27/30	18850	50000	999
GNP-RL	184172	5	1.2E-210	313855	2700	782/900 26/30	46350	49200	983
GNP-RL-CC-OS-TP	279561	2	1.7E-23	363331	2745	829/900 27/30	20900	50000	999
CDA*-GNP	317791	1	-	389523	1729	847/900 28/30	12686	32449	600

5.5 Summary

In this chapter, we added the CDA* to GNP algorithm to add a systematic improvement to the node function selection inside the graph. The CDA* works to explore the possible node functions, excluding the functions that cause a conflict inside the graph. This algorithm proved its efficiency in providing higher performance results in less time than the other algorithms when applying it to three problems (Tile World Problem, Tile World Problem with Heavy Tile, and Prey and Predator Problem).

Chapter 6

Using Conflict Directed A* with GNP in a Heterogeneous System

6.1 Motivation

Cooperative multi-agent systems (MAS) are systems where multiple agents work together to solve tasks or maximize utility through their interactions. As the number of agents or their complexity increases, the complexity of multi-agent problems can escalate rapidly due to their interactions. Designing control rules for multi-agent systems is challenging because an agent's behaviour depends not only on its interactions with the environment but also on the behaviour of other agents. As the number of interacting agents in the team increases, or when agent behaviours become more sophisticated, creating appropriate control rules becomes increasingly complex. This complexity is especially pronounced when agents are expected to coordinate or cooperate to achieve a common goal collectively. The evaluation process in the heterogeneity systems is very long and complicated due to the high number of possibilities of forming a team out of a multitude of agents to choose from. For each generation to evaluate the teamwork, it is impossible to evaluate all the possible combinations of different types of agents, so Gomes et al. [99] used a random 30 different combinations team for evaluating 30 simulations in each generation; this number is much less than the number of possible combinations. That could result in losing the optimal solution. Applying the CDA*-GNP to heterogeneity systems will help in two ways. The first one is increasing the team's performance by exploring all the possible processing nodes in the members' graphs and choosing the best combination of the processing nodes for the best Team. The second way is to use the CDA* to explore all the possible combinations of team members that are not causing a conflict between each other by simulating them in an environment. Using CDA* will ensure that any member who causes a conflict with another member of the team is excluded and that the best member is found to replace this

one. By using this technique, we ensure that the team's performance increases within a minimum training time. Therefore, this approach will combine Conflict-Directed A* with the GNP in the heterogeneity systems to find the best team agents.

6.2 Related work

Genetic heterogeneity in multiagent systems is the ability of different agents with different controllers to execute the same task effectively by dividing the work between them [100] [101] [102] [103]. It has been used in many problems such as the Robot Soccer [104] [105], predator-prey pursuit [106] [107], etc. Evolutionary algorithms have been used to improve the cooperation between agents in genetically Heterogeneous system. There are two types of heterogeneous systems: team learning and concurrent learning [100] [108]. Team learning combines the structure for all agents to be in one chromosome (each agent has different genes) and evolves it using one population, whereas concurrent learning uses a different chromosome for each agent with more than one population for the evolution running in parallel. Many studies [109] [110] [111] [112] have proved the superiority of using concurrent learning over team learning.

Cooperative coevolutionary algorithms (CCEAs) [110] use concurrent learning with a fully heterogeneous technique; that means there is more than one population in the experiment, and each population is related to a specific agent (one-to-one) mapping [109]. To evaluate an individual, a group of agents should be evaluated together by running the simulation for the team, as each agent is represented by a different chromosome from a different population. By using a fitness function, the fitness value for each agent is the same value for all of the team. Thus, the evaluation depends on the effect of the agents working in a group. The number of agents in the problem can limit the algorithm's ability to expand cooperation [100]. The bigger number of agents leads to an increase in the used memory space and the training time due to the number of individuals in each population.

Cooperative coevolution must address the complex dynamics arising from multiple coevolving populations [113]. Some of the primary challenges include convergence to mediocre stable states [114] , [115] and loss of fitness gradients [114]. The classic CCEA architecture has inherent scalability issues related to the team's number of agents. As each agent evolves into a separate population, the number of populations increases with the number of agents. This expands the search space and raises computational complexity. Furthermore, when the team is large, the impact of a single agent's behaviour on the team can be almost unnoticeable, causing the fitness gradients to disappear [116]. The problem of scalability is also related to the issue of reinvention [114]. CCEAs often divide agents into separate groups, establishing a clear separation between them. In many multi-agent tasks, there can be significant overlap between the policies of each

agent [117]. The evolutionary process may inefficiently duplicate learning behaviors across multiple populations, resulting in resource wastage. In previous studies, the issue of reinvention has been approached by pre-programming the common skillset in the robots [118], or by implementing a shaping phase to develop the fundamental all robots' skillsets [119]. Another approach was used in CONE-2 [117], which is that evolution begins with a single agent and additional agents are added based on existing ones throughout the evolution process. That means CONE-2 adapts the number of agents for the problem and decreases the reinvention issue.

One method to enhance the scalability of multiagent learning is reducing heterogeneity within the system [120]. Reducing heterogeneity decreases the number of agent controllers that need to be learned, which will improve scalability [121]. The issue of reinvention could be addressed using partial heterogeneity, as performing the same task by different agents means that these agents are related to the same homogeneous sub-team, which will avoid evolving similar individuals in different populations. Previous research on hybrid team evolution has focused on team learning. Luke [121] developed hybrid teams for the RoboCup challenge, specifying the team composition manually beforehand. Each genome consisted of a collection of GP trees that encoded the multiple sub-team's behaviour. In [122] Hara proposed a new technique called Automatically Defined Groups (ADG), that discovers the best number of groups and their compositions. The Legion System was proposed by Bongard [123], which is based on genetic programming. In this approach, one sub-tree was designed for each behaviour class while the genome encodes the team's composition. Lichocki et al. [124] employed crossover operators to evolve hybrid teams by swapping agents between teams. In [125] a different neuroevolution approach was implemented. A single genome was used to encode the agent controllers, and HyperNEAT (the indirect encoding technique) was used to take advantage of similarities in agents' policies. HyperNEAT enables agents to share policies while still displaying differences. Although storing the entire team's genetic information in one genome can make it easier for team compositions to evolve, these approaches do not have the benefits offered by coevolutionary algorithms. In order to manage large and heterogeneous multi-robot systems, Nitschke [126] proposed a method called CONE, which allows organized generation between different populations, with each corresponding to a different agent. If there are different populations sharing the same specialisations, it is allowable to apply crossover between them. The experimenter manually specifies the potential specializations. It has been demonstrated that CONE can enhance coevolution performance in tasks requiring a high degree of specialization. However, the evolved teams remain fully heterogeneous, as complete controllers are not shared by different agents. In many heterogeneous multi-agent systems, more than one agent can share the same sensors and abilities [127]. One way

to solve this problem is to use one population for more than one agent, as in previous research [100] [104] (see Figure 6.1 (a)). In [99], a Hyb-CCEA has been used as an extension of CCEA to use a dynamic number of populations by allowing the algorithm to merge and separate the populations to decrease and increase heterogeneity, respectively.

In 2018, Gomes et al. proposed an improved version of Hyb-CCEA, merging and splitting the population without prior input from the programmer, detecting the agent behaviour for each population and deciding which populations can be merged [128]. Firstly, the first populations are initialized randomly with a random number of populations; it could be one population for each agent or one population for more than one agent. Then, after evaluating the chosen individual for each agent in the team, the merge and split operations are applied to the populations. Merging the population is done depending on the measured distance between the sets of behaviour (see Figure 6.1 (b)). For each population, a set of behaviours is extracted after evaluating the individuals, which contains the average value for each sensor and action for the agent within the evaluation process (simulation). If the distance between the two populations is low then these two populations are similar, and they will be merged. Splitting the population is done on the populations whose age is more than the maturation period which is a random number that has been assigned to the population when initializing it. The population with the highest age is chosen to split its agent and individuals randomly into two new populations that have the same random maturation number (see Figure 6.1 (c)).

6.3 Proposed Algorithms (Architecture)

To test the CDA*-GNP with a heterogeneous system, we modified the Heavy Tile World Problem by replacing one of the agents in each environment with a stronger one that can push a heavy tile alone and also push the normal tile. Then we combined all the 10 environments into one big environment that has 20 agents, 10 strong agents, 20 tiles, 10 heavy tiles, and 30 holes (see Figure 1.3). In this proposed algorithm, we will use fully and partly heterogeneous systems, which means a separate population for each agent for the fully heterogeneous system and a shared population for a number of agents who share the same behaviour for the partly heterogeneous system. Each population has 300 individuals that are generated randomly in the first generation (see Figure 6.2).

The CDA*-GNP has been used in two ways in the heterogeneous system. The first way, CDA* is used to explore the nodes and find the best function for each node, as it was used before with the homogeneous system. The second way, CDA* is used to explore the individuals for each agent and find the best chromosome that represents

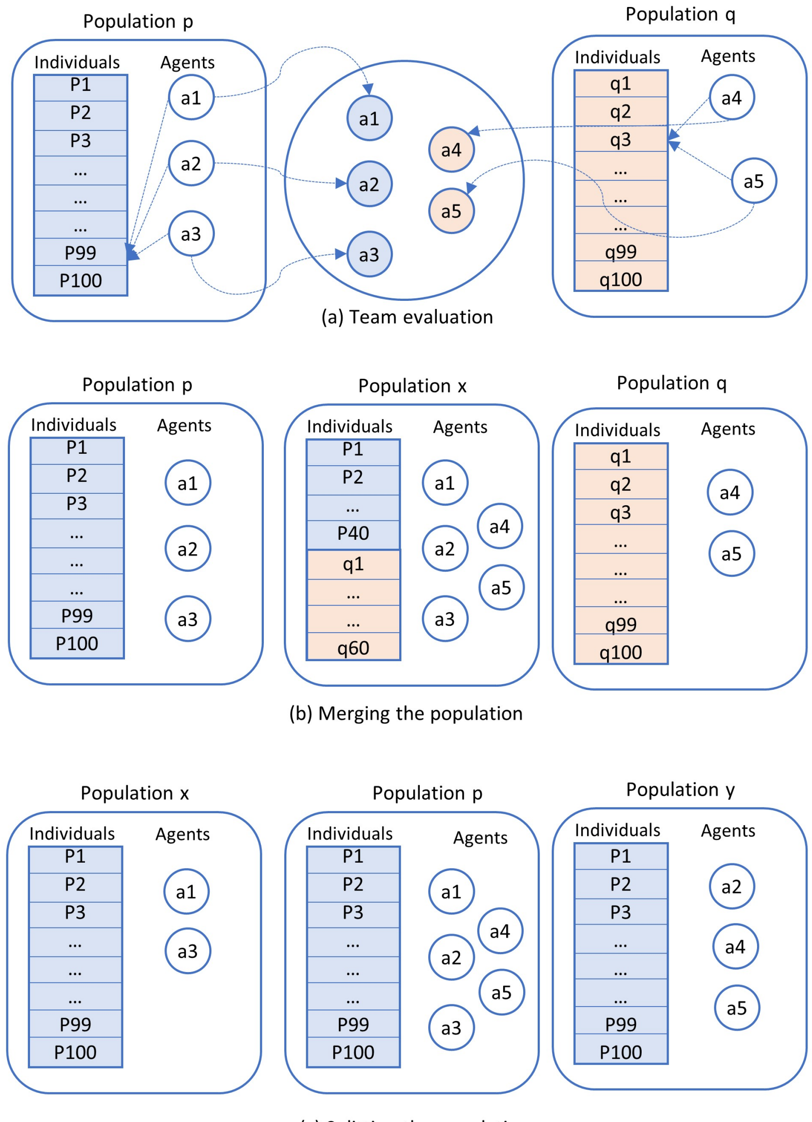


Figure 6.1: Merging and Splitting the population in the Teams. (a) Evaluating a team by combining the assigned individual for each agent, (b) merging the population, (c) Splitting the population [128]

each agent and doesn't conflict with another agent's individual.

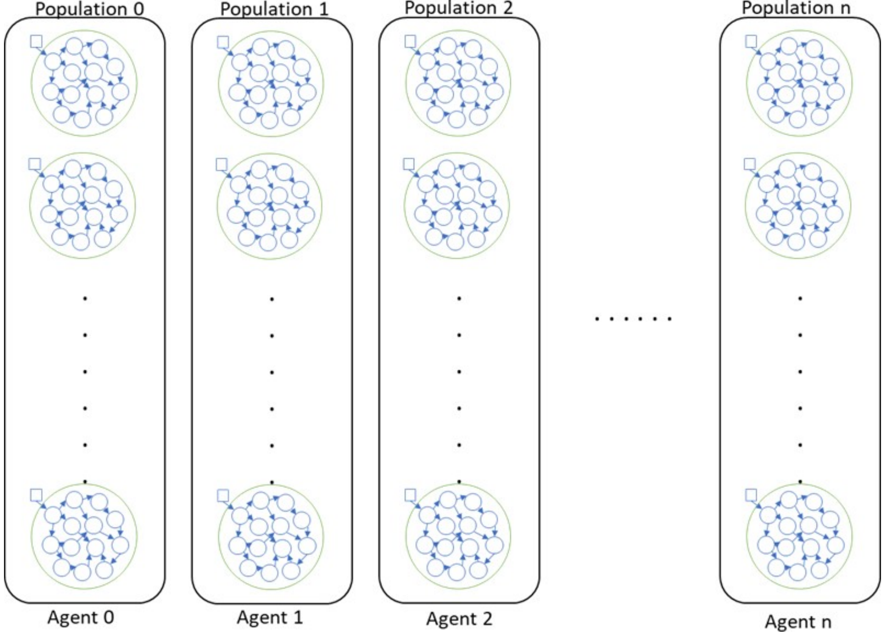


Figure 6.2: Fully heterogeneous system structure

6.3.1 Random Heterogeneous system

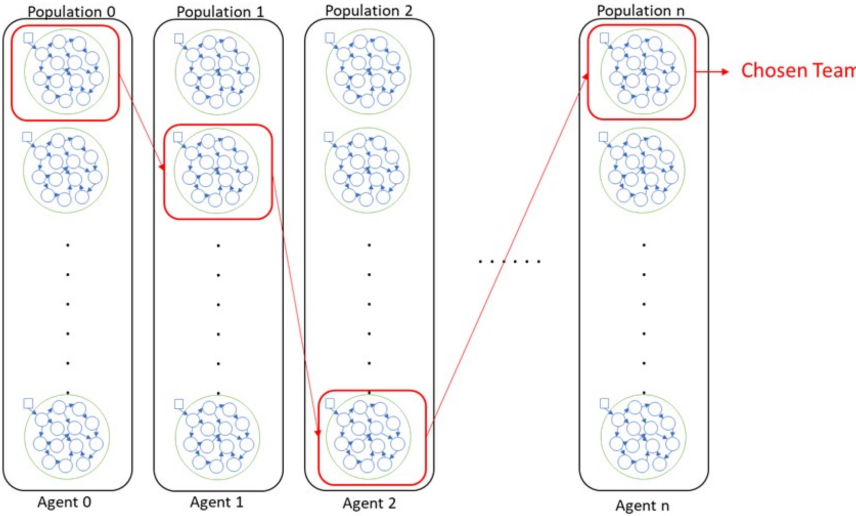


Figure 6.3: Randomly chosen team

In a fully heterogeneous system, each agent has its own population. In our example, each population has 300 individuals. To create a team, one individual from each population will be chosen to represent an agent. The selection of the individual can be random or according to a specific technique. In the first experiment, the selection of the team members will be random, like the way used by Gomes et al. [128]. In this experiment, 150 random teams will be chosen from the populations to create 150 teams. Then, the algorithm evaluates these teams and chooses the team with the highest fitness to be passed to the next generation (see Figure 6.3).

Each chromosome has a similar structure to the chromosome in the homogeneous system as shown in Table 6.1.

Table 6.1: Parameters for the CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN).

<table>
<tr><th>Problem Domain</th><th colspan="3">CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN)</th></tr>
<tr><td>Judgement Nodes</td><td>ID</td><td>Query Definition</td><td>Possible response to query</td></tr>
<tr><td rowspan="8">Considering that the direction here is calculated using the A* algorithm. It gives the direction to the first point in the shortest path to the goal that A* finds [17] and [14]. More explanation about how to use A* with heterogenous system describe below (*).</td><td>J1</td><td>Judge what is in front position (JF)</td><td rowspan="4">- Agent
- SAgent
- Tile
- Htile
- Hole
- Floor
- Obstacle</td></tr>
<tr><td>J2</td><td>Judge what is in back position (JB)</td></tr>
<tr><td>J3</td><td>Judge what is in right position (JR)</td></tr>
<tr><td>J4</td><td>Judge what is in left position (JL)</td></tr>
<tr><td>J5</td><td>Direction to the nearest Tile (TD)?</td><td rowspan="3">- Forward
- Backward
- Left
- Right
- None</td></tr>
<tr><td>J6</td><td>Direction to the nearest Hole (HD)?</td></tr>
<tr><td>J7</td><td>Direction to the second nearest Tile (THD)?</td></tr>
<tr><td>J8</td><td>Type of the nearest Tile (TT)?</td><td>- Normal
- Heavy</td></tr>
<tr><td>Processing Nodes</td><td>ID</td><td colspan="2">Process definition</td></tr>
<tr><td rowspan="4"></td><td>P1</td><td colspan="2">Move forward (MF) (MF for the SAgnet can push Htile and Tile without help)</td></tr>
<tr><td>P2</td><td colspan="2">Turn Left (TL)</td></tr>
<tr><td>P3</td><td colspan="2">Turn Right (TR)</td></tr>
<tr><td>P4</td><td colspan="2">Stay (ST)</td></tr>
<tr><td>Connections</td><td colspan="2">Judgement Node Connections</td><td>Processing Node Connections</td></tr>
<tr><td></td><td colspan="2">J1, J2, J3, J4 → Floor, Obstacle, SAgent, Agent, Tile, Hole, HTile
J5, J6, J7 → Forward, Backward, Left, Right, Nothing
J8 → Normal → sub-program 0; Heavy → sub-program 1

For the Judgment node (8), which returns the type of the nearest Tile, there are two connections: one if the answer is normal Tile and it will connect to a node from sub-program 0 and the other connection for the answer heavy Tile and it connect to a node from the sub-program 1. Only the Judgment node number (8) can connect to the other sub-program. All the other nodes should have a connection to the same sub-program.</td><td>MF → With Conflict, Without Conflict
TL, TR, ST → Next Node</td></tr>
<tr><td>Number of Steps</td><td colspan="3">Each agent is allowed to take a maximum of 180 steps to solve the problem.</td></tr>
<tr><th colspan="4">Chromosome Structure</th></tr>
<tr><td>Number of Nodes</td><td colspan="3">A total of 120 number of nodes is defined for the problem.
40 → processing nodes and 80 → judgment nodes.</td></tr>
<tr><td>Number of sub-nodes</td><td colspan="3">Only one sub-node in this experiment.</td></tr>
<tr><td>Number of Sub-programs</td><td colspan="3">Two sub-programs with index 0 for the first sub-program and 1 for the second sub-program. Sub-program 0 to solve the Normal Tiles which contains 72 nodes → 48 Judgment nodes and 24 Processing nodes. Sub-program 1 to solve the Heavy Tiles which contains 48 nodes → 32 Judgment nodes and 16 Processing nodes.</td></tr>
<tr><th colspan="4">Algorithm Parameters</th></tr>
<tr><td>Number of individuals</td><td colspan="3">300</td></tr>
<tr><td>Number of Elites</td><td colspan="3">5</td></tr>
<tr><td>Number of Crossover individuals</td><td colspan="3">120</td></tr>
<tr><td>Number of Mutation individuals</td><td colspan="3">175</td></tr>
<tr><td>Mutation rate</td><td colspan="3">Pm: 0.001, Pm1: 0.005</td></tr>
<tr><td>Crossover rate</td><td colspan="3">Pc: 0.001</td></tr>
<tr><td>Tournament selection size</td><td colspan="3">7</td></tr>
</table>

(*) Using A* as a judgment node, in the heterogeneity system can face two issues; the first issue is the size of the environment. Because the environment size is big that means the A* needs more time to explore all the paths and detect the shortest path. The second issue is the number of objects (Tiles, Agents, Holes) in the environment. So, when asking about the direction to the nearest tile, the algorithm first should detect the shortest path to all the tiles then the tile with the shortest path will be detected as the nearest tile, so the algorithm will return back the direction to the first way point in the shortest path for the nearest tile. That takes a long time. To solve this problem, we decided to use the Euclidean distance to detect the nearest three tiles, and then use A* to find the shortest path to each one of them. The tile with the shortest path will be chosen as the Nearest tile, and the algorithm will return the direction to the first point in it.

Why do we not just use Euclidean distance to find the nearest tile? Using the Euclidean distance to detect the direction will find the nearest tile regardless of any obstacles in the way. Suppose that we want to find the nearest tile to the $A1$. $A1(1,4)$, $T1(4,2)$, $T2(3,5)$, as in Figure 6.4 the Euclidean distance between $A1 \rightarrow T1 = 5$, $A1 \rightarrow T2 = 3$, so the nearest tile is $T2$. While using A*, the path cost $A1 \rightarrow T1 = 5$, $A1 \rightarrow T2 = 9$, so the nearest tile is $T2$. From the photo, it is clear that $T1$ is the nearest tile to $A1$. Because of that using A* is better than using Euclidean distance.

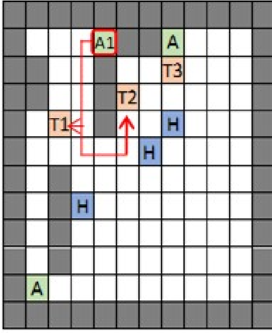


Figure 6.4: The diference between Euclidean distance and A*

To evaluate the individuals in the populations, a team of 30 members, one from each population, is created and evaluated. The fitness value that this team can get will be the fitness value for each member of this team, as the agents are working as a team, and they all get the same result.

Pseudocode 11 and Figure 6.5 illustrate the steps that the algorithm takes to apply a random heterogeneous system. Firstly, the algorithm initializes a random 300 individuals for each population that represents an agent. The aim of this algorithm is to find the Best-Team and pass it to the next generation. The algorithm is repeated until the agents finish pushing all the tiles into their holes or until a specific number of

Algorithm 11 Random Heterogeneous System

```
1: for all the Agents in the system do
2:     Populations[Agent] ← Randomly initialize the first population ( )
3: end for
4: Best-Team ← []
5: while (solution is not found) do
6:     for each i is a random team do
7:         Team[i] ← Randomly chosen members from each population
8:         Team[i].Fitness = evaluate(Team[i])
9:     end for
10:    if (Team[Highest-fitness].Fitness > Best-Team.Fitness) then
11:        Best-Team ← Team[Highest-fitness]
12:    end if
13:    for each ind in the Populations do
14:        for each a in the Agents do
15:            Team[ind] ← Populations[a][ind]
16:        end for
17:        Team[ind].Fitness ← evaluate(Team[ind])
18:        if (Team[ind].Fitness > Best-Team.Fitness) then
19:            Best-Team ← Team[ind]
20:        end if
21:        for each a in the Agents do
22:            Populations[a][ind].Fitness ← Team[ind].Fitness
23:        end for
24:    end for
25:    for each a in the Agents do
26:        Sort(Populations[a])
27:        Selection (Populations[a])
28:        Crossover (Populations[a])
29:        Mutation (Populations[a])
30:        Populations[a] ← Next Generation
31:    end for
32: end while
```

generations. In each generation, the algorithm will choose a number of teams randomly and evaluate them, the team with the maximum fitness, which beats the fitness of the Previous Best-Team will be chosen as the Best-Team. The chosen number of random Teams in our experiments is set to 150 combinations of Teams. After that, the algorithm evaluates the chromosomes in all the populations. To evaluate the individuals, each individual in a population should participate in a team and get the same fitness as the team's one. To do that, the algorithm will create 300 teams, each one of them having 30 individuals, each from a population that shares the same id. For example, Team 0 has individual 0 from population 0, individual 0 from population 1, individual 0 from population 2, and so on. The team contains the individuals with id 1 from all

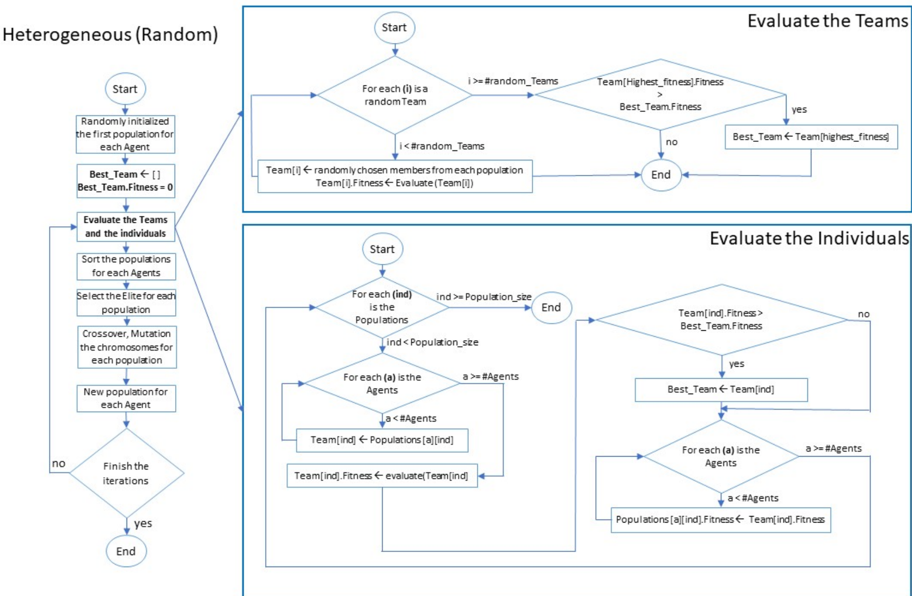


Figure 6.5: Heterogeneous (Random)

the populations etc. Each one of these 300 Teams is evaluated separately and is set its fitness value for each member in it. Depending on this fitness the individuals in the populations are sorted and the best 5 individuals are chosen as the (elite) and passed on to the next generation. After that, the Crossover and Mutation operations are applied to each population separately to generate the individuals for the new generation for each one of them.

6.3.2 CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN)

The first way of using CDA* with heterogeneous is to explore the nodes in the members' graphs. Pseudocode 12 and Figure 6.6 explain the technique used in this method. Firstly, the algorithm starts with randomly initializing the first population for each agent. For each generation, the individuals in these populations are evaluated by making each one of them join a team and getting its fitness value as explained before. The Team with the highest fitness is chosen to be the Best-Team. CDA*-GNP is applied on the Best-Team for each generation, as explained below. After that, the algorithm applies (selection, crossover, and mutation) operations to generate new individuals for each population to be passed to the next generation.

Pseudocode 13 and Figure 6.6 show in detail the technique that CDA*-GNP used to explore the nodes. It starts with setting the Target-fitness with the Best-Team fitness. Then, it detects all the conflicts from the Best-Team that are extracted from each graph

Algorithm 12 CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN)

```
1: for all the Agents in the system do
2:     Populations[Agent] ← Randomly initialize the first population ( )
3: end for
4: Best-Team ← []
5: while (solution is not found) do
6:     for each ind in the Populations do
7:         for each a in the Agents do
8:             Team[ind] ← Populations[a][ind]
9:         end for
10:        Team[ind].Fitness ← evaluate(Team[ind])
11:        if (Team[ind].Fitness > Best-Team.Fitness) then
12:            Best-Team ← Team[ind]
13:        end if
14:        for each a in the Agents do
15:            Populations[a][ind].Fitness ← Team[ind].Fitness
16:        end for
17:    end for
18:    CDA*-GNP(Best-Team)
19:    for each a in the Agents do
20:        Sort(Populations[a])
21:        Selection (Populations[a])
22:        Crossover (Populations[a])
23:        Mutation (Populations[a])
24:        Populations[a] ← Next Generation
25:    end for
26: end while
```

separately in the evaluation process. The Decision-nodes list in the Heterogeneous system contains all the visited processing nodes in all the graphs from the Best-Team together (as in Figure 6.7). After that, the algorithm calculates g( ), h( ), and f( ) for each node in the Decision-nodes. The g( ) is the Best-Team Fitness value, and h( ) is (the number of nodes in the path – level). In this algorithm, the h value is not weighted by 10 as the number of nodes in the path is a lot. It could reach a maximum of (40*30) 1200 nodes. The f( ) is g( ) – h( ). The start path is added to the Queue. Then, the algorithm explores the paths in the Queue and adds them to the Expanded-list until the Queue is empty or the algorithm reaches 150 different explorations. Finally, the algorithm chooses the path with the maximum fitness to be the Best-Team.

Algorithm 13 CDA*-GNP(Best-Team)

```
1: Target-fitness ← Best-Team.Fitness
2: Conflict-list[ ]← conflicts from all the Graphs in the Best-Team
3: Decision-nodes[ ]← All the used processing nodes in all the Graphs in the Best-
   Team
4: Calculate g( ), h( ), f( ) for all the nodes in the Decision-nodes[ ]
5: Start-path ← nodes in the Decision-nodes[ ] with their functions and g,h,f
   value.
6: Add start-path to the Queue[ ]
7: while (Queue[ ] is not empty) do
8:     Choose the path from the Queue[ ] with the maximum f( ).
9:     if the path is not in the Expanded-list[ ] then
10:        Add the path to the expanded-list[ ]
11:        Delete the path from the Queue[ ]
12:        for each (i) function of the processing nodes (1, 2, 3, 4) do
13:            if it is not in the Conflict-list[ ] then
14:                Change the function of the selected node to (i).
15:                Calculate the fitness, g( ), h( ), f( ).
16:                Add it to Queue[ ] if the fitness >= target_fitness.
17:                i=i+1.
18:                Update Queue[ ]
19:            end if
20:        end for
21:    else
22:        Delete the path from the Queue[ ]
23:    end if
24: end while
25: Choose the path with the maximum fitness from the Expanded-list[ ]
```

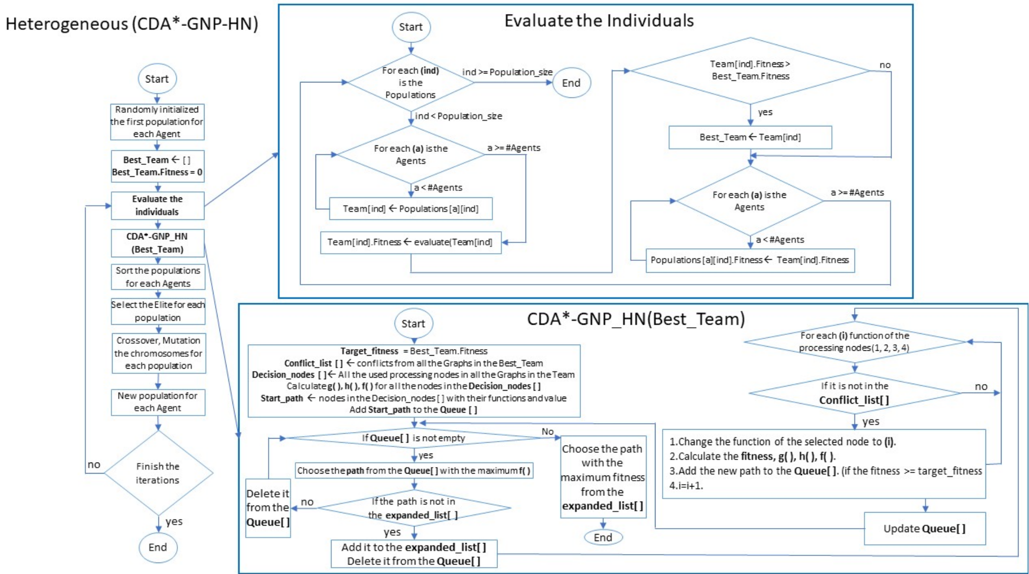


Figure 6.6: CDA*-GNP with Heterogeneous system to explore the Nodes (CDA*-GNP-HN)

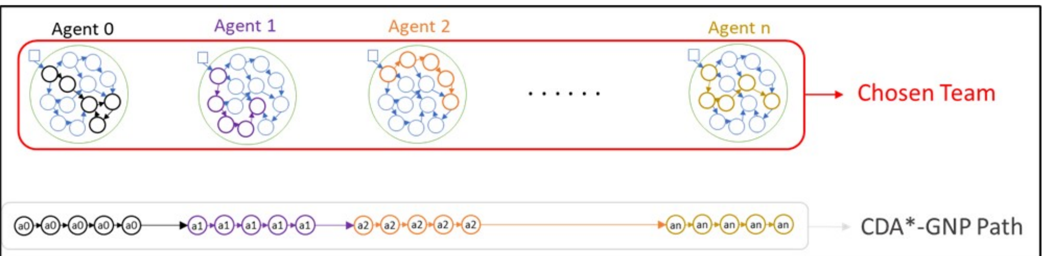


Figure 6.7: Extracting CDA*-GNP path from a heterogeneous Team

How to prevent losing diversity in the CDA*-GNP-H?

1. We keep the mutation rate at (0.001) when changing the node function as the Team contains 30 agents, which means 30 chromosomes each with 120 nodes, so that 3600 nodes are affecting the teamwork. A high mutation rate will increase the number of nodes that will be changed during the mutation process and that leads to breaking the good connections and losing the nodes that are already improved during the exploring phase.

2. Since the CDA* ensures choosing the best function for the decision nodes, we set the mutation rate when changing the connections between the nodes to (0.005). Because not all the nodes are visited by the agents when running the simulation, so changing the connection on the unused node, will not make any improvement. Because of that, we increase the mutation rate for the connection.

3. Each time the CDA* start working on a Team, it will make a random change in the order of the decision nodes that have been visited by the agents. So, the CDA* will start exploring a different agent each time it works (see Figure 6.8).

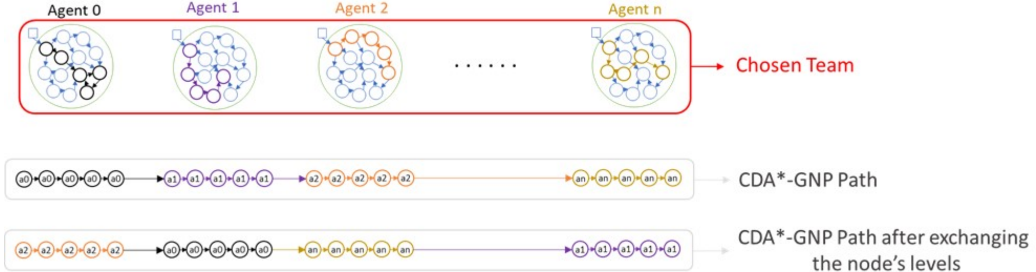


Figure 6.8: Exchanging the nodes' level

6.3.3 CDA*-GNP with Heterogeneous system to explore the Members (CDA*-GNP-HM)

The second way of using CDA* with heterogeneous is to explore different members of a team from the populations. Pseudocode 14 and Figure 6.9 explain the technique used in this method. Firstly, the algorithm starts with randomly initializing the first population for each agent. For each generation, the individuals in these populations are evaluated by making each one of them join a team and get its fitness value as explained before. The Team with the highest fitness is chosen to be the Best-Team. CDA*-GNP-Members( ) is applied to the populations for each generation, as explained below. If the explored Team from CDA*-GNP-Members has a greater fitness value than the Best-Team, it will be passed to the next generation as the Best-Team. After

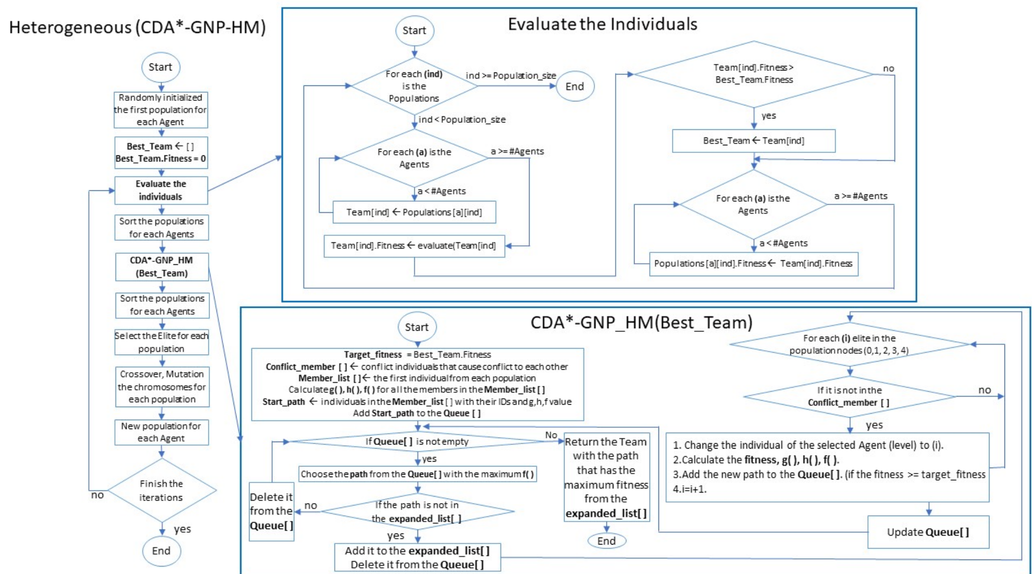


Figure 6.9: CDA*-GNP with Heterogeneous system to explore the Members (CDA*-GNP-HM)

that, the algorithm applies (selection, crossover, and mutation) operations to generate new individuals for each population to be passed to the next generation.

Pseudocode 15 and Figure 6.9 show in detail the technique that CDA*-GNP used to explore the members. It starts with setting the Target-fitness with the Best-Team fitness. Then, it detects all the members that cause a conflict with another member. In this experiment, a member can cause a conflict with another member if it stops its way when pushing a tile, as in Figure 6.10. Figuring out the conflict between the members happens when the Team is evaluated. The Member-list in the Heterogeneous system contains the first individual from each population. After that, the algorithm calculates g( ), h( ), and f( ) for each member in the Member-list. The g( ) is the fitness value, and h( ) is (the number of members in the path – level). In this algorithm, the h value is weighted by 10 as the number of members in the path is 30. The f( ) is (g( ) – h( )). The start path is added to the Queue. Then, the algorithm explores the paths in the Queue and adds them to the Expanded-list until the Queue is empty or the algorithm reaches 150 different explorations. The algorithm explores the first five members from each population (elite) and changes them on the Team to find the best combination of team members (see Figure 6.11). Finally, the algorithm chooses the path with the maximum fitness to be the Best-Team.

Algorithm 14 CDA*-GNP with Heterogeneous system to explore the Members (CDA*-GNP-HM)

```
1:  for all the Agents in the system do
2:      Populations[Agent] ← Randomly initialize the first population ( )
3:  end for
4:  Best-Team ← []
5:  while (solution is not found) do
6:      for each ind in the Populations do
7:          for each a in the Agents do
8:              Team[ind] ← Populations[a][ind]
9:          end for
10:         Team[ind].Fitness ← evaluate(Team[ind])
11:         if (Team[ind].Fitness > Best-Team.Fitness) then
12:             Best-Team ← Team[ind]
13:         end if
14:         for each a in the Agents do
15:             Populations[a][ind].Fitness ← Team[ind].Fitness
16:         end for
17:     end for
18:     for each a in the Agents do
19:         Sort(Populations[a])
20:     end for
21:     New-Team ← CDA*-GNP-Members(Populations)
22:     if (New-Team.Fitness > Best-Team.Fitness) then
23:         Best-Team ← New-Team
24:     end if
25:     for each a in the Agents do
26:         Sort(Populations[a])
27:         Selection (Populations[a])
28:         Crossover (Populations[a])
29:         Mutation (Populations[a])
30:         Populations[a] ← Next Generation
31:     end for
32: end while
```

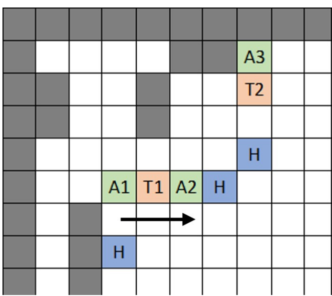


Figure 6.10: Conflict between members. A1 try to push T1 into the hole, while A2 is in the way between them, so A2 causes a conflict with A1

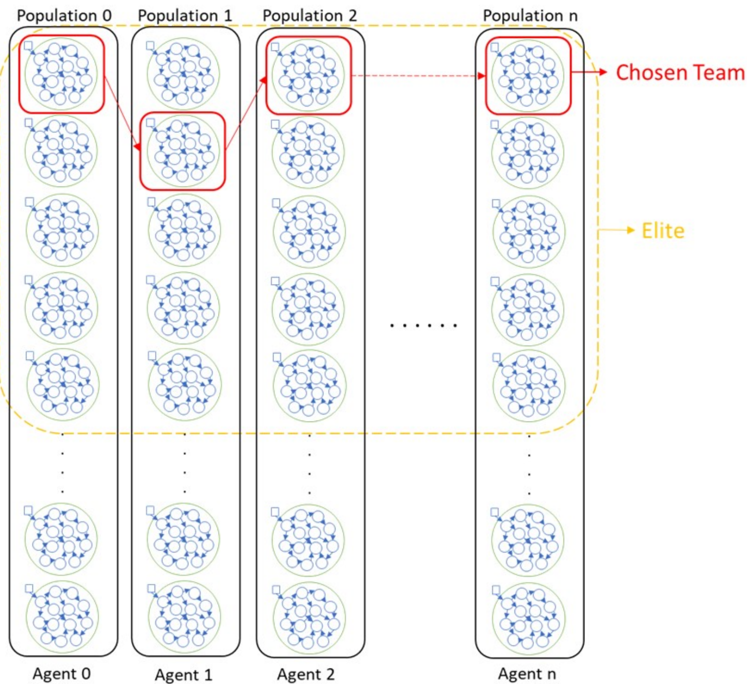


Figure 6.11: Exploring Members with CDA*-GNP in Heterogeneous system

Algorithm 15 CDA*-GNP-Members(Populations)

```
1:  Target-fitness ← Best-Team.Fitness
2:  Conflict-Members[ ]← conflicts individuals that cause conflict to each other
3:  Members-List[ ]← The first individual from every populations
4:  Calculate g( ), h( ), f( ) for all the members in the Members-List[ ]
5:  Start-path ← individuals in the Members-List[ ] with their IDs and g,h,f value.
6:  Add start-path to the Queue[ ]
7:  while (Queue[ ] is not empty) do
8:      Choose the path from the Queue[ ] with the maximum f( ).
9:      if the path is not in the Expanded-list[ ] then
10:         Add the path to the expanded-list[ ]
11:         Delete the path from the Queue[ ]
12:         for each (i) elite in the Populations (0,1, 2, 3, 4) do
13:             if it is not in the Conflict-Members[ ] then
14:                 Change the individual of the selected Agent (level) to (i).
15:                 Calculate the fitness, g( ), h( ), f( ).
16:                 Add it to Queue[ ] if the fitness >= target_fitness.
17:                 i=i+1.
18:                 Update Queue[ ]
19:             end if
20:         end for
21:     else
22:         Delete the path from the Queue[ ]
23:     end if
24: end while
25: Return the Team with the path that has the maximum fitness from the Expanded-
    list[ ]
```

6.4 Testing and Analysis

This section shows the proposed algorithm's performance results in two different experiments. The first one is when using a fully heterogeneous system with a different population for each agent. The second one is when using a partially heterogeneous system where each agent with the same skill set shares the same population. A full explanation of each one is discussed below.

6.4.1 Fully Heterogeneous system

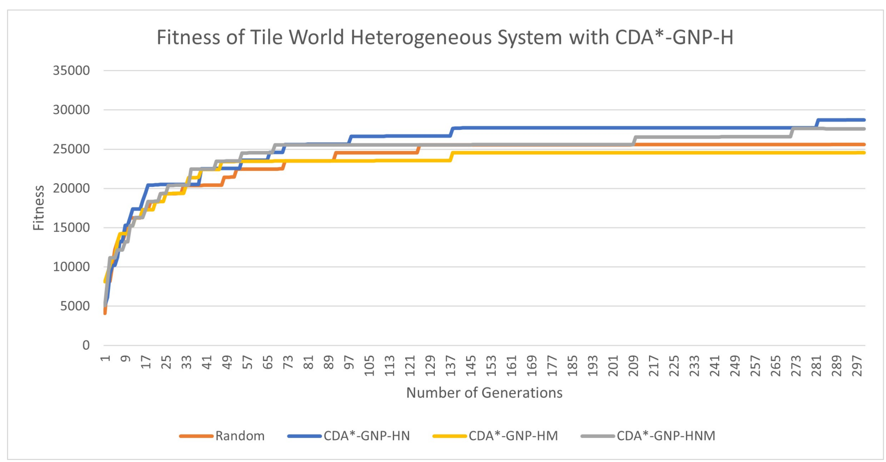


Figure 6.12: Fitness of the CDA*-GNP-H in Fully Heterogeneous system

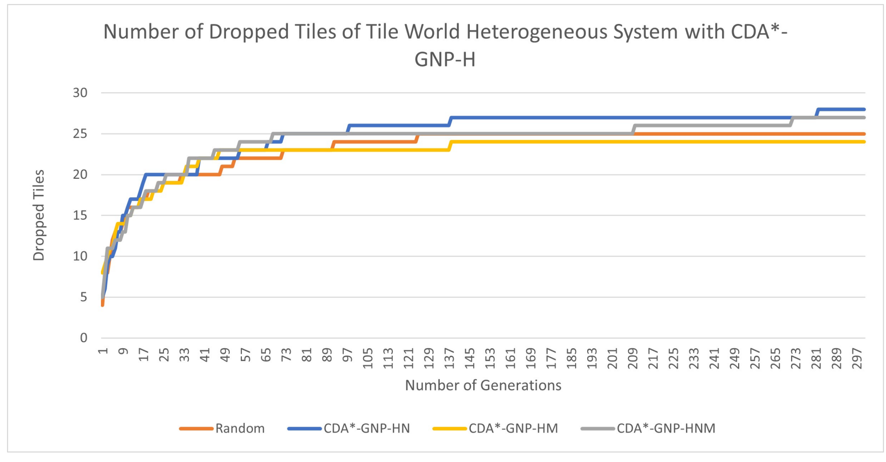


Figure 6.13: Number of dropped Tiles of the CDA*-GNP-H in Fully Heterogeneous system

Figures (6.12 and 6.13) show the fitness values and the number of dropped tiles for the Fully Heterogeneous Tile World Problem when using a different population for each Agent. Four different experiments were conducted: The randomly heterogeneous, the

Table 6.2: Comparing the Algorithms' results on Fully Heterogeneous Tile World Problem.

Algorithm	Mean of Fitness	Rank	P-value	Min of Fitness	Max of Fitness	Best training results Best Chromosome	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
Random Heterogeneous	23782	3	1.54E+10	4084	25582	25/30 S1 2/30 S2 2/30 S3	135450	300
CDA*-GNP-HN	25675	1	-	5081	28744	27/30 S1 1/30 S2 2/30 S3	62550	138
CDA*-GNP-HM	23210	4	1.89E+18	8144	24567	24/30 S1 2/30 S2 2/30 S3	113400	251
CDA*-GNP-HNM	24717	2	1.31E-03	5102	27626	27/30 S1 2/30 S2 3/30 S3	135000	299

CDA*-GNP-HN when exploring only the nodes, the CDA*-GNP-HM when exploring only the team members, and the CDA*-GNP-HNM when exploring both the nodes and team members. The experiments were run for 300 generations. In every generation for all the experiments, the maximum number of evaluations was 450, where 300 evaluations were performed to evaluate the chromosomes, and 150 evaluations for the (random, exploring the nodes, and exploring the members). In the fourth experiment, where the algorithm explored the nodes and members, the odd generations explored the nodes, and the evens explored the members. Table 6.2 compares the results of the four experiments. The algorithm that received the first rank was CDA*-GNP-HN, which explores the nodes only. It was able to reach 28744 fitness value in the training phase, but the testing results were not successful as the best Team was able to only push (1/30) and (2/30) for the Testing Set (2) and Testing Set (3) respectively. This negative testing result is due to two main reasons. The first one is the large search space, as there are 30 individuals in the Team, and each one of them has 120 nodes with all the probabilities of their functions and connections, so it takes a very long time to explore all the possible permutations. The second reason is that each individual in the Team was used by only one agent, and that will not give enough chance for each node in the graph to be explored as when the individual is used by many agents. These two main reasons were clear in the testing results for the four experiments. The second-ranked algorithm was CDA*-GNP-HNM, with a maximum fitness of 27626. The best Team appear in generation number 299 after 135000 number of evaluations with (27/30), (2/30), and (3/30) dropped tiles for the Training set (1), Testing set (2), and Testing set (3), respectively. The third-ranked algorithms were Randomly heterogeneous with a p-value of 1.54E+10 compared with the first-ranked experiment. It was able to get (25/30), (2/30), and (2/30) in the Training Set (1), Testing Set (2), and Testing Set (3), respectively, in generation number 300 after the 135450 number of evaluations. In

the end, the CDA*-GNP-HM was not able to improve the algorithm.

6.4.2 Partly Heterogeneous system

In the partly heterogeneous system experiments, there are two populations, one for the normal agents (Agent)and the other for the strong agents (SAgnet). Each population has 300 individuals, and the same techniques, parameters, and chromosome structure as the fully heterogeneous system except the crossover and mutation rate are (Pc = 0.01, Pm = 0.01, and Pm1 = 0.01).

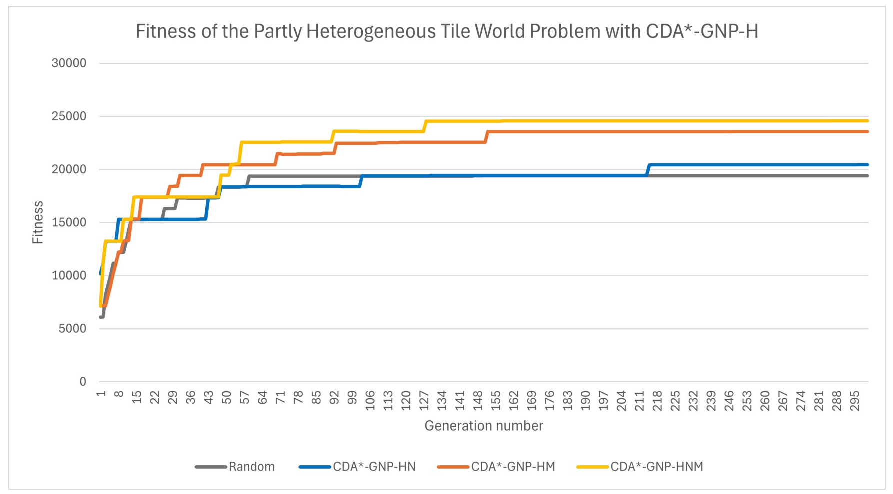


Figure 6.14: Fitness of the CDA*-GNP-H in Partly Heterogeneous system

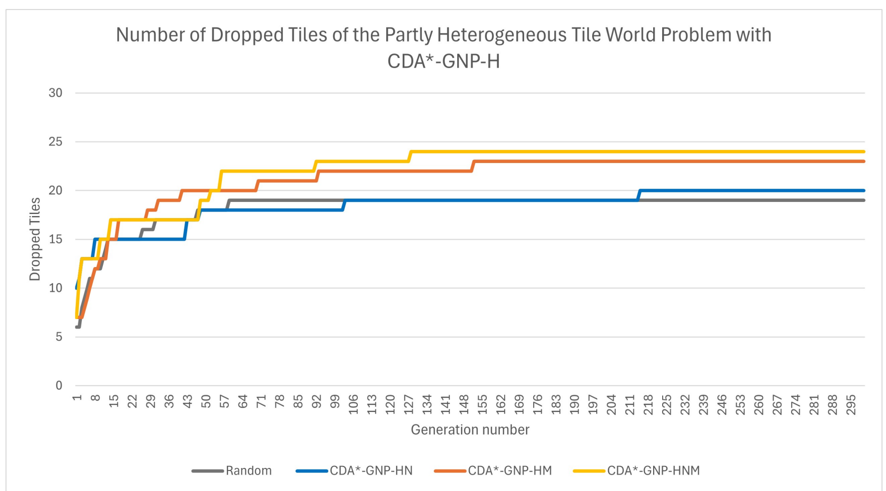


Figure 6.15: Number of dropped Tiles of the CDA*-GNP-H in Partly Heterogeneous system

Figures (6.14 and 6.15) show the fitness value and the number of dropped tiles for the Partly Heterogeneous Tile World Problem when using one population for the agents which share the same behaviour. Four different experiments were applied: The random

Table 6.3: Comparing the Algorithms' results on Partly Heterogeneous Tile World Problem.

Algorithm	Mean of Fitness	Rank	P-value	Min of Fitness	Max of Fitness	Best training results Best Chromosome	Number evaluation until get the best Chromosome	Number generation for the best Chromosome
Random Heterogeneous	18649	4	9.62E-66	6096	19393	19/30 S1 6/30 S2 5/30 S3	27000	59
CDA*-GNP-HN	18845	3	2.30E-63	10193	20436	20/30 S1 2/30 S2 3/30 S3	119250	264
CDA*-GNP-HM	21846	2	0.000145	7154	23582	23/30 S1 5/30 S2 7/30 S3	75150	166
CDA*-GNP-HNM	22796	1	-	7134	24588	24/30 S1 11/30 S2 9/30 S3	68850	152

heterogeneously, the CDA*-GNP-HN when exploring only the nodes, the CDA*-GNP-HM when exploring only the members, and the CDA*-GNP-NM when exploring the nodes and members. The experiments were run for 300 generations. In every generation for all the experiments, the maximum number of evaluations was 450, where 300 evaluations to evaluate the chromosomes and 150 evaluations for the (random, exploring the nodes, and exploring the members). In the fourth experiment, where the algorithm explored the nodes and members, the odd generations explored the nodes, and the evens explored the members.

Table 6.3 compares the results of the four experiments. The algorithm that received the first rank was CDA*-GNP-HNM, which explores the nodes and members. It was able to reach 24588 fitness value in the training phase. On the testing results, the best Team was able to push (24/30), (11/30), and (9/30) for the Training Set (1), Testing Set (2), and Testing Set (3) respectively. The second-ranked algorithm was CDA*-GNP-HM, with a maximum fitness of 23582. The best Team appears in generation number 166 after 75150 number of evaluations with (23/30), (5/30), and (7/30) dropped tiles for the Training set (1), Testing set (2), and Testing set (3), respectively. The third-ranked algorithms were CDA*-GNP-HN with a p-value of 2.30E-63 compared with the first-ranked experiment. It was able to get (20/30), (2/30), and (3/30) in the Training Set (1), Testing Set (2), and Testing Set (3), respectively, in generation number 264 after the 119250 number of evaluations. In the end, the Random Heterogeneous was able to reach 19393 fitness value for the training and (19/30), (6/30), and (5/30) dropped tiles in the Training Set (1), Testing Set (2), and Testing Set (3), respectively.

The smaller search space for the Partly Heterogeneous system when implementing the CDA*-GNP-HNM was a reason for improving the testing results particularly. As many agents work on one chromosome, that leads to incorporating many rules and covering many situations. However, the problem was not completely solved. There are

two ways that could potentially be used to increase the performance. The first one is to increase the chromosome size, the current size is 120 nodes, and this size could not cover all the available rules and situations. Increasing the chromosome size could give the algorithm a chance to incorporate more rules and situations. The second way is to divide the agents that share the same actions into many populations instead of using only one population. By dividing and merging the populations and the agent, the algorithm could divide the rules into many individuals, each one is used by the appropriate agents. These two methods are examples of future work.

6.5 Summary

In this chapter, we applied the CDA*-GNP algorithm to the partially and fully Heterogeneous systems. Four experiments were implemented: Random, CDA*-GNP-HN, CDA*-GNP-HM, and CDA*-GNP-HNM. The proposed algorithm achieved higher results when applied to the Partially heterogeneous system than the fully heterogeneous one.

Chapter 7

Summary and Conclusion

7.1 Introduction

This chapter summarizes and gives a brief description of the following algorithms: GA, GP, GNP, GNP-RL, DGNP-RL, VSGNP-RL, GNP-RL-CC-OS-TP, GNP-CC-OS-TP, PCK-GNP, CDA*-GNP, and CDA*-GNP-H. Then, it compares the algorithm parameters and results when applying them on different testbeds: Tile World Problem, Tile World Problem with Heavy Tiles, Prey and Predator Problem, and Heterogenous Tile World Problem with (Heavy Tiles and Strong Agents). Additionally, this chapter shows recorded videos for the best individuals in each algorithm when applying them to different problems. Finally, it gives some ideas about how these algorithms can be used in future work.

7.2 The Algorithms Summary

7.2.1 (Review) Genetic Algorithms (GA)

A genetic algorithm is a method that imitates genetic evolution in living organisms [1]. It employs the chromosome structure to represent the solutions. These chromosomes are improved and refined through evolutionary operations, generation by generation until they find the optimal solution. GA has a string structure, and when it comes to complex issues that require complex roles, it becomes more challenging to represent the solution [30]. To address this issue, a new algorithm called Genetic Programming (GP) was introduced.

7.2.2 (Review) Genetic Programming (GP)

A genetic programming algorithm was used to develop a non-linear chromosome structure that can solve decision-making problems. The structure is based on a tree with a

root starting point, if-then nodes, and terminal nodes. One major drawback of genetic programming is that the tree can become too large due to evolutionary operations. This issue has been addressed through the development of an algorithm called the Genetic Networking Programming Algorithm (GNP).

7.2.3 (Review) Genetic Network Programming (GNP)

In 2000, a new algorithm called Genetic Networking Programming (GNP) was introduced by Katagiri et al. to address the issue of generating oversized trees in GP [5]. The algorithm uses a networking graph structure to represent a chromosome, which enables multi-use nodes without end nodes to solve complex problems. The graph contains three types of nodes, namely, start, judgment, and processing nodes, that are connected using directed connections. Unlike a tree structure, each graph has only one start node and no terminal node. A node can be visited more than once and by multiple agents. The benefit of this structure is that it can cover more rules with a smaller size structure. However, for more complex problems, more probabilities are needed to cover all the rules. This requires an increase in the number of nodes in the graph, which in turn increases the chromosome size.

7.2.4 (Review) Genetic Networking Programming with Reinforcement Learning (GNP-RL)

In 2007, Mabu et al. [6] proposed a new approach called Genetic Networking Programming with Reinforcement Learning (GNP-RL). This approach combined learning with evolution by adding sub-nodes for each node in the graph, which allowed the reinforcement learning to detect the best sub-node after exploring and exploiting the graph. The good actions are reinforced with positive rewards, which increases the probability of their execution when the agent revisits the same node. This approach solves complex problems while keeping the chromosome structure simple. The GNP-RL algorithm is powerful and can solve many problems that its predecessors couldn't. However, it does have some limitations. For instance, when any agent visits a node within the same simulation on the same chromosome, it can use a different function type each time. This means that within the same evaluation, the node could be utilized with a different function type, causing the testing result for the chromosome to differ from the training result in the same training environment. This happens because of the randomization involved in the exploration of Reinforcement Learning. The GNP-RL algorithm may suffer from overfitting [17]. Additionally, it is a non-systematic algorithm. As a result, it may lose the optimal solution by incorrectly choosing the elite based on the training results that used exploration and exploitation. As the GNP-RL algorithm aims to increase the number of possible solutions to a problem, it also results in an increase in the

size of the graph. The algorithm needs to save all the nodes with their respective sub-nodes, including their functions and connections. This, in turn, requires more memory space. Although the GNP-RL algorithm outperformed the GNP algorithm [6], there is still room for improvement in solving the Tile World Problem.

7.2.5 (Review) Distributed Genetic Network Programming (DGNP-RL) and Variable-sized Genetic Network Programming (VSGNP-RL)

The method of dividing structure with the GNP-RL was initially suggested by Yang et al. in their study [16]. The method involves breaking down the problem into smaller tasks to simplify the chromosome structure and enhance the algorithm's functionality. In 2014[9], Mabu et al. presented a new version of the Distributed Genetic Networking Program with Reinforcement Learning (GNP-RL), called the Variable-Sized Genetic Networking Program with Reinforcement Learning (VSGNP-RL). In this version, the distributed structure can consist of sub-programs that have different sizes. Each sub-program may contain a different number of nodes (genes). VSGNP-RL is an improvement over GNP-RL, as it divides the chromosome into sub-programs to simplify problem-solving. The variable-sized structure can learn general action rules in graph structures, enabling it to adapt to various situations. Determining the parameters for evolution is a sensitive task that must be executed with care to avoid significant changes to the program structure, which may remove essential building blocks. However, the program's flexible size can create difficulties for this process. Moreover, the algorithm for learning or evolving the parameters still remains an unresolved issue.

7.2.6 (Review) Genetic Network Programming with Reinforcement Learning, Constraint Conformance, Optimal Search, and Task Prioritization (GNP-RL-CC-OS-TP)

In our previous work [17], we introduced three novel techniques to the GNP-RL algorithm. These techniques are constraint conformance, optimal search, and task prioritization. The use of these techniques has resulted in a significant improvement in the performance of the GNP-RL algorithm in multi-agent systems. By using constraint conformance, the agent can learn to avoid unwanted situations when receiving punishment signals. Additionally, optimal search helps the agent to find accurate paths using the A* algorithm. Finally, task prioritization arranges tasks for each agent using a scalar reward and punishment scheme.

Using A* as a heuristic function in the graph increases feedback accuracy from the environment, leading to improved agent decision-making. When an agent makes a wrong decision, updating the Q-value using punishment and directing the agent to

make a different decision can quickly improve the agent's performance and prevent it from making the same mistake again. When updating the Q-value, selecting scalable rewards and punishments encourages agents to prioritize tasks.

Although using A* as a heuristic function can enhance performance, it can be time-consuming for larger environments and more objects. The constraint conformance technique is effective in detecting conflicts and preventing agents from taking actions that lead to conflicts. However, it does not solve the function node in the graph that caused the conflict. This algorithm outperformed GNP-RL; however, it has the same limitations as GNP-RL since it is an extension of it. Further improvements are needed to tackle complex problems like the Tile World problem.

7.2.7 (Proposed) Genetic Networking Programming with Constraint Conformance – Optimal Search – Task Prioritization (GNP-CC-OS-TP)

Because applying the three techniques (Constraint conformance, Optimal search, and task prioritization) on the GNP-RL was successful, in this thesis, we decided to apply these three techniques to the GNP without RL. The same way used for the (optimal search and Constraint conformance) was used with GNP. For (Task prioritization), scalable rewards and punishments were used when updating the Q-value in the GNP-RL, whereas, with GNP, scalable rewards and punishments were used when calculating (Distance) in the fitness function. The results of applying the GNP-CC-OS-TP on the Tile World Problem beat the one with GNP. We chose this algorithm as a base for improving our proposed algorithms in this thesis.

7.2.8 (Proposed) Using Private Conflict Kernels with Genetic Network Programming (PCK-GNP)

We have incorporated a systematic strategy to enhance the efficiency and accuracy of the GNP algorithm. This approach involves identifying the subsets of nodes that cause conflicts and compiling them into conflict kernels. These kernels are then used to refine the evolutionary mechanisms of the algorithm and prevent the occurrence of conflict-generating structures in the graph in future generations of chromosomes. We utilized a private conflict kernel for each chromosome, which contains all sub-structures with conflicts. These kernels will be employed to enhance Selection, Mutation, and Crossover. In genetic algorithms, the selection of the best individuals is typically determined by their fitness value. However, this approach also considers the number of conflict-generating structures for each individual. In the crossover, the two parents are chosen based on their private kernels' diversity. The crossover point occurs on one of the nodes of those conflict-generating substructures, resulting in a new combination sub-structure that

can potentially resolve the conflict. When it comes to mutation, there are two ways to utilize the private kernel. The first one involves selecting the mutation gene (node) from the conflict-generating structures. The second way involves adopting an incremental repair technique to be applied to these conflict-generating structures.

When adding private-conflict-kernels to GNP, 100% accuracy was achieved during training and testing in some experiments related to the Tile World problem and Heavy Tile World problem, while using a single sub-program. However, there are three limitations to this approach. Firstly, performing the algorithm can be complex. Secondly, the extracted conflict is related to the environment, which means it is not a generalization technique. Thirdly, due to randomization, the algorithm was not successful in all the experiments. Tables 7.1, 7.2, and 7.3 compare the different combinations of the PCK-GNP algorithm with their parameters and results.

Table 7.1: Performance comparison of the different variants of PCK-GNP on the Tile World Problem

Tile World Problem							
Parameters	**Selection PCK-GNP (S)**	**Crossover PCK-GNP (C)**	**Mutation PCK-GNP (M)**	**Selection & Mutation PCK-GNP (S&M)**	**Selection & Crossover PCK-GNP (S&C)**	**Crossover & Mutation PCK-GNP (C&M)**	**Selection & Crossover & Mutation PCK-GNP (S&C&M)**
By	Us	Us	Us	Us	Us	Us	Us
Individuals	300						
Nodes	120						
Node Type	8 Judgment Nodes & 4 Processing Nodes						
Number of Processing Nodes	40						
Number of Judgment Nodes	80						
Using A* as Judgment Node	Yes						
Number of sub-nodes	1						
Number of sub-programs	1						
Crossover	120						
Mutation	175						
Elite	5						
Crossover Rate	Pc: 0.1						
Mutation Rate	Pm1: 0.1 - Pm2: 0.01 - CKm:0.1						
Tournament size	7						
Fitness Function	Equation (2.4) each with different calculation of Ddistance						
Agent Steps	60						
Delay Time	1 $\rightarrow$ Judgment - 5 $\rightarrow$ Processing						
Q-value	-						
Best Testing results set (1)	27/30 (90%)	30/30 (100%)	30/30 (100%)	30/30 (100%)	28/30 (93%)	30/30 (100%)	28/30 (93%)
Best Testing results set (2)	9/30 (30%)	15/30 (50%)	23/30 (77%)	13/30 (43%)	12/30 (40%)	19/30 (63%)	4/30 (13%)
Best Testing results set (3)	21/30 (70%)	24/30 (80%)	30/30 (100%)	25/30 (83%)	13/30 (43%)	30/30 (100%)	14/30 (47%)

Table 7.2: Performance comparison of the different variants of GNP with one sub-program (and parameter settings) on the Heavy Tile World

Tile World Problem with Heavy Tile with one sub-program							
Parameters	**Selection PCK-GNP (S)**	**Crossover PCK-GNP (C)**	**Mutation PCK-GNP (M)**	**Selection & Mutation PCK-GNP (S&M)**	**Selection & Crossover PCK-GNP (S&C)**	**Crossover & Mutation PCK-GNP (C&M)**	**Selection & Crossover & Mutation PCK-GNP (S&C&M)**
By	Us	Us	Us	Us	Us	Us	Us
Individuals	300						
Nodes	120						
Node Type	8 Judgment Nodes & 4 Processing Nodes						
Number of Processing Nodes	40						
Number of Judgment Nodes	80						
Using A* as Judgment Node	Yes						
Number of sub-nodes	1						
Number of sub-programs	1						
Crossover	120						
Mutation	175						
Elite	5						
Crossover Rate	Pc: 0.1						
Mutation Rate	Pm1: 0.1 - Pm2: 0.01 - CKm:0.1						
Tournament size	7						
Fitness Function	Equation (2.4) each with different calculation of Ddistance						
Agent Steps	100						
Delay Time	1 $\rightarrow$ Judgment - 5 $\rightarrow$ Processing						
Q-value	-						
Best Testing results set (1)	30/30 (100%)	30/30 (100%)	30/30 (100%)	19/30 (63%)	28/30 (93%)	29/30 (97%)	21/30 (70%)
Best Testing results set (2)	9/30 (30%)	10/30 (33%)	13/30 (43%)	5/30 (17%)	9/30 (30%)	14/30 (47%)	7/30 (23%)
Best Testing results set (3)	14/30 (47%)	26/30 (87%)	30/30 (100%)	13/30 (43%)	16/30 (53%)	27/30 (90%)	13/30 (43%)

Table 7.3: Performance comparison of the different variants of VSGNP with two sub-programs (and parameter settings) on the Heavy Tile World

Tile World Problem with Heavy Tile with two sub-programs							
Parameters	**Selection PCK-GNP (S)**	**Crossover PCK-GNP (C)**	**Mutation PCK-GNP (M)**	**Selection & Mutation PCK-GNP (S&M)**	**Selection & Crossover PCK-GNP (S&C)**	**Crossover & Mutation PCK-GNP (C&M)**	**Selection & Crossover & Mutation PCK-GNP (S&C&M)**
By	Us	Us	Us	Us	Us	Us	Us
Individuals	300						
Nodes	120						
Node Type	8 Judgment Nodes & 4 Processing Nodes						
Number of Processing Nodes	40						
Number of Judgment Nodes	80						
Using A* as Judgment Node	Yes						
Number of sub-nodes	1						
Number of sub-programs	2 Subprograms: SubProgram1 $\rightarrow$ 72 Nodes - SubProgram2 $\rightarrow$ 48 Nodes						
Crossover	120						
Mutation	175						
Elite	5						
Crossover Rate	Pc: 0.1						
Mutation Rate	Pm1: 0.1 - Pm2: 0.01 - CKm:0.1						
Tournament size	7						
Fitness Function	Equation (2.4) each with different calculation of Ddistance						
Agent Steps	100						
Delay Time	1 $\rightarrow$ Judgment - 5 $\rightarrow$ Processing						
Q-value	-						
Best Testing results set (1)	30/30 (100%)	30/30 (100%)	30/30 (100%)	15/30 (50%)	27/30 (90%)	30/30 (100%)	19/30 (63%)
Best Testing results set (2)	19/30 (63%)	18/30 (60%)	8/30 (27%)	1/30 (3%)	6/30 (20%)	12/30 (40%)	3/30 (10%)
Best Testing results set (3)	27/30 (90%)	24/30 (80%)	23/30 (77%)	8/30 (27%)	18/30 (60%)	29/30 (97%)	14/30 (47%)

7.2.9 (Proposed) Using Conflict Directed A* inside GNP Graph (CDA*-GNP)

We have improved the GNP technique by incorporating the optimal search. This enables us to identify the best combination of processing nodes within the graph to achieve optimal results. To enhance our approach, we integrated Conflict Directed A* with GNP. This strategy reduces exploration time by eliminating conflict-generating paths from consideration. The CDA*-GNP explores the potential functions of nodes within a graph while excluding any that may cause conflicts. This approach has proven to be highly efficient in providing superior performance results in less time than other algorithms. This has been demonstrated when applying the CDA*-GNP algorithm to three problems: the Tile World Problem, the Tile World Problem with Heavy Tile, and the Prey and Predator Problem. Tables 7.4, 7.5, 7.6, and 7.7 compare the parameters and results for the algorithms (GNP, GNP-CC-OS-TP, GNP-RL, GNP-RL-CC-OS-TP, PCK-GNP, and CDA*-GNP). The chosen version of the PCK-GNP is the one that received the best result when applying it to the presented problem as below.

Table 7.4: Performance comparison of the different variants of the GNP algorithm (with parameter settings) on the Tile World Problem

Tile World Problem							
Parameters	**GNP (Review)**	**GNP-CC-OS-TP**	**GNP-RL (Review)**	**GNP with Rule accumulation (Review)**	**GNP-RL-CC-OS-TP (Review)**	**PCK-GNP (C&M)**	**CDA*-GNP**
By	Xianneng et al. [6]	Us	Mabu et al. [6]	Li et al. [10]	Alshehri et al. [17] [14]	Us	Us
Individuals	300						
Nodes	120						
Node Type	8 Judgment Nodes & 4 Processing Nodes						
Number of Processing Nodes	40						
Number of Judgment Nodes	80						
Using A* as Judgment Node	No	Yes	No	No	Yes	Yes	Yes
Number of sub-nodes	1	1	4	4	4	1	1
Number of sub-programs	1						
Crossover	120						
Mutation	175						
Elite	5						
Crossover Rate	Pc: 0.1						
Mutation Rate	Pm1: 0.1 - Pm2: 0.01						
Tournament size	7						
Fitness Function	Equation (2.4)	Equation (2.4)	Equation (2.4)	Equation (2.1)	Equation (2.4)	Equation (2.4)	Equation (2.4)
Agent Steps	60						
Delay Time	1 → Judgment - 5 → Processing						
Q-value	-	-	Equation (2.2)	Equation (2.2)	Equation (2.2)	-	-
Best Testing results set (1)	21/30 (70%)	24/30 (80%)	17/30 (57%)	28/30 (93%)	30/30 (100%)	30/30 (100%)	30/30 (100%)
Best Testing results set (2)	2/30 (7%)	6/30 (20%)	2/30 (7%)	19/30 (63%)	22/30 (73%)	19/30 (63%)	29/30 (97%)
Best Testing results set (3)	8/30 (27%)	15/30 (50%)	13/30 (43%)	16/30 (53%)	25/30 (83%)	30/30 (100%)	30/30 (100%)

Table 7.5: Performance comparison of the different variants of the GNP algorithm (with one subprogram and parameter settings) on the Heavy Tile World

<table>
<tr><th colspan="7">Tile World Problem with Heavy Tile with one sub-program</th></tr>
<tr><th>Parameters</th><th>GNP</th><th>GNP-CC-OS-TP</th><th>GNP-RL</th><th>GNP-RL-CC-OS-TP</th><th>PCK-GNP (C)</th><th>CDA*-GNP</th></tr>
<tr><td>By</td><td>Us</td><td>Us</td><td>Us</td><td>Us</td><td>Us</td><td>Us</td></tr>
<tr><td>Individuals</td><td colspan="6">300</td></tr>
<tr><td>Nodes</td><td colspan="6">120</td></tr>
<tr><td>Node Type</td><td colspan="6">8 Judgment Nodes & 4 Processing Nodes</td></tr>
<tr><td>Number of Processing Nodes</td><td colspan="6">40</td></tr>
<tr><td>Number of Judgment Nodes</td><td colspan="6">80</td></tr>
<tr><td>Using A* as Judgment Node</td><td>No</td><td>Yes</td><td>No</td><td>Yes</td><td>Yes</td><td>Yes</td></tr>
<tr><td>Number of sub-nodes</td><td>1</td><td>1</td><td>4</td><td>4</td><td>1</td><td>1</td></tr>
<tr><td>Number of sub-programs</td><td colspan="6">1</td></tr>
<tr><td>Crossover</td><td colspan="6">120</td></tr>
<tr><td>Mutation</td><td colspan="6">175</td></tr>
<tr><td>Elite</td><td colspan="6">5</td></tr>
<tr><td>Crossover Rate</td><td colspan="6">Pc: 0.1</td></tr>
<tr><td>Mutation Rate</td><td colspan="6">Pm1: 0.1 - Pm2: 0.01</td></tr>
<tr><td>Tournament size</td><td colspan="6">7</td></tr>
<tr><td>Fitness Function</td><td colspan="6">Equation (2.4) each with different calculation of Ddistance</td></tr>
<tr><td>Agent Steps</td><td colspan="6">100</td></tr>
<tr><td>Delay Time</td><td colspan="6">1 → Judgment - 5 → Processing</td></tr>
<tr><td>Q-value</td><td>-</td><td>-</td><td>Equation (2.2)</td><td>Equation (2.2)</td><td>-</td><td>-</td></tr>
<tr><td>Best Testing results set (1)</td><td>19/30 (63%)</td><td>24/30 (80%)</td><td>21/30 (70%)</td><td>30/30 (100%)</td><td>30/30 (100%)</td><td>30/30 (100%)</td></tr>
<tr><td>Best Testing results set (2)</td><td>2/30 (7%)</td><td>3/30 (10%)</td><td>7/30 (23%)</td><td>17/30 (57%)</td><td>10/30 (33%)</td><td>22/30 (73%)</td></tr>
<tr><td>Best Testing results set (3)</td><td>8/30 (27%)</td><td>8/30 (27%)</td><td>15/30 (50%)</td><td>30/30 (100%)</td><td>26/30 (87%)</td><td>30/30 (100%)</td></tr>
</table>

Table 7.6: Performance comparison of the different variants of the VSGNP algorithm (with two subprograms and parameter settings) on the Heavy Tile World

<table>
<tr><th colspan="7">Tile World Problem with Heavy Tile with two sub-programs</th></tr>
<tr><th>Parameters</th><th>GNP</th><th>GNP-CC-OS-TP</th><th>GNP-RL</th><th>GNP-RL-CC-OS-TP</th><th>PCK-GNP (C&M)</th><th>CDA*-GNP</th></tr>
<tr><td>By</td><td>Us</td><td>Us</td><td>Us</td><td>Us</td><td>Us</td><td>Us</td></tr>
<tr><td>Individuals</td><td colspan="6">300</td></tr>
<tr><td>Nodes</td><td colspan="6">120</td></tr>
<tr><td>Node Type</td><td colspan="6">8 Judgment Nodes & 4 Processing Nodes</td></tr>
<tr><td>Number of Processing Nodes</td><td colspan="6">40</td></tr>
<tr><td>Number of Judgment Nodes</td><td colspan="6">80</td></tr>
<tr><td>Using A* as Judgment Node</td><td>No</td><td>Yes</td><td>No</td><td>Yes</td><td>Yes</td><td>Yes</td></tr>
<tr><td>Number of sub-nodes</td><td>1</td><td>1</td><td>4</td><td>4</td><td>1</td><td>1</td></tr>
<tr><td>Number of sub-programs</td><td colspan="6">2 Subprograms: SubProgram1 → 72 Nodes - SubProgram2 → 48 Nodes</td></tr>
<tr><td>Crossover</td><td colspan="6">120</td></tr>
<tr><td>Mutation</td><td colspan="6">175</td></tr>
<tr><td>Elite</td><td colspan="6">5</td></tr>
<tr><td>Crossover Rate</td><td colspan="6">Pc: 0.1</td></tr>
<tr><td>Mutation Rate</td><td colspan="6">Pm1: 0.1 - Pm2: 0.01</td></tr>
<tr><td>Tournament size</td><td colspan="6">7</td></tr>
<tr><td>Fitness Function</td><td colspan="6">Equation (2.4) each with different calculation of Ddistance</td></tr>
<tr><td>Agent Steps</td><td colspan="6">100</td></tr>
<tr><td>Delay Time</td><td colspan="6">1 → Judgment - 5 → Processing</td></tr>
<tr><td>Q-value</td><td>-</td><td>-</td><td>Equation (2.2)</td><td>Equation (2.2)</td><td>-</td><td>-</td></tr>
<tr><td>Best Testing results set (1)</td><td>19/30 (63%)</td><td>21/30 (70%)</td><td>16/30 (53%)</td><td>30/30 (100%)</td><td>30/30 (100%)</td><td>30/30 (100%)</td></tr>
<tr><td>Best Testing results set (2)</td><td>2/30 (7%)</td><td>6/30 (20%)</td><td>7/30 (23%)</td><td>17/30 (57%)</td><td>12/30 (40%)</td><td>23/30 (77%)</td></tr>
<tr><td>Best Testing results set (3)</td><td>9/30 (30%)</td><td>13/30 (43%)</td><td>13/30 (43%)</td><td>30/30 (100%)</td><td>29/30 (97%)</td><td>30/30 (100%)</td></tr>
</table>

Table 7.7: Performance comparison of the different variants of the GNP algorithm (with their parameters) on the Prey and Predator Problem

Prey and Predator Problem					
Parameters	**GNP (Review)**	**GNP-CC-OS-TP**	**GNP-RL (Review)**	**GNP-RL-CC-OS-TP**	**CDA*-GNP**
By	Roshanzamir et al. [42]	Us	Roshanzamir et al. [42]	Us	Us
Individuals	50				
Nodes	120				
Node Type	5 Judgment Nodes & 4 Processing Nodes				
Number of Processing Nodes	40				
Number of Judgment Nodes	80				
Using A* as Judgment Node	No	Yes	No	Yes	Yes
Number of sub-nodes	1	1	4	4	1
Number of sub-programs	1				
Crossover	20				
Mutation	29				
Elite	1				
Crossover Rate	Pc: 0.1				
Mutation Rate	Pm1: 0.1 - Pm2: 0.01				
Tournament size	2				
Fitness Function	Equations (3.1) & (3.2)				
Agent Steps	60				
Delay Time	1 $\rightarrow$ Judgment - 5 $\rightarrow$ Processing				
Q-value	-	-	Equation (2.2)	Equation (2.2)	-
Best Testing results	831/900 27/30 (92%)	826/900 27/30 (92%)	782/900 26/30 (87%)	829/900 27/30 (92%)	847/900 28/30 (94%)

7.2.10 (Proposed) Using Conflict Directed A* with GNP in a Heterogeneous System (CDA*-GNP-H)

The process of evaluating the computational graph for teams in heterogeneous systems can be lengthy and complicated due to the large number of potential agent intelligence combinations to choose from. It is impossible to evaluate every possible combination of different agent types in each generation. Therefore, a combination of Conflict-Directed A* and GNP is used in heterogeneous systems to identify the best team agents. The CDA*-GNP has been utilized in two ways in a heterogeneous system. Firstly, CDA* is used to explore the nodes and determine the best function for each node, as it was used previously in a homogeneous system. Secondly, CDA* is employed to explore the population of individuals of each agent to identify the most suitable chromosome that represents each agent without conflicting with other agents in a team. The proposed algorithm was tested on two different heterogeneous systems, the partly and fully heterogeneous systems. The partly heterogeneous system was able to reach better results than the fully heterogeneous one. Tables 7.8 and 7.9 compare the parameters and results for the algorithms (Random heterogeneous, CDA*-GNP-HN, CDA*-GNP-HM, and CDA*-GNP-HNM) for the fully and partly heterogeneous systems.

Table 7.8: Performance comparison of the different variants of the CDA*-GNP-H algorithm on the Fully Heterogeneous Tile World

Fully Heterogeneous Tile World Problem				
Parameters	**Random**	**CDA*-GNP-HN**	**CDA*-GNP-HM**	**CDA*-GNP-HNM**
By	Us	Us	Us	Us
Populations	30, one population for each agent			
Individuals	300			
Nodes	120			
Node Type	8 Judgment Nodes & 4 Processing Nodes			
Number of Processing Nodes	40			
Number of Judgment Nodes	80			
Using A* as Judgment Node	Yes			
Number of sub-nodes	1			
Number of sub-programs	2 Subprograms: SubProgram1 → 72 Nodes - SubProgram2 → 48 Nodes			
Crossover	120			
Mutation	175			
Elite	5			
Crossover Rate	Pc: 0.001			
Mutation Rate	Pm1: 0.001 - Pm2: 0.005			
Tournament size	7			
Fitness Function	Equation (2.4) with the same calculation of Ddistance as in section 3.3.2			
Agent Steps	180			
Delay Time	1 → Judgment - 5 → Processing			
Q-value	-			
Best Testing results set (1)	25/30 (83%)	27/30 (90%)	24/30 (80%)	27/30 (90%)
Best Testing results set (2)	2/30 (6%)	1/30 (3%)	2/30 (6%)	2/30 (6%)
Best Testing results set (3)	2/30 (6%)	2/30 (6%)	2/30 (6%)	3/30 (10%)

Table 7.9: Performance comparison of the different variants of the CDA*-GNP-H algorithm on the Partly Heterogeneous Tile World

Partly Heterogeneous Tile World Problem				
Parameters	**Random**	**CDA*-GNP-HN**	**CDA*-GNP-HM**	**CDA*-GNP-HNM**
By	Us	Us	Us	Us
Populations	2, one population for 10 Strong agents, one population for 20 normal agents			
Individuals	300			
Nodes	120			
Node Type	8 Judgment Nodes & 4 Processing Nodes			
Number of Processing Nodes	40			
Number of Judgment Nodes	80			
Using A* as Judgment Node	Yes			
Number of sub-nodes	1			
Number of sub-programs	2 Subprograms: SubProgram1 → 72 Nodes - SubProgram2 → 48 Nodes			
Crossover	120			
Mutation	175			
Elite	5			
Crossover Rate	Pc: 0.01			
Mutation Rate	Pm1: 0.01 - Pm2: 0.01			
Tournament size	7			
Fitness Function	Equation (2.4) with the same calculation of Ddistance as in section 3.3.2			
Agent Steps	180			
Delay Time	1 → Judgment - 5 → Processing			
Q-value	-			
Best Testing results set (1)	19/30 (63%)	20/30 (67%)	23/30 (77%)	24/30 (80%)
Best Testing results set (2)	6/30 (20%)	2/30 (6%)	5/30 (17%)	11/30 (37%)
Best Testing results set (3)	5/30 (17%)	3/30 (10%)	7/30 (23%)	9/30 (30%)

7.3 Managing Genetic Diversity in the Population

1. PCK-GNP:

   (a) PCK-GNP (S) Selection: The selection process of the elite portion that is passed to the next generation is contingent upon the size of their Private-Conflict-Kernel. This methodology ensures the diversity of the succeeding generation, as not all individuals are evaluated only on the basis of their fitness value. Certain individuals are also ranked based on both their fitness value and Private-Conflict-Kernel size.

   (b) PCK-GNP (C) Crossover: The crossover operation selects parents based on the diversity of their Private-Conflict-Kernels in order to enhance population diversity through the crossover process.

   (c) PCK-GNP (M) Mutation: An incremental repair technique can work in the mutation when it comes to improving the overall algorithm. By applying it to a selected node within the Private-Conflict-Kernel, we can ensure that any changes made are truly useful and effective.

2. CDA*-GNP:

(a) We keep the mutation rate of 0.01 when modifying the node function, as the CDA* algorithm will only apply to one individual. Therefore, we ensure that we alter the node function for the remaining individuals to enhance diversity. The mutation rate of 0.01 guarantees minimal modifications in individuals since, in most cases, modifying only one function is sufficient to enhance their performance. Making too many changes can lose the improvement process and cause individuals to regress.

(b) The CDA* ensures the selection of the most appropriate function for the decision nodes. To modify the connections between the nodes, we have decided to increase the mutation rate to 0.1. Since the agents don't visit all nodes in the simulation, changing the connection on an unused node won't improve anything. Therefore, we increase the mutation rate for changing connections.

(c) CDA* is applied to only one individual per generation, apart from the first population, where it is applied to the best five individuals to increase diversity in future generations.

(d) Whenever the CDA* algorithm begins working on an individual, it will randomly change the order in which the decision nodes have been visited by the agents. This means that each time the CDA* is used, it will explore a different node.

(e) CDA* follows a specific process to select an individual to work on it. While it may not always pick the most qualified individual, it ensures that two points are verified. Firstly, CDA* checks if the best individual was explored in the previous generation and hasn't been altered by crossover and mutation. If that's the case, CDA* moves on to the next best individual and repeats the two-point check until the appropriate individual is found.

3. CDA*-GNP-H:

(a) When we change the node function for the Team of 30 agents, we keep the mutation rate at 0.001. This is because each agent has 120 nodes, which means that 3600 nodes are affecting teamwork. If we set a high mutation rate, too many nodes will be changed during the mutation process, breaking the good connections and losing the nodes that have already improved during the exploring phase. Therefore, we keep the mutation rate low to ensure that the changes are minimal and don't affect the overall performance of the team.

(b) To ensure that the best function is chosen for the decision nodes, we have set the mutation rate to 0.005 when changing the connections between the nodes. This is because not all nodes are utilized by the agents during the

simulation and changing the connection on an unused node will not result in any improvement. Therefore, we have increased the mutation rate for the connection.

(c) Each time the CDA* algorithm begins working on a team, it will randomly alter the order in which the decision nodes visited by the agents are explored. This means that the CDA* will explore a different agent each time it operates.

7.4 Best Chromosomes Extracted by the Proposed Algorithms

In order to get a better understanding of the intelligence evolved by the proposed algorithms, we recorded videos of the best chromosomes (from each algorithm) in action, on every benchmarking problem, in both training and testing environments. Each item in the list below provides a link to a YouTube video. :

1. PKC-GNP on Tile World Problem.
2. PCK-GNP on Heavy Tile World Problem using one sub-program.
3. PCK-GNP on Heavy Tile World Problem using two sub-programs.
4. CDA*-GNP on Tile World Problem.
5. CDA*-GNP on Heavy Tile World Problem using one sub-program.
6. CDA*-GNP on Heavy Tile World Problem using two sub-programs.
7. CDA*-GNP on Prey and Predator Problem.
8. CDA*-GNP-HNM on Fully Heterogenous Tile World Problem.
9. CDA*-GNP-HNM on Partly Heterogenous Tile World Problem.

Dissection of the best chromosomes

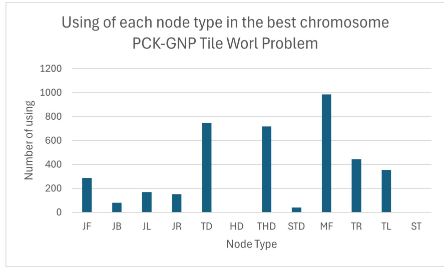
In this section, we dissect the best chromosomes from each of the proposed algorithms, to show the number of times each node type was utilised to solve the different problems. This would helps us understand further how the PCK-GNP and CDA*-GNP algorithms differ from each other when solving the same set of problems.

Tile World Problem: As can be seen in Fig. 7.1, the most used node types from the best chromosomes when solving the Tile World Problem using PCK-GNP and CDA*-GNP are the following: (TD) the direction to the nearest tile, (THD) the direction from the nearest tile to the nearest hole, and (MF) move forward nodes. On

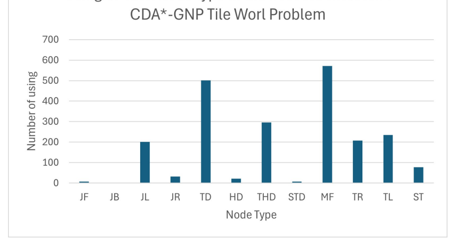
the other hand, the following node types were almost not utilised: (HD) the direction to the nearest hole and (STD) the direction to the second nearest tile. Next, examining the Judgment nodes, it can be observed that (THD), (TD), and one of the (JF, JB, JL, JR) nodes were enough for the algorithm to inquire information from the environment and make judgements; other judgment nodes were almost not used in the Tile World Problem.

Heavy Tile World Problem: It can be seen that solving the Heavy Tile Problem with one sub-program required similar node type usage as the Tile World Problem. Adding to that, the algorithms didn't rely on (TT) the type of the nearest tile node, as the algorithms were using only one sub-program. When solving the Heavy Tile Problem with two sub-programs, using PCK-GNP, the (TT) node was used a lot. That is because the chromosome structure has two sub-programs, one for dealing with the Normal tile and another for the heavy tile. On the contrary, When applying CDA*-GNP with two sub-programs on the Heavy Tile World problem, it can be viewed from the graph that the algorithm didn't use the (TT) node. It can be inferred that the algorithm was able to solve the problem regardless of the existence of sub-programs.

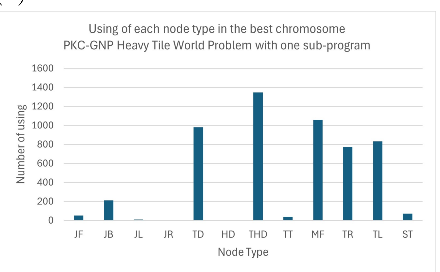
Prey and Predator Problem: It can be observed that the CDA*-GNP algorithm relied mostly on (TP) direction to the Prey and one of the judgment nodes(JR) to be able to make a decision and move forward (MF).



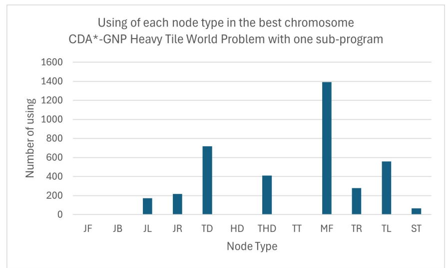
(a) PCK-GNP on Tile World Problem



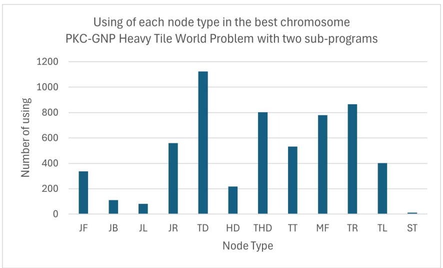
(b) CDA*-GNP on Tile World Problem



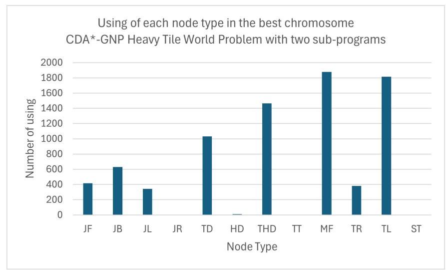
(c) PCK-GNP on Heavy Tile World Problem using one sub-program



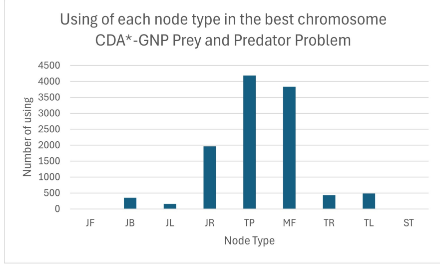
(d) CDA*-GNP on Heavy Tile World Problem using one sub-program



(e) PCK-GNP on Heavy Tile World Problem using two sub-programs



(f) CDA*-GNP on Heavy Tile World Problem using two sub-programs



(g) CDA*-GNP on Prey and Predator Problem

Figure 7.1: Usage of the various node types in the best chromosomes returned by PCK-GNP and CDA*-GNP on the three problems

7.5 Total Computational Costs of the Used Algorithms

The fitness evaluation process is the most computationally expensive process in the used algorithms. It accounts for 90.46% of the time, while the rest of the processes (sort, selection, crossover, mutation) account for 9.54% of the time. If a GNP individual can solve the problem before using all available steps for each agent, the time consumed will be much less than when the individual spends all available steps. Because of that, if the algorithm can reach the top result at an earlier generation, the total computational time for the 1000 generations will be much shorter.

1. In GNP-CC-OS-TP, the extra time is used on the (OS). When the agent visits a judgment node, the A* algorithm will be used to answer this judgment. We proved in [17] that although the A* needed extra time, the total computational time for the algorithm was better because the algorithm was able to get more performance results.

2. In PCK-GNP, the algorithm consumes more time while extracting the conflicts and within the crossover and mutation operation when comparing the PCK and the incremental repair technique. It also needs more memory size to save the PCK for each chromosome.

3. In CDA*-GNP, the algorithm uses more time when applying the CDA* on the best chromosome, as it needs to evaluate the chromosome each time the CDA* change a function of a processing node.

Table 7.10: Comparing the computational cost in the proposed algorithms when testing the Tile World Problem

Algorithm	Average used time for finishing 1000 generations	Average time used for reaching the top training results	Testing Success rate	Percentage of speed improvement comparing to the slowest algorithm
GNP	(3:45 hours) (225 min)	-	27%	82.5%
GNP-CC-OS-TP	(15:39 hours) (939 min)	-	50%	27.0%
GNP-RL	(7:25 hours) (445 min)	-	43%	65.4%
GNP-RL-CC-OS-TP	(12:31 hours) (751 min)	(2:04 hours) (124 min)	83%	41.6%
PCK-GNP	(21:28 hours) (1288 min)	(4:56 hours) (296 min)	100%	-
CDA*-GNP	(12:29 hours) (749 min)	(0:53 hours) (53 min)	100%	41.8%

Table 7.10 shows the average used time for finishing 1000 generations and the average time used to reach the top training results in hours and minutes for the algorithms (GNP, GNP-CC-OS-TP, GNP-RL, GNP-RL-CC-OS-TP, PCK-GNP, CDA*-GNP). It is clearly noticed that both the GNP-RL-CC-OS-TP and CDA*-GNP have the shortest time for finishing 1000 generations, while there is a big difference in the time used to reach the top training results for both algorithms. GNP-RL-CC-OS-TP needed an average of 2:04 hours to be able to reach the top training results, while CDA*-GNP was

able to reach the top training results within an average of 53 minutes. It is noteworthy that in the CDA*-GNP, although each generation did more evaluations than the other algorithms because of applying CDA*, the average time was the shortest, which was 41.8% faster than the slowest algorithm (PCK-GNP). Adding to this, the CDA*-GNP was able to reach 100% of testing success rate.

7.6 Conclusions

Here are some general observations and findings we have gathered from our study, along with the conclusions that we have reached:

1. All the evolutionary algorithms tested suffer from losing genetic diversity over time. If an algorithm succeeds in finding the best solution before it enters a phase of losing diversity, then it is successful. As the number of generations increases, the probability of entering a phase of losing diversity increases. Therefore, the number of needed evaluations to find the best solution from the population is just as important as the number of needed generations to improve the population of GNP individuals, if it could help steer the algorithm to avoid significant genetic diversity loss.

2. When using A* as a judgment node in a system with heterogeneity, two issues may arise. Firstly, in case the environment is large, A* takes more time to explore all the paths and to detect the shortest one. Secondly, when there are many objects, such as Tiles, Agents, and Holes, it takes a long time for the algorithm to determine the direction to the nearest tile. This is because the algorithm has to first detect the shortest path to all the tiles and then select the one with the shortest path as the nearest tile. To overcome this problem, we decided to use the Euclidean distance as an estimator to identify the approximate three nearest tiles and then use A* to obtain the shortest path to each of them. The tile with the shortest path is then chosen as the Nearest tile, and the algorithm returns the direction to the first point in it.

3. This work proposed techniques for extracting and dealing with conflicts. Extracting conflicts can be performed based on observed environmental conditions or based on computational graph structures. In PCK-GNP, the conflicts were extracted through interaction with the environment, while in CDA*-GNP, the conflicts were extracted from the graph. Extracting the conflicts from the environment was successful in speeding up finding the solution by preventing the generation of the substructs that cause conflicts. However, these conflicts are specific to the problems, such as (the Tile World Problem), while the Prey and

Predator Problem doesn't have a conflict that could be extracted from the environment. Extracting the conflicts from the graph as proposed by CDA*-GNP provides a more generalised approach as it lends itself amenable for use in the Tile Problem and its three other variants (Normal, Heavy, and Heterogeneous), as well as the Prey and Predator Problem. In CDA*-GNP-H, we extracted the conflicts from the graph in the CDA*-GNP-HN, and from the environment in CDA*-GNP-HM.

4. From our previous work [17], we found that using a more complex training environment with GNP-RL was not useful in improving the results. Rather, it was difficult for the algorithm to improve and to find a good solution. Similarly, in this work, we tried to use a complex training environment (which has different locations for the tiles, holes, and obstacles for each one of the 10 environments) with CDA*-GNP, but the algorithm failed to find a good solution. Each time the chromosome tries to configure a series of rules inside its graph that can push one tile into a hole, it will lose these rules when it tries to push another tile that has a totally different situation. To make use of a complex training environment useful in improving the solution, another method could be used, which is training the algorithm in simple environments in the first generations of the algorithm. Then, when the algorithm reaches an appropriate solution, the complex environment could be added to the training phase to increase the performance (example of future work).

5. GNP relies on stochastic processes in its core. The advantage of using stochastic optimization is that it can quickly explore interesting regions in the search space more quickly as compared to brute force approaches. However, incorporating randomness introduces a degree of unpredictability, which can sometimes result in losing track of good solutions. On the other hand, optimal search explores every possible path to find a solution in a complex graph, which needs a lot of exploration. Merging optimal search with stochastic optimisation strategie as in CDA*-GNP was able to optimally explore all the possible paths within a given graph that don't cause a conflict and randomly generate new individuals in the evolution process. Using both of them in one hybrid approach was able to improve the performance highly and significantly decrease the time needed to find the best solution.

6. The CDA*-GNP algorithm has been shown to outperform other GNP variants when tested on the standard TileWorld and Prey and Predator benchmarking problems for other GNP variants algorithms. Additionally, we created the Heavy Tile World and Heterogeneous problems as additional benchmarks. It achieved

the best performance measurements on the TileWorld problem with 100% accuracy on the test set compared to only 83% reported in the literature. Moreover, CDA*-GNP is 78% faster than the second-ranked algorithm (GNP-RL-CC-OS-TP) in terms of the average number of generations and 74% faster in terms of the average number of fitness evaluations. In the Heavy Tile World problem using (One sub-program & two sub-programs), the algorithm was able to get 100% accuracy on the test set in both experiments. Using two sub-programs with the Heavy Tile World problem, the algorithm was 78% faster in terms of the average number of fitness evaluations and 86% faster in terms of the average number of generations. In the Prey and Predator Problem, the algorithm achieved a 94% testing accuracy compared to the second-ranked algorithm (GNP-RL-CC-OS-TP), which got a 92% testing accuracy in the testing set. It was 39% faster in terms of the average number of fitness evaluations and 44% faster in terms of the average number of generations. In the heterogeneous system, CDA*GNP was able to achieve 90% accuracy in the training stage in the Fully heterogeneous system compared with 80% training accuracy in the Partially heterogeneous system.

7. Apart from the novel strategies we proposed in this thesis, we integrated recent well-tested techniques that exist in the literature:

   (a) The processing node (MF) was designed to deal with conflicts by using two connections that direct to two nodes instead of one [14].

   (b) The A* algorithm was used as a component node in the graph to answer the judgment nodes by calculating the shortest path [14].

   (c) The task prioritization technique was used to calculate the Distance in the Fitness function on the proposed algorithms [14].

   (d) The Incremental Repair [61] was used to improve the mutation in the PCK-GNP.

   (e) The variable-sized graph technique [9] was used in the experiments with the Heavy Tile World Problem when using two sub-programs.

   (f) Multiple diversity management strategies were explored in the proposed algorithms, as discussed in section 7.3. Keeping genetic diversity is deemed important to prevent getting stuck at local minima. We considered approaches that were utilised separately during the selection, crossover and mutation operations, as well as strategies that were employed on the combinations of those operations.

7.7 Future Work

In this section, we explore potential future directions based on our findings and proposals.

1. PCK-GNP:
   As we used Private-Conflict-Kernel for each individual that contains the substructure which causes a conflict, a Public-Conflict-Kernel for the population can be used to contain all the substructures that appear in the individuals' Private-Conflict-Kernels and use this public kernel to rank the individuals and prevent the individuals from having a conflict substructure during the evolutionary operations.

2. CDA*-GNP:
   CDA*-GNP can be used to solve other real-world problems involving multi-agents and control systems. We explored strategies for eliminating conflicts due to the nodes. Other conflicts arising due to the connections and assignment of the start node could also be explored.

3. CDA*-GNP-H:
   The CDA*-GNP-H algorithm can be tested on a partly heterogeneous system that divides and merges the populations between the agents to try to improve its performance. Another way to enhance the algorithm is to rank members while exploring them using CDA*. This could help detect weaker members and replace them with stronger ones from different populations. Another technique that could be used to improve the algorithm is to apply evolutionary operations on individuals that are selected from different populations that share the same type of agents.

   In general, the proposed algorithm is well-suited for problem domains that necessitate the development of intelligence for decision-making, control, navigation, and coordination of agents. Its limitations are solely defined by the computing capabilities of the judgment and processing nodes. The algorithm operates on the assumption that by creating a computational graph based on predefined judgement nodes and processing nodes, a specific problem can be effectively solved. Different challenges require different sets of processing and judgment nodes, but the same training strategy can be applied. Here are various examples of heterogeneous multi-agent domains to which the algorithm could be applied:

   (a) Self-driving Cars
       The decision-making process in modern autonomous driving systems is typically structured hierarchically into four layers, encompassing route planning,

behavioural decision-making, local motion planning, and feedback control.

i. **Route Planning Layer**: At the highest level, a vehicle's decision-making system must select a route through the road network from its current position to the requested destination.

ii. **Behavioural Planning Layer**: Given a sequence of road segments specifying the selected route, the behavioural layer is responsible for selecting an appropriate driving behaviour at any point in time based on the perceived behaviour of other traffic participants, road conditions, and signals from infrastructure.

iii. **Motion Planning Layer**: The motion planning layer takes the goal provided by the behavioural layer and finds a safe, comfortable and collision-free path based on the current dynamic constraints of the car.

iv. **Control Layer**: In order to execute the reference path or trajectory from the motion planning system, a feedback controller is used to select appropriate actuator inputs to carry out the planned motion and correct tracking errors.

The decision-making system is responsible for guiding the car from its starting position to the user-defined final goal. It takes into account the car's current state, the internal representation of the environment, traffic rules, and the safety and comfort of the passengers. Cars use cameras, radar, and other sensor systems to detect road features and obstacles. Based on the sensor feedback, the car should take action, such as adjusting speed, stopping, or turning left or right.

(b) Robot Soccer game

Another potential application of the proposed algorithm is Robot Football, a game involving two teams with attacking, defending, and goalkeeping players. Applying the proposed algorithms to this problem would require addressing conflicts that may arise from the environments (e.g. pushing the ball to a player from the opposing team, scoring an own goal, touching the ball with the hand or arm, and so on). Secondly, the conflict that can be eliminated from the graph would be similar to what was discussed in this thesis (e.g. turning actions that cause a loop, repetition in judgment nodes, and so on). Thirdly, the conflict between the members can be as follows: when an agent is in the way of another one, when two agents from the same team try to push the ball together at the same time, etc.

(c) Automated Storage Systems

Business owners are exploring options to automate their storage systems in order to save time and money. However, the main challenge is based on

finding a way for the agent to store objects in the most efficient manner possible, using the appropriate amount of space without leaving any gaps between the objects. Additionally, the system should be able to reuse any free space that becomes available. Applying the proposed algorithms to this problem would require addressing conflicts that may arise from the environment such as: misplacement of a small object in an oversized location, attempting to fit a large object into an insufficiently sized space, placing an object in an incorrect location, and so on. Secondly, the conflict that can be extracted from the graph is similar to what we discussed in this thesis (e.g. turning actions that cause a loop, repetition in judgment nodes, and so on). Thirdly, the conflicts that may arise involving peer agents are as follows: when an agent is in the way of another one, when two agents try to push the same object, and so on.

7.8 List of Abbreviations

Table 7.11: Abbreviations

#	Abbreviation	Definition
1	A*-GNP	A* with Genetic Network Programming
2	ACNP	Graph Structure Optimization of Genetic Network Programming with Ant Colony Mechanism
3	ACO	Ant Colony Optimization
4	CBA*	Constraint-Based A*
5	CBS	Conflict-Based Search
6	CCEAs	Cooperative Coevolutionary Algorithms
7	CDA*	Conflict Directed A*
8	CDA*-GNP	Conflict Directed A* with Genetic Network Programming
9	CDA*-GNP-H	Conflict-Directed A* and Genetic Network Programming with Heterogeneous system
10	CDA*-GNP-HM	Conflict-Directed A* and Genetic Network Programming with Heterogeneous system to explore the Members
11	CDA*-GNP-HN	Conflict-Directed A* and Genetic Network Programming with a Heterogeneous system to explore the Nodes
12	CDA*-GNP-HNM	Conflict-Directed A* and Genetic Network Programming with Heterogeneous system to explore the Nodes and the Members
13	CDA*-VSGNP	Conflict Directed A* with Variable-sized Genetic Network Programming
14	CSP	Constraint Satisfaction Problem
15	DBA*-GNP	Decision-Based A* with Genetic Network Programming
16	DGNP-RL	Distributed Genetic Network Programming with Reinforcement Learning
17	EGSCS	Elevator Group Supervisory Control Systems
18	GA	Genetic Algorithm
19	GNP	Genetic Network Programming
20	GNP-CC-OS-TP	Genetic Network Programming with Constraint Conformance, Optimal Search, and Task Prioritization
21	GNP-RL	Genetic Network Programming with Reinforcement Learning
22	GNP-RL-CC-OS-TP	Genetic Network Programming with Reinforcement Learning, Constraint Conformance, Optimal Search, and Task Prioritization
23	GP	Genetic Programming
24	HD	Direction to the nearest Hole
25	JB	What is in the Backward position
26	JF	What is in the Forward position
27	JL	What is in the Left position
28	JR	What is in the Right position
29	LPA*	Lifelong Planning A*
30	MAPF	Multi-Agent Path Finding
31	MAS	Multi-agent System
32	MBD	Model-Based Diagnosis
33	MF	Move Forward
34	PBS	Priority-Based Search
35	Pc	Crossover Rate
36	PCK-GNP	Private Conflict Kernels with Genetic Network Programming
37	PCK-GNP (C)	Using the Private-Conflict-Kernels on the Crossover
38	PCK-GNP (M)	Using the Private-Conflict-Kernels on the Mutation
39	PCK-GNP (S)	Using the Private-Conflict-Kernels on the Selection
40	PCK-VSGNP	Private Conflict Kernels with Variable-sized Genetic Network Programming
41	PD	Direction to the nearest Prey
42	Pm	Mutation Rate when changing the node connection
43	Pm1	Mutation Rate when changing the node function
44	PT	Priority Tree
45	STD	Direction to the second nearest Tile
46	STD	Stay
47	TD	Direction the the nearest Tile
48	THD	Direction from the nearest Tile to the nearest Hole
49	TL	Turn Left
50	TR	Turn Right
51	TT	Type of the nearest Tile
52	VSGNP	Variable-sized Genetic Network Programming
53	VSGNP-CC-OS-TP	Variable-sized Genetic Network Programming with Constraint Conformance, Optimal Search, and Task Prioritization
54	VSGNP-RL	Variable-sized Genetic Network Programming with Reinforcement Learning
55	VSGNP-RL-CC-OS-TP	Variable-sized Genetic Network Programming with Reinforcement Learning, Constraint Conformance, Optimal Search, and Task Prioritization

Glossary

Average number of evaluations until get the top training result The average number of evaluations that the algorithm needs until gets the highest results in the training stage. 57

Best Chromosome When we say the best chromosome, we mean the chromosome that gives the top result that has been found from the algorithm-generated chromosomes. 14

Best combination of functions The combination of selected functions for each node that achieves the highest results. 127

Best Experiment In this study, we evaluated various algorithms by conducting multiple experiments for each one. The best experiment was determined by achieving the highest results compared to the others. 121

Best Graph When we say the best graph, we mean the graph that gives the top result that has been found from the algorithm-generated graphs. 2, 136

Best Individual When we say the best individual, we mean the individual that gives the top result that has been found from the algorithm-generated individuals. 51

Best Solution When we say the best solution, we mean the solution that gives the top result that has been found from the algorithm-generated solutions. 2, 16

Best Sub-node The sub-node that achieves the highest fitness value among the other sub-nodes. 31

Best-Team The team that achieves the highest results. 185

Conflict A conflict represents a set of states, all of which are conclusively proven inconsistent using the same proof. 2, 43

Conflict and Kernel diagnosis Conflict and kernel diagnosis are utilized to remove inconsistent subspace around each state diagnosis. 107

Conflict Substructs Conflict substructs are part or series of nodes from the graph that lead to a conflict. In this chapter, the extracted conflicts are from the Tile World Problem. One of the conflicts that we have noticed is when the agent pushes a tile to a trapped location, as that will lead to the tile not being able to be pushed anywhere else. 107

Conflicts from the networking graph The nodes function that cause a conflict in the graph. We used these four types of graph conflicts: Loop, Repetition in turns, Repetition in Go Forward, or Repetition in Judgment nodes. For example, when there are three or more (Turn Left / Turn Right) nodes after each other without any other nodes between them. When there is a loop of nodes (from 2 to 4 nodes) that repeats after each other and it does not contain a processing node. When there are two or more (Go Forward) nodes after each other without any other nodes between them. When there is a repetition of the Judgment nodes in the records without any processing nodes between them. 127

Constraint Satisfaction Problem Constraint Satisfaction Problem (CSP) is a problem-solving technique that Artificial Intelligence utilizes to tackle a diverse range of issues. 106

Constraints The constraints are the conditional equations that govern the linear function. 129

Decision Variables The variables whose values are to be determined. The equations are solved to obtain the optimal value of these variables. 129

Domain The domain is the set of values that a variable is able to take. 129

Feasible States It is the region in the graph where the constraints are met, and the decision variables are located at the corners of the region. 107

Fully Heterogeneous System A fully heterogeneous system means a separate population for each agent. 7, 180

Heterogeneous System A heterogeneous system implies that each group of agents has access to the same actions but with different actions from other groups of agents. 2, 178

Homogenous System A homogenous system implies that all agents have access to the same actions in any given state, and these actions yield the same outcomes irrespective of which agents execute them.

Kernels The kernel describes a set of states that resolve the known conflicts. 106

Number of evaluations until get the best chromosome The number of evaluations that the algorithm needs until gets the chromosome that achieves the highest result among all the experiments. 57

Number of generations for the best chromosome The number of generations that the algorithm needs until it gets the chromosome that achieves the highest result among all the experiments. 57

Objective Function and Fitness Function The objective function essentially defines the objective of the problem. The fitness function is a particular type of objective function that is used to quantify how closely a given design solution aligns with the set aims. 1, 16

Optimal Search The optimal search algorithm is designed to find the shortest path between a start state and a goal state and typically uses heuristic methods to estimate the path. 127

Optimal Solution An optimal solution is a feasible solution in which the objective function reaches its maximum or minimum value. 2, 17

Partly Heterogeneous System A partly heterogeneous system means a shared population of a number of agents who share the same behaviour. 7, 180

Private-conflict-kernel An assignment containing all conflicting sub-structures for a specific individual. We utilized a private conflict kernel for each chromosome, which contains all sub-structures with conflicts. 2

Substructs Substructs are part or series of nodes from the graph. 107

Bibliography

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems.* The MIT Press, 1992.

[2] J. R. Koza, *Statistics and Computing.* Springer, 1994, vol. 4, ch. Genetic programming as a means for programming computers by natural selection, pp. 87–112.

[3] V. Ciesielski, D. Mawhinney, and P. Wilson, "Genetic programming for robot soccer," in *RoboCup 2001: Robot Soccer World Cup V*, A. Birk, S. Coradeschi, and S. Tadokoro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 319–324.

[4] B. Li, S. Mabu, and K. Hirasawa, "Genetic network programming with automatic program generation for agent control," *Transaction of the Japanese Society for Evolutionary Computation*, vol. 1, no. 1, pp. 43–53, 2010.

[5] H. Katagiri, K. Hirasama, and J. Hu, "Genetic network programming - application to intelligent agents," in *Smc 2000 conference proceedings. 2000 IEEE international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0)*, vol. 5, 2000, pp. 3829–3834.

[6] S. Mabu, K. Hirasawa, and J. Hu, "A graph-based evolutionary algorithm: Genetic network programming (gnp) and its extension using reinforcement learning," *Evolutionary Computation*, vol. 15, no. 3, pp. 369–398, 2007.

[7] P. Stone, G. A. Kaminka, S. Kraus, and J. S. Rosenschein, "Ad hoc autonomous agent teams: Collaboration without pre-coordination," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, ser. AAAI'10. AAAI Press, 2010, pp. 1504—-1509.

[8] M. E. Pollack and M. Ringuette, "Introducing the tileworld: Experimentally evaluating agent architectures," in *Proceedings of the Eighth National Conference*

on Artificial Intelligence - Volume 1, ser. AAAI'90. AAAI Press, 1990, pp. 183–189.

[9] S. Mabu, K. Hirasawa, M. Obayashi, and T. Kuremoto, "A variable size mechanism of distributed graph programs and its performance evaluation in agent control problems," *Expert Systems with Applications*, vol. 41, no. 4, Part 2, pp. 1663–1671, 2014.

[10] X. Li, M. Yang, and S. Wu, "Niching genetic network programming with rule accumulation for decision making: An evolutionary rule-based approach," *Expert Systems with Applications*, vol. 114, pp. 374–387, 2018.

[11] M. Benda, V. Jagannathan, and R. Dodhiawala, "On optimal cooperation of knowledge sources - an empirical investigation," Boeing Advanced Technology Center, Boeing Computing Services, Seattle, WA, USA, Tech. Rep. BCS–G2010–28, July 1986. [Online]. Available: http://www.cs.utexas.edu/~shivaram

[12] M. Roshanzamir, M. Palhang, and A. Mirzaei, "Efficiency improvement of genetic network programming by tasks decomposition in different types of environments," *Genetic Programming and Evolvable Machines*, vol. 22, pp. 229–266, 2021.

[13] M. T. Cox and A. Raja, *Metareasoning: Thinking about Thinking.* The MIT Press, 2011. [Online]. Available: https://doi.org/10.7551/mitpress/9780262014809.001.0001

[14] M. Alshehri, N. Reyes, and A. Barczak, "Evolving meta-level reasoning with reinforcement learning and a* for coordinated multi-agent path-planning," in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '20, 2020, p. 1744–1746.

[15] J. R. Koza, *Genetic Programming On the Programming of Computers by Means of Natural Selection.* Cambridge: MIT Press, 1992.

[16] Y. Yang, Z. He, S. Mabu, and K. Hirasawa, "A cooperative coevolutionary stock trading model using genetic network programming-sarsa," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 16, no. 5, pp. 581 – 590, 2012.

[17] M. A. Alshehri, "Genetic network programming with reinforce-ment learning and optimal search component," Master Thesis, Computer Science, INMS, Massey University, Auckland, NZ, 2019.

[18] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, vol. 4, pp. 65–85, 1994.

[19] H. Kargupta, K. Buescher, and J. Gattiker, “Credit card fraud detection: An application of the gene expression messy genetic algorithm,” in *Knowledge Discovery and Data Mining (KDD96)*, Portland, Aug. 1996.

[20] W. Min and Y. Shuyuan, “A hybrid genetic algorithm-based edge detection method for sar image,” in *IEEE International Radar Conference, 2005.*, 2005, Journal Article, pp. 503–506.

[21] Y. Sun, C. F. Babbs, and E. Delp, “A comparison of feature selection methods for the detection of breast cancers in mammograms: Adaptive sequential floating search vs. genetic algorithm,” in *IEEE Engineering in Medicine and Biology 27th Annual Conference Engineering in Medicine and Biology Society*, 2005.

[22] S. S. Juneja, P. Saraswat, K. Singh, J. Sharma, R. Majumdar, and S. Chowdhary, “Travelling salesman problem optimization using genetic algorithm,” in *2019 Amity International Conference on Artificial Intelligence (AICAI)*, 2019, pp. 264–268.

[23] H. Mühlenbein and D. Schlierkamp-Voosen, “Predictive models for the breeder genetic algorithm i. continuous parameter optimization,” *Evolutionary Computation*, vol. 1, no. 1, pp. 25–49, 1993.

[24] T. Blickle and L. Thiele, “A comparison of selection schemes used in evolutionary algorithms,” *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996.

[25] J. D. Schaffer and A. Morishima, “An adaptive crossover distribution mechanism for genetic algorithms,” in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application (ICGA’87)*. USA: L. Erlbaum Associates Inc., 1987, p. 36–40.

[26] D. Fogel and J. Atmar, “Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems,” *Biological Cybernetics*, vol. 63, p. 111–114, 1990.

[27] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, “Genetic algorithms for the travelling salesman problem: A review of representations and operators,” *Artificial Intelligence Review*, vol. 13, p. 129–170, 1999.

[28] K. Deep and M. Thakur, “A new mutation operator for real coded genetic algorithms,” *Applied Mathematics and Computation*, vol. 193, p. 211–230, 2007.

[29] J.-Y. Potvina, P. Sorianoa, and M. Vale, “Generating trading rules on the stock markets with genetic programming,” *Computers & Operations Research*, vol. 31, no. 7, p. 1033 – 1047, 2004.

[30] M. L. Raymer, W. F. Punch, E. D. Goodman, and L. A. Ku, "Genetic programming for improved data miningapplication to the biochemistry of protein interactions," in *Proc. 1st Annu. Conf. Genetic Program*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., Stanford University, 1996, pp. 375–380.

[31] C. Estbanez, J. M. Valls, and R. Aler, "Gppe: A method to generate ad-hoc feature extractors for prediction in financial domains," *Applied Intelligence*, vol. 29, no. 3, pp. 174–185, 2008.

[32] H. Guo and A. K. Nandi, "Breast cancer diagnosis using genetic programming generated feature," *Pattern Recogn.*, vol. 39, no. 5, p. 980–987, may 2006.

[33] M. T. Ahvanooey, Q. Li, M. Wu, and S. Wang, "A survey of genetic programming and its applications," *KSII Transactions on Internet and Information Systems*, vol. 13, no. 4, pp. 1765–1794, Apr 2019.

[34] P. G. Espejo, S. Ventura, and F. Herrera, "A survey on the application of genetic programming to classification," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 2, pp. 121–144, 2010.

[35] K. Hirasawa, M. Okubo, H. Katagiri, J. Hu, and J. Murata, "Comparison between genetic network programming (gnp) and genetic programming (gp)," in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, vol. 2, 2001, pp. 1276–1282.

[36] S. Mabu, K. Hirasawa, J. Hu, and J. Murata, "Online learning of genetic network programming (gnp)," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 1, 2002, pp. 321–326.

[37] S. Mabu, K. Hirasawa, J. Hu, and J. Murata, "Online Learning of Genetic Network Programming and its Application to Prisoner's Dilemma Game," *IEEJ Transactions on Electronics, Information and Systems*, vol. 123, no. 3, pp. 535–543, Jan. 2003.

[38] W. Wang, "Genetic network programming with fuzzy reinforecement learning nodes for multi-behaviour robot control," Master's thesis, Computer Science, INMS, Massey University, Auckland, NZ, 2014. [Online]. Available: https://mro.massey.ac.nz/items/62440af8-b856-4e88-8d41-dc1bd89a9309

[39] W. Wang, N. H. Reyes, A. L. C. Barczak, T. Susnjak, and P. Sincak, "Multi-behaviour robot control using genetic network programming with fuzzy reinforcement learning," in *Robot Intelligence Technology and Applications 3 - Results from the 3rd International Conference on Robot Intelligence Technology*

and Applications, RiTA 2014, Beijing, China, November 6-8, 2014, ser. Advances in Intelligent Systems and Computing, J. Kim, W. Yang, J. Jo, P. Sincak, and H. Myung, Eds., vol. 345. Springer, 2014, pp. 151–158. [Online]. Available: https://doi.org/10.1007/978-3-319-16841-8_15

[40] S. Mabu, Y. Chen, S. Eto, K. Shimada, and K. Hirasawa, "Genetic network programming with intron-like nodes," *Electronics and Communications in Japan*, vol. 93, no. 8, p. 1312–1319, 2010.

[41] A. Damia, M. Esnaashari, and M. Parvizimosaed, "Adaptive genetic algorithm based on mutation and crossover and selection probabilities," in *2021 7th International Conference on Web Research (ICWR)*, 2021, pp. 86–90.

[42] M. Roshanzamir, M. Palhang, and A. Mirzaei, "Graph structure optimization of genetic network programming with ant colony mechanism in deterministic and stochastic environments," *Swarm and Evolutionary Computation*, vol. 51, p. 100581, 2019.

[43] Y. Chen, S. Mabu, and K. Hirasawa, "A model of portfolio optimization using time adapting genetic network programming," *Comput. Oper. Res.*, vol. 37, no. 10, p. 1697–1707, 2010.

[44] X. Li and K. Hirasawa, "Continuous probabilistic model building genetic network programming using reinforcement learning," *Applied Soft Computing*, vol. 27, p. 457–467, 2015.

[45] S. Mabu, C. Chen, N. Lu, K. Shimada, and K. Hirasawa, "An intrusion-detection model based on fuzzy class-association-rule mining using genetic network programming," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 1, pp. 130–139, 2011.

[46] X. Li, H. Yang, and M. Yang, "Revisiting genetic network programming (gnp): Towards the simplified genetic operators," *IEEE Access*, vol. 6, pp. 43 274–43 289, 2018.

[47] S. Mabu, H. Hatakeyama, M. Thu Thu, K. Hirasawa, and J. Hu, "Genetic network programming with reinforcement learning and its application to making mobile robot behaviour," *IEEJ Transactions on Electronics Information and Systems*, vol. 126, no. 8, pp. 1009–1015, 2006.

[48] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[49] Y. Yang, J. Li, S. Mabu, and K. Hirasawa, "Gnp-sarsa with subroutines for trading rules on stock markets," in *2010 IEEE International Conference on Systems, Man and Cybernetics*, 2010, pp. 1161–1165.

[50] Y. Chen, S. Mabu, K. Shimada, and K. Hirasawa, "A genetic network programming with learning approach for enhanced stock trading model," *Expert Systems with Applications*, vol. 36, no. 10, pp. 12 537–12 546, 2009.

[51] R. K. Ursem, "Diversity-guided evolutionary algorithms," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2002, pp. 462–471.

[52] H. Shimodaira, "A diversity-control-oriented genetic algorithm (dcga): development and initial experimental results," *Trans. Inf. Process. Soc. Jpn.(Japan)*, vol. 40, no. 6, pp. 2708–2716, 1999.

[53] F. Oppacher, M. Wineberg *et al.*, "The shifting balance genetic algorithm: Improving the ga in a dynamic environment," in *Proceedings of the genetic and evolutionary computation conference*, vol. 1, 1999, pp. 504–510.

[54] S. Tsutsui, Y. Fujimoto, and A. Ghosh, "Forking genetic algorithms: Gas with search space division schemes," *Evolutionary computation*, vol. 5, no. 1, pp. 61–80, 1997.

[55] W. B. Langdon, "Data structures and genetic programming: Genetic programming+ data structures= automatic programming!, volume 1 of genetic programming," 1998.

[56] L. Eshelman and J. Schaffer, "Crossover's niche," in *Proc. 5th Int. Conf. Genetic Algorithms*. Morgan Kaufmann, 1993, pp. 9–14.

[57] C. Ryan, *Pygmies and civil servants*. Cambridge, MA, USA: MIT Press, 1994, pp. 243–263.

[58] A. Damia, M. Esnaashari, and M. Parvizimosaed, "Adaptive genetic algorithm based on mutation and crossover and selection probabilities," in *2021 7th International Conference on Web Research (ICWR)*. IEEE, 2021, pp. 86–90.

[59] W. Hamscher, L. Console, and J. de Kleer, Eds., *Readings in model-based diagnosis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[60] J. de Kleer and B. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, no. 1, p. 97–130, 1987.

[61] B. Selman, H. J. Levesque, and D. G. Mitchell, "A new method for solving hard satisfiability problems," in *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*, W. R. Swartout, Ed. AAAI Press / The MIT Press, 1992, pp. 440–446. [Online]. Available: http://www.aaai.org/Library/AAAI/1992/aaai92-068.php

[62] O. Dressler and A. Farquhar, "Problem solver control over the atms," in *GWAI-89 13th German Workshop on Artificial Intelligence*, D. Metzing, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 17–26.

[63] J. de Kleer and B. C. Williams, "Diagnosis with behavioral modes," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, p. 1324–1330.

[64] J. de Kleer and B. Williams, "Focusing the diagnosis engine," in *Proceedings of DX-90*, 1990.

[65] B. Williams and P. Nayak, "A model-based approach to reactive self-configuring systems," in *Proceedings of AAAI-96*, vol. 13, Portland, Oregon, Aug. 1996, Journal Article, p. 971–978.

[66] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, p. 57–95, 1987.

[67] J. de Kleer, A. Mackworth, and R. Reiter, "Characterizing diagnoses and systems," *Artificial Intelligence*, vol. 56, no. 2, p. 197–222, 1992.

[68] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence.* New York: McGraw-Hill, 1971.

[69] H. Anton and C. Rorres, *Elementary Linear Algebra (11th ed.).* United States of America: Wiley, 2014.

[70] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp. 3310–3317 vol.4.

[71] S. Koenig and M. Likhachev, "Incremental a*," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 14, 2001, Journal Article, pp. 1539–1546.

[72] ——, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

[73] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, R. Bartak, and E. Boyarski, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *In Proceedings of the International Symposium on Combinatorial Search (SoCS 2019)*, 2019, pp. 151 – 159.

[74] F. Ho, A. Salta, R. Geraldes, A. Goncalves, M. Cavazza, and H. Prendinger, "Multi-agent path finding for uav traffic management," in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '19. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2019, p. 131–139.

[75] J. Li, T. A. Hoang, E. Lin, H. L. Vu, and S. Koenig, "Intersection coordination with priority-based search for autonomous vehicles," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 10, pp. 11 578–11 585, Jun. 2023. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/26368

[76] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Magazine*, vol. 29, no. 1, pp. 9–20, Mar. 2008.

[77] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI)*, 2013, p. 1443–1449.

[78] E. Boyarski, A. Felner, R. Stern, G. Sharon, O. Betzalel, D. Tolpin, and E. Shimony, "Icbs: The improved conflict-based search algorithm for multi-agent pathfinding," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 6, 2015, pp. 223–225.

[79] J. Li, D. Harabor, P. J. Stuckey, H. Ma, and S. Koenig, "Symmetry-breaking constraints for grid-based multi-agent path finding," in *Proceedings of the AAAI Conference on Artifcial Intelligence (AAAI)*, vol. 33, Jul. 2019, p. 6087–6095.

[80] J. Li, G. Gange, D. Harabor, P. J. Stuckey, H. Ma, S. Koenig, J. Li, G. Gange, D. Harabor, P. J. Stuckey, H. Ma, and S. Koenig, "New techniques for pairwise symmetry breaking in multi-agent path finding," in *In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, vol. 30, 2020, Journal Article, p. 193–201. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/6661

[81] D. Silver, "Cooperative pathfnding," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 1, 2005, p. 117–122.

[82] H. Ma, D. Harabor, P. J. Stuckey, J. Li, and S. Koenig, "Searching with consistent prioritization for multi-agent path finding," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019, p. 7643–7650. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33017643

[83] B. C. Williams and R. J. Ragno, "Conflict-directed a* and its role in model-based embedded systems," *Discrete Applied Mathematics*, vol. 155, p. 1562 – 1595, 2007.

[84] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[85] E. J. Hoffman, J. C. Loessi, and R. C. Moore, "Constructions for the solution of the m queens problem," *Mathematics Magazine*, vol. 42, pp. 66–72, 1969. [Online]. Available: https://api.semanticscholar.org/CorpusID:120212027

[86] C. Boyer and W. Trump, "Sudoku's french ancestors," Weblink, 2007. [Online]. Available: http://www.multimagie.com/English/SudokuAncestors.htm

[87] L. Barenboim, M. Elkin, and F. Kuhn, "Distributed $(\Delta + 1)$-coloring in linear (in Δ) time," *SIAM Journal on Computing*, vol. 43, no. 1, pp. 72–95, 2014. [Online]. Available: https://doi.org/10.1137/12088848X

[88] S. Levine, "Conflict-directed a* - a gentle introduction," Open CourseWare, 2016. [Online]. Available: https://ocw.mit.edu/courses/16-412j-cognitive-robotics-spring-2016/1b31309f2e8e0692fb3e1c6e51f901f8_MIT16_412JS16_RR1.pdf

[89] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an effcient sat solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

[90] M. Sachenbacher and B. C. Williams, "Conflict-directed a* search for soft constraints," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, J. C. Beck and B. M. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 182–196.

[91] P. Yu and B. Williams, "Continuously relaxing over-constrained conditional temporal problems through generalized conflict learning and resolution," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, p. 2429–2436.

[92] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part-B*, vol. 26, no. 1, pp. 29–41, 1996.

[93] T. Stutzle and H. Hoos, "Max-min ant system and local search for the traveling salesman problem," in *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, 1997, pp. 309–314.

[94] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, 1997.

[95] L. Yu, J. Zhou, S. Mabu, K. Hirasawa, J. Hu, and S. Markon, "Double-deck elevator group supervisory control system using genetic network programming with ant colony optimization with evaporation," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 11, no. 9, pp. 1149–1158, 2007.

[96] ——, "Elevator group control system using genetic network programming with aco considering transitions," in *SICE Annual Conference 2007*, 2007, pp. 1330–1336.

[97] G. Zhou, W. Feng, B. Jiang, and C. Li, "Computing minimal hitting set based on immune genetic algorithm," *International Journal of Modelling, Identification and Control*, vol. 21, no. 1, pp. 93–100, 2014.

[98] Y. F. Yiu, J. Du, and R. Mahapatra, "Evolutionary heuristic a* search: Heuristic function optimization via genetic algorithm," in *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, 2018, pp. 25–32.

[99] J. Gomes, P. Mariano, and A. L. Christensen, "Cooperative coevolution of partially heterogeneous multiagent systems," in *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '15. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2015, p. 297–305.

[100] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Auton. Agents Multi Agent Syst*, vol. 11, no. 3, p. 387–434, 2005.

[101] A. Campbell and A. S. Wu, "Multi-agent role allocation: Issues, approaches, and multiple perspectives," *Auton. Agent Multi-Agent Syst.*, vol. 22, no. 2, p. 317–355, 2011.

[102] T. Balch, "Behavioral diversity in learning robot teams," Ph.D. dissertation, College Comput., Georgia Inst. Technol., Atlanta, GA, USA, 1998.

[103] M. Waibel, L. Keller, and D. Floreano, "Genetic team composition and level of selection in the evolution of cooperation," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 3, pp. 648–660, 2009.

[104] S. Luke, "Genetic programming produced competitive soccer softbot teams for robocup97," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, Jul. 1998, p. 214–222.

[105] L. Iocchi, D. Nardi, M. Piaggio, and A. Sgorbissa, "Distributed coordination in heterogeneous multi-robot systems," *Autonomous Robots*, vol. 15, no. 2, pp. 155–168, 2003.

[106] J. Gomes, M. Duarte, P. Mariano, and A. L. Christensen, "Cooperative coevolution of control for a real multirobot system," in *Parallel Problem Solving from Nature – PPSN XIV*, J. Handl, E. Hart, P. R. Lewis, M. López-Ibáñez, G. Ochoa, and B. Paechter, Eds. Cham: Springer International Publishing, 2016, pp. 591–601.

[107] G. S. Nitschke, A. E. Eiben, and M. C. Schut, "Evolving team behaviors with specialization," *Genetic Programming and Evolvable Machines*, vol. 13, no. 4, p. 493–536, 2012.

[108] P. Lichocki, S. Wischmann, L. Keller, and D. Floreano, "Evolving team compositions by agent swapping," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, p. 282–298, 2013.

[109] M. A. Potter, L. Meeden, and A. C. Schultz, "Heterogeneity in the coevolved behaviors of mobile robots: the emergence of specialists," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 1337–1343.

[110] M. A. Potter and K. A. De Jong, "Cooperative coevolution: An architecture for evolving coadapted subcomponents," *Evol. Comput.*, vol. 8, no. 1, p. 1–29, mar 2000.

[111] T. Jansen and R. P. Wiegand, "Exploring the explorative advantage of the cooperative coevolutionary (1+1) ea," in *Genetic and Evolutionary Computation GECCO 2003*, vol. 2723, Berlin, Heidelberg, 2003, p. 310–321.

[112] C. H. Yong and R. Miikkulainen, "Coevolution of role-based cooperation in multiagent systems," *IEEE Transactions on Autonomous Mental Development*, vol. 1, no. 3, p. 170–186, 2009.

[113] R. P. Wiegand, *An analysis of cooperative coevolutionary algorithms.* George Mason University, 2004.

[114] L. Panait, "Theoretical convergence guarantees for cooperative coevolutionary algorithms," *Evolutionary computation*, vol. 18, no. 4, pp. 581–615, 2010.

[115] P. Mariano, A. Christensen, and J. Gomes, "Avoiding convergence in cooperative coevolution with novelty search," *Avoiding convergence in cooperative coevolution with novelty search*, pp. 1149–1156, 2014.

[116] A. Agogino and K. Tumer, "Efficient evaluation functions for evolving coordination," *Evolutionary Computation*, vol. 16, no. 2, pp. 257–288, 2008.

[117] G. Nitschke, "Behavioral heterogeneity, cooperation, and collective construction," in *2012 IEEE Congress on Evolutionary Computation.* IEEE, 2012, pp. 1–8.

[118] M. A. Potter, L. A. Meeden, A. C. Schultz *et al.*, "Heterogeneity in the coevolved behaviors of mobile robots: The emergence of specialists," in *International joint conference on artificial intelligence*, vol. 17, no. 1. Citeseer, 2001, pp. 1337–1343.

[119] G. S. Nitschke, "Neuro-evolution for emergent specialization in collective behavior systems," 2009.

[120] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous agents and multi-agent systems*, vol. 11, pp. 387–434, 2005.

[121] S. Luke *et al.*, "Genetic programming produced competitive soccer softbot teams for robocup97," *Genetic Programming*, vol. 1998, pp. 214–222, 1998.

[122] A. Hara, "Emergence of cooperative behavior using adg; automatically defined groups," in *GECCO-99: Proc. Genetic and Evolutionary Computation Conference.* Morgan Kaufmann, 1999, pp. 1039–1046.

[123] J. C. Bongard, "The legion system: A novel approach to evolving heterogeneity for collective problem solving," in *European Conference on Genetic Programming.* Springer, 2000, pp. 16–28.

[124] P. Lichocki, S. Wischmann, L. Keller, and D. Floreano, "Evolving team compositions by agent swapping," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 2, pp. 282–298, 2012.

[125] D. B. D'Ambrosio and K. O. Stanley, "Generative encoding for multiagent learning," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 2008, pp. 819–826.

[126] G. S. Nitschke, M. C. Schut, and A. Eiben, "Collective neuro-evolution for evolving specialized sensor resolutions in a multi-rover task," *Evolutionary Intelligence*, vol. 3, pp. 13–29, 2010.

[127] G. Nitschke, "Behavioral heterogeneity, cooperation, and collective construction," in *2012 IEEE Congress on Evolutionary Computation*, 2012, pp. 1–8.

[128] J. Gomes, P. Mariano, and A. L. Chri, "Dynamic team heterogeneity in cooperative coevolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 6, pp. 934–948, 2018.