PATHFINDING ALGORITHM OPTIMIZATION VIA EVOLUTION

A Dissertation

by

YING FUNG YIU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Rabi Mahapatra
Committee Members,	Aakash Tyagi
	Thomas Ioerger
	Paul Gratz
Head of Department,	Scott Schaefer

December 2020

Major Subject: Computer Engineering

ABSTRACT

Pathfinding is a popular computer science problem in both academic research and industrial development. The objective of pathfinding is to search for a path, often the shortest path, from one location to another on a graph. Many real world applications can be considered as pathfinding problems, including motion planning, video games, logistics, and decision making. A* search algorithm is the de facto pathfinding algorithm that uses a heuristic function to determine the best action to take based on the given information. The performance of A* is heavily dependent on the quality of the heuristic function. Hence, designing good heuristic functions for specific domains becomes the primary research focus on pathfinding algorithm optimization. In this dissertation, we address and solve several commonly known challenges in pathfinding problems and A* algorithm.

First, designing new heuristic functions is a difficult and time-consuming task, especially when they are used to solve complex problems. The task requires the user to have expert knowledge of the problem. Moreover, a single heuristic function might not be enough to digest all the provided information and return the best guidance during the search. Previous works suggest that multiple heuristics for complex problems can dramatically speed up the search. However, choosing the appropriate combination of heuristic functions is tricky. Current optimization approaches rely on hand-tuning the parameters via trial and error by engineers over many iterations. There is a need to reduce the difficulty of designing heuristic functions for search performance maximization.

Our first contribution is to propose an improved A* with a self-evolving heuristic function named Evolutionary Heuristic A* (EHA*) that reduces engineering effort to design the heuristic function for A* and maximize the search performance. Our experiment results show that EHA* (i) preserves path optimality; (ii) is not limited to a particular application; (iii) speeds up the path searching process; and (iv) most importantly, dramatically reduces the difficulty for software engineers to design heuristic functions for A* search. Moreover, our work can be applied to other existing works on the performance improvement of A* search.

Search, A* search suffers from poor performance on large search spaces. Although EHA*

improves the quality of heuristic functions, large search space still leads to many unnecessary searches. Our second contribution is Regions Discovery Algorithm (RDA), a map clustering technique to partition a grid based map into different categories to reduce search spaces and increase search speed. Our approach reduces the size of search spaces by partitioning a graph into many segments and identifying the segments by their characteristics. Unlike the existing approaches that might result in non-optimal solutions, our results show that RDA guarantees optimal solutions.

Our third contribution, the Hierarchical Evolutionary Heuristic A* (HEHA*), further improves the search ability of handling complex pathfinding problems and boosting the search performance, by reducing search spaces and exploiting parallelism techniques. HEHA* combines the strength of EHA* and RDA to reduce search spaces and improve search speed. HEHA* shows that it provides better search performance with less memory consumption.

Fourth, we improve and apply HEHA* to Multi-Agent Pathfinding (MAPF) problems. MAPF is the fundamental problem of many robotic and logistic applications, where the main constraint is that all agents can find the shortest paths while not colliding with each other. While the current trend favors the central controlled system, our approach is to develop a distributed version of HEHA* that can efficiently plan the optimal path for each agent. Such a system requires data sharing and exchanging among the agents, so that each agent can make its own decision without a supervising system. Our experiment results show that the Multi-Agent version of HEHA* maintains a high success rate when the number of agents increases.

While EHA* and HEHA* provide a novel approach for heuristic function design, the preprocessing times are not trivial. To boost the performance of the preprocessing steps in EHA* and HEHA*, we propose a FPGA-based reconfigurable hardware accelerator that is not bound to any specific applications as our fifth contribution. Since GA requires many independent processes, it is suitable to implement it in a hardware accelerator to gain maximum performance. We apply the following techniques to enhance performance: deep pipelining, reconfigurable computing, massive parallel processing, and degree of parallelism maximization. Our results show that the FPGA accelerator for EHA* improves the scalability, throughput, and latency.

DEDICATION

I dedicate this dissertation to my mother who love and support me in all things great and small.

ACKNOWLEDGMENTS

I would like to express my gratitude to all the people I worked with during my candidature. It was my honor to have the opportunity to work with and receive guidance from these talented individuals. This dissertation would not have been possible without the support and nurturing of my advisor, Prof. Rabi Mahapatra. He have given me so much of his time, patience, encouragnment, and his expertise that I don't think I can ever repay.

I would like to extend my sincere thanks to my committee members and collaborators. I am thankful to Prof. Aakash Tyagi for his support, suggestions, and guidance. He have helped me not only with my academic challenges, but also problems from daily bases. I very much appreciate Prof. Thomas Ioerger and Prof. Paul Gratz for the insightful feedback and expertise they have provided for my research. Their lectures have provided a solid foundation for my research. I also thank Prof Eric Jing Du from University of Florida, who has mentored and inspired me on my earlier works.

A special thank you to my friends Chris Lai, Chris Siu, Jiayi Huang, Charles Yu, and Alan Yung, who have been giving me research advices, sharpening my presentation skills, and proofreading my papers since I started the program. I am grateful to my girlfriend, Steffi, who supported and encouraged me during this difficult time. Her love has been like the sun in my life, warming my heart and keeping me away from the darkness.

Hailing from Hong Kong, I would like to use this opportunity to applaud the freedom fighters in Hong Kong, who have been risking their lives in quest of freedom. In freedom we blossom and advance. Thank you, my fellow Hong Kongers.

Finally, to my family, especially my parents, Steve and Amy, who love and support me unconditionally. Without their guidance and support, I cannot possibly complete this dissertation. They have taught how to be a decent human being. I also want to thank my sister and brother-in-law, Karen and Lockie, who support me emotionally.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Rabi Mahapatra (advisor), Aakash Tyagi, and Thomas Ioerger of the Department of Computer Science and Engineering, and Professor Paul Gratz of the Department of Electrical and Computer Engineering.

All work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by Teaching Assistantships from Texas A&M University from the Department of Computer Science and Engineering, Texas A&M University and partially supported by research grant 1416730 from National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material do not necessarily reflect the views of the NSF.

NOMENCLATURE

AI	Artificial Intelligence
BFS	Breadth First Search
DFS	Depth First Search
WA*	Weighted A*
HPA*	Hierarchical Pathfinding A*
MHA*	Multi-Heuristic A*
PDB	Pattern Databases
EA	Evolutionary Algorithm
GA	Genetic Algorihtm
MAPF	Multi-Agent Pathfinding
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
EHA*	Evolutionary Heuristic A*
PE	Processing Element
RDA	Regions Discovery Algorithm
HEHA*	Hierarchical Evolutionary Heuristic A*

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Pathfinding is a computer science problem to search for the shortest path within a graph. It is often seen in many applications including video games, robots, logistics, and navigations in the real world. Each application may have their different pathfinding objectives, which makes pathfinding a complex problem with many variations. On top of having different applications, pathfinding problems also are susceptible to different types of environments: static, dynamic, and real time. The combination of different pathfinding problems and environments leads to many challenges in this field. These challenges include long search time, large memory consumption, non-optimal paths, which draw a lot of research interests from both the industries and academics.

Although there are already many researches on boosting techniques for pathfinding algorithms, this problem still attracts the attention from researchers, looking for further improvement due to the continuous challenges. Just like there are many types of graphs, many pathfinding algorithms have already been introduced. We can summarize all the pathfinding algorithms into two techniques, uninformed search and informed search. Since there are many real world problems that can be mapped as pathfinding problems, the variants of the pathfinding problem increase accordingly. For instance, some devices may only have limited power and computational resources, thus searching for optimal paths might exceed their capability. Therefore, engineers need to design an appropriate approach based on the objective(s) of each problem in order to maximize the pathfinding performance. All of these approaches trade off one of the following: optimality, search speed, and resource consumption.

1.1 The Problem: Large Scale Complex Pathfinding Problems

Our first research challenge, referring to the complexity of paths searching on grids, is a well-known topic in the academic literature. Many techniques have been proposed, including spatial abstraction, graph pruning, precomputed paths, and better heuristic function. Each of these approaches trades-off either computing resources or path optimality for speed gain. Moreover,

pathfinding faces more complex environment nowadays that increases the design difficulty for users when applying the techniques.

Our second research challenge is that A* search suffers from large scale search problems. For many applications that require real time computation, A* is not fast enough to explode the entire search space to return a path. It is necessary to reduce the search space before any search can happen. Many existing approaches would divide a graph into many subgraphs of even sizes and eliminate some of the subgraphs. While evenly dividing a graph can be done quickly and efficiently, it might not be the best partitioning technique because it can cause pathfinding algorithms to return non-optimal solutions.

Our third research challenge, relating to the difficulty to design a high quality heuristic function, is one of the major research topics in pathfinding algorithms. The proposed techniques in this domain are usually domain-specific, including multiple heuristic functions and memory heuristic. While the approaches are powerful, they complexify the heuristic function design process, which is already a difficult task. Multiple heuristic functions for pathfinding algorithms do not only require more computing resources, but also they are more difficult for users to calibrate the design parameters. Memory heuristic approaches store some of the precomputed paths to reduce the estimated error and computation time, however, they are resource hungry techniques that constantly require recomputation as the environment changes.

In our fourth research challenge, we visit a different but related issue called Multi-Agent Pathfinding (MAPF) problem. MAPF problems are commonly seen in video games, robotics, and warehouse logistics. The objective of MAPF is to find the possible shortest path for each agent without colliding with each other. The heuristic function of each agent needs to consider the consequences of their actions and how they affect each other. Without a centralized control system, the agents have to communicate and exchange information in order to maximize the performance.

Search speed is always the primary concern of many applications and speeding up the search process should not come with any compromise in optimality. Existing approaches usually involve preprocessing. If the preprocessing time is too long and the memory consumption is too high,

it might not be worth to pay such a high price for the rewards. Moreover, mobile robots do not usually have as much computation resources to compute and store the preprocessing results. Our final research challenge is the scalability issue arisen from algorithms that causes long execution time and high power consumption.

1.2 The Solution: Problem Complexity Reduction

Efficient complexity reduction techniques are urgently needed to keep up with the fast growing industries that encounter bigger and more complex pathfinding problems everyday. Traditional pathfinding algorithms like A* are known for the lack of scalability. Moreover, the performance of A* heavily depends on the quality of the heuristic function. A heuristic function determines the number of states to visit during each search, hence affects the execution time and memory consumption. A* with a well-designed heuristic function that works fast and generates high-quality solutions, even when it comes to high-dimensional problems. However, designing a heuristic function by hand that accurately estimates the cost in a complex environment is extremely difficult, not to mention time-consuming. Furthermore, A* algorithm is known for guaranteed completeness and optimum, thus ensuring heuristics are admissible and consistent is necessary. To overcome these challenges mentioned in the previous section, this dissertation proposes three algorithm solutions and one architecture solution as an attempt towards efficient pathfinding with high scalability and search performance.

1.2.1 Self-Evolving Heuristic Function

Designing heuristic functions for pathfinding problems is a difficult task. Users are required to have expert knowledge of the environment and the ability to convert such knowledge into mathematical functions for A* algorithm to accurately estimate the distance between a starting point and a goal. The designing process requires a long period of trial and error to calibrate the parameters of heuristic functions. This dissertation proposes the Evolutionary Heuristic A* (EHA*) algorithm, a heuristic evolving approach to minimize a user's design effort while increasing the quality of heuristic functions.

1.2.2 Search Space and Complexity Reduction

There are two factors in pathfinding problems which affect the search difficulty: the search space scale and the degree of freedom of a problem. The former increases the depth of a search tree, which is the length of a solution. The latter increases the breadth of a search tree, which is the number of moves that are available for each step. Both factors have to be addressed in order to maximize the search speed. To reduce the search space, this dissertation proposes a novel map clustering technique named Regions Discovery Algorithm (RDA) to remove unnecessary space. To reduce the breadth of a search tree, an algorithm cannot simply reduce the degree of freedom, instead, it should carefully evaluate the available actions based on the given situations. This dissertation proposes a hierarchical pathfinding algorithm named Hierarchical Evolutionary Heuristic A* (HEHA*) that combines and improves the previous two works, EHA* and RDA, to allow more accurate estimating during search, hence leads to better performance. To prove that HEHA* is capable of solving complex pathfinding problems, we propose a multi-agent version of HEHA* (MA-HEHA*) to solve multi-agent pathfinding (MAPF) problems with growing numbers of agents.

1.2.3 High Performance Computing

Same as many pathfinding performance enhancement approaches, the works this dissertation proposes require data preprocessing. Mobile robots that need pathfinding algorithms to complete their task usually do not have enough power or resources to execute the program in a long run. These applications need a domain-specific hardware accelerator that reduces the power, area, and memory consumption, while providing high scalability and high performance. This dissertation proposes a FPGA accelerator architecture for EHA* to speed up the heuristic function evolution process. This architecture takes the advantage of simple digital logic circuit design, high degree of parallelism, and deep pipelining to improve the preprocessing latency and throughput. Moreover, the process elements are independent of each other, thus opening up opportunities for achieving large scale and high performance computing.

1.3 Thesis Statement

Self-evolving heuristic function for A* is needed to improve and optimize search performance for complex pathfinding problems.

1.4 Research Contributions

This dissertation addresses the challenges including the algorithm scalability, search performance, and computational resources consumption due to the increasing complexity of pathfinding problems. The main contributions of this research are the following.

1. This research illustrates the challenges and difficulties of heuristic functions design for complex pathfinding problems. The proposed self-evolving heuristic function for A* algorithm minimizes the user's effort to design and calibrate complex heuristic functions. The proposed EHA* algorithm successfully calibrates and balances multiple heuristic functions to achieve better search performance while guaranteeing solution optimality. This self-evolving heuristic function approach can be applied to other A* improvement techniques for search performance enhancement.

2. This research is the first study that introduce the idea of self-evolving heuristic function for pathfinding algorithm. The proposed work including EHA*, HEHA*, and MA-HEHA* recognize the need of a specific tailored heuristic function for each pathfinding problem. By learning the environment with an evolution engine, self-evolving heuristic functions are now more accurate and able to provide better search performance solving complex pathfinding problems.

3. This research reduces the search spaces and search complexity of pathfinding problems by introducing a novel map clustering algorithm to identify and remove unnecessary search spaces. The proposed works, RDA and HEHA*, are preprocessing techniques that learn grid maps based on the local features and construct abstract maps to reduce search space while preserving solution optimality. While the map clustering technique is specifically designed

for grid maps, the general idea of map abstraction and search space removal can be applied to other search domains as well.

4. This research identifies the parallelism opportunities that are unique to the Genetic Algorithm and proposes a FPGA accelerator architecture for large-scale distributed genetic algorithm that exploits parallelism and deep pipelining. The proposed FEHA* accelerator massively increases throughput compared to the software version of EHA*, while showing promising potential to increase the scalability for large scale genetic algorithm. The simple modularized processing elements minimizes latency and increases pipeline depth, which leads to higher throughput as a result.

1.5 Dissertation Organization

The remainder of this proposal is organized as follows.

- Chapter 2 reviews the related work on pathfinding algorithms and provides the background for the main chapters.
- Chapter 3 presents the Evolutionary Heuristic A* algorithm. The contributions from this chapter have been previously reported in [Yiu, Du, Mahapatra, AIKE 2018; Yiu et al., IJSC 2019] [1] [2].
- Chapter 4 illustrates the clustering algorithm named Regions Discovery Algorithm (RDA) that partitions the search graph into multiple segments to speed up the pathfinding process. The contributions from this chapter have been previously reported in [Yiu and Mahapatra, TransAI 2020] [3].
- Chapter 5 shows the Hierarchical Evolutionary Heuristic A* algorithm. The contributions from this chapter have been previously reported in [Yiu and Mahapatra, HCCAI 2020] [4].
- Chapter 6 presents the distributed EHA* tackling MAPF problems. The contributions from this chapter is recently submitted and currently under review in [Yiu and Mahapatra, submitted for publication] [5].

- Chapter 7 illustrates the FPGA accelerator architecture to enhance the performance of Evolutionary Heuristic A* algorithm. The contributions from this chapter is recently submitted and currently under review in [Yiu and Mahapatra, submitted for publication] [6].
- We conclude this dissertation and summarize future work in Chapter 8.

2. BACKGROUND AND LITERATURE REVIEW

In this chapter, we would like to introduce some basic concepts about pathfinding problems and prior researches on pathfinding algorithms that are related to our work which we hope that it will help readers to have a better understanding of the rest of the discussion.

2.1 Pathfinding Problems

The objective of pathfinding problems is to plan a route from an initial location to a goal location in an environment [7]. These problems can be found in many real-life applications, such as video games, robotics, logistics, crowd simulation, motion planning, and vehicle routing. In general, the research directions in pathfinding problems can be divided into two steps:

1. Graph representations are methods about how to construct a search graph from a real world environment.

2. Pathfinding algorithms are responsible to guide a search agent to search the path it wants in this environment

2.1.1 Types of Search Graphs

To construct a search graph from a real world environment, we often need to map the objects in the environment into different categories. Depending on the objectives of the applications, the categories can vary as well. However, in general, we choose to simplify the environment as much as possible because the applications are unlikely to need all this information. It is similar to a map that represents a city. While a city has different landforms, buildings, facilities, and roads with different widths, all this information is redundant to the map users because their objective is to find the correct path to the destinations. This is the same for graph representations in pathfinding problems.

While there are many types of graph representations that are used in pathfinding problems, there is no best graph representation. Algorithm designers have to decide which graph type fits

the best based on the different features of the environment. In this dissertation we study how heuristic function affects pathfinding performance in the context of a two-dimensional grid based environment. This particular format is chosen because of its simplicity and popularity. Grid based graph is often used in video games, whether it is built with square, triangle, or hexagon cells. These cells are either classified as traversable (walkable area) or non-traversable (obstacles) [8]. While our principal results are derived in the context of two-dimensional static maps, they are also applicable to other pathfinding application domains and other environments such as GPS navigation, robot motion planning, logistic planning, or any other problems that can be framed as a graph search problem.

2.1.1.1 Grid Maps

Grid maps are popularly applied in many applications for the following reasons: (i) they are simple to understand and construct; (ii) their simple format leads to low memory consumption; (iii) it has high flexibility that the grid resolution can be changed for different maps. However, grid maps suffer from the disadvantages of enormous space and time complexity. The problem with the grid is that the cells have uniform size and they are often small to maintain the resolution and the detail of the map. Since the cells are tightly connected with each other, many cells are needed to explore before a path is being found. It is the first priority of a pathfinding algorithm to reduce the search time, while maintaining the optimality.

2.2 Pathfinding Algorithms

Pathfinding algorithms search a graph and find a path with the lowest cost by starting at a given node as a starting point and exploring adjacent nodes until the destination node is reached. Pathfinding problems often have to be solved under many constraints including limited processing time, memory, and computational power. In general, pathfinding algorithms are categorized into uninformed search and informed search.

2.2.1 Uninformed Search

Uninformed search is also called blind search and brute force search [9]. As the names stated, uninformed search strategies have no knowledge about states other than information that is provided in the problem definition. Uninformed search generates successors and searches an optimal path to a goal state from the root node, which is also called the starting node. There are different types of uninformed search that the differences are the order in which nodes are expanded.

Breadth-first search (BFS) [10] is one of the most simplest pathfinding strategy that searches all possible moves before expanding to the next level. Starting from the root node, BFS expands all the successors, and then expands the next depth level, which are the grandchildren of the root node until the goal node is discovered. In general, all the nodes at a given depth have to be expanded in a search tree before the next level nodes can be expanded. The time complexity of BFS is $O(b^d)$, where b represents the breadth of the search tree and d represents the depth of the search tree.

Depth-first search (DFS) [11] is a backtracking technique which has also been widely used for pathfinding problems in many domains. Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search backtracks to the next deepest node that still has unexplored successors. The time complexity of DFS is $O(b^m)$.

2.2.2 Informed Search

A* search algorithm [12] is one of the best and most popular techniques used in low-dimensional pathfinding because of its efficiency and guarantee for completeness and optimality. However, A* search suffers from low performance and resource-hungry on high-dimensional search problem, such as motion planning for robotic arms and autonomous vehicles. It is not trivial to efficiently find an optimal solution on high-dimensional problems. The performance and reliability of A* search greatly rely on the accuracy of the heuristic functions. Designing an accurate heuristic function for these problems could be a time consuming and complex task. Therefore, manually

designing heuristic functions that are both admissible and consistent is still one of the biggest challenges for complex pathfinding problems.

2.2.3 A* Algorithm Enhancement Techniques

A heuristic function controls the behavior of A* search by estimating minimum cost from any vertex n to the goal. A* search estimates minimum cost by using the equation $f(n) = g(n) + h(n)$. Therefore, heuristic design requires careful consideration and significant amount of time. Heuristics often suffer from overestimation or underestimation, which degrades the effectiveness of A* search. An inaccurate heuristic function leads to unnecessary priority queue operations and nonoptimal solutions. The quality of heuristic functions affects both the accuracy and search speed.

*2.2.3.1 Weighted A**

Weighted A* [13] is one of the most well-known A* variants that sacrifices optimal solutions for search speed-up. As the name states, it is based on A* search and it adds a weight onto the evaluation function f(n) = g(n) + h(n), where g(n) represents the cost from the starting point to the node n, h(n) is the estimation of the true cost from node n to the goal, calculated by the heuristic function. The weight is added to multiply the h(n) so that the evaluation function for weighted A* becomes f(n) = g(n) + w*h(n), which the weight w is a real value greater than one. Heuristic inflation is a simple but powerful technique for performance boosting. By inflating the estimated cost, it speeds up the search for a solution in most scenarios. Generally, the larger the w value is, the faster weighted A* can search. Figure 2.1 shows the impact of weight to the search behaviors and solutions quality of WA*. While increasing the weight reduces the number of expansions, the accuracy of the solution decreases as well. The characteristics of WA* makes it popular for many time constrained applications, including motion planning, map navigation, and robotics. Although WA* reduces the computation time considerably, it is difficult to decide the weight value that provides the perfect balance between speed and accuracy.

While WA* shows its practical use on motion planning and robotics, Wilt et al. [15] showed that there are several occasions where WA* and its variants failed to accomplish their goal. WA*

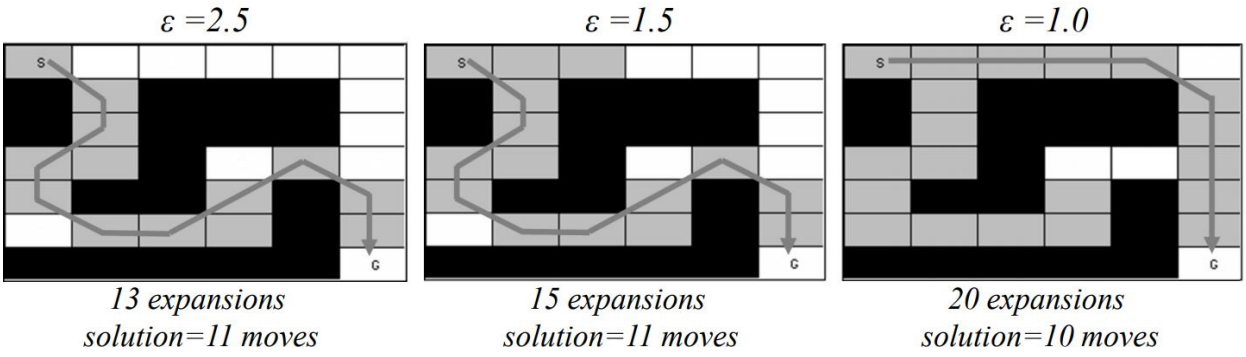


Figure 2.1: The impact of weight w to Weighted A* search performance. Higher weight value leads to faster computation but a less accurate result. Reprinted from [14].

was developed under the assumption that the increment of weight leads to faster searches. As the weight gradually increases, WA* search becomes a greedy-best-first search. However, the greedy algorithm does not necessarily speed up the search for several reasons. First, the greedy algorithm does not perform well in certain domains, such as inverse tile and top spin problems. Second, the greedy search performs poorly when there are many local minima, or when the local minimum is large. Third, when the heuristic function is poorly designed, inflating the heuristic function will degrade the search performance greatly [15].

To address the shortcomings of WA*, Likhachev [14] developed Anytime A* (ARA*) based on WA*. First ARA* finds a feasible solution in a short time, then recomputes and improves the existing solution within the deliberation time. ARA* first starts with a large weight to perform a quick search to obtain an initial solution. Once a solution is found, it decreases the weight and recomputes for a better result until the given time is over. This algorithm guarantees a suboptimal solution and continuously improves when enough time is given.

2.2.3.2 Pattern Database

Pattern databases (PDBs) are heuristic functions implemented in the form of lookup tables. The optimal solutions of subproblems of the original problem are stored in PDBs. By precomputing these solutions of subproblems, PDBs consume extra memory and preprocessing time for speed-up. The main drawback of PDBs is that it requires an enormous amount of memory to store all the

solutions of subproblems. Due to the memory constraints, algorithm designers often have to choose only a small amount of patterns from the problem. This technique is often used for approximate pathfinding techniques: by precomputing some of the shortest paths between certain points in the graph, part of the search can be replaced by using lookup tables. One of the examples is Landmark Pathfinding between Intersections (LPI) [16]. The basis of LPI is to place landmarks at vertices throughout the graph and precompute the shortest path from itself to all other vertices. LPI reduces the searching time by giving up the solution optimality.

2.2.3.3 Hierarchical Pathfinding

Hierarchical approaches introduce an idea of searching solution paths on an abstract level and then refining on the lower level. An abstract graph is a reduction of the full state graph, where each node represents one or more states in the lower level graph, and an edge exists between two nodes if there is any operator that can be applied to any state abstracted by the first node which will take you into any other state abstracted by the second node [17]. An approximate solution can be obtained quickly by searching from the abstract space. For instance, consider the problem of planning a road trip between two cities: Instead of searching for all roads between two cities, finding a highway that connects two cities would probably be an easier and more intuitive approach. Then we look into more detailed planning, including how to get to the highway and when to exit the highway. Hierarchical pathfinding works exactly like this. We first construct an abstracted solution from the abstract map, then refine the solution so that it becomes nearly optimal.

Hierarchical Pathfinding A* (HPA*) [18] [19] is one of the most famous hierarchical pathfinding algorithms. The algorithm starts with creating an abstract map by dividing the map into many local clusters and adding extra vertices to connect each cluster. The extra vertices are precomputed before any searches begin, so that they can be reused during the search. Therefore, by combining the technique of PDBs and A* searches in multiple levels of abstractions, the complexity of pathfinding is decreased and divided into multiple searches. During the search, the first step is to find an abstract solution from the starting zone to the goal zone. Then the agent searches from the starting node to the node that leaves the starting zone. In the end it searches for the path from the

entrance of the goal zone to the actual goal. While this is not able to return an optimal solution, this approach shows a great reduction of search effort.

While HPA* uses a simple clustering method to minimize the preprocessing, the clusters it created might not fully exploit the advantage of hierarchical pathfinding. Since it requires no domain specific knowledge other than the width and height of the map, it can only divide the map with the simplest method into N by M subgrids that each equal-sized subgrid. It is indeed a fast clustering method since it does not require any learning, however it might increase the preprocessing time for abstract vertice creation of the abstract map.

A similar approach has been introduced on different graph representations. Hierarchical pathfinding for Navigation Meshes (HNA*) [20] [21] adopted the similar idea of map abstraction onto Navigation Meshes instead of grid based map. It uses a multilevel k-way partitioning algorithm to create the hierarchical representation. However, it shows that under certain configurations, the performance of hierarchical pathfinding approaches could be worse than A* search.

*2.2.3.4 Multi Heuristics A**

As the name states, Multi-Heuristic A* (MHA*) [22] is when A* search uses multiple heuristic functions simultaneously to search for suboptimal paths. It starts with a consistent heuristic function, meaning the heuristic never overestimates the actual step cost, as the main guidance, and uses multiple inadmissible heuristics that each can provide complementary guiding power. These additional heuristics cannot search for a path when used in isolation, but they can help the main heuristic yields better results. Compared to traditional pathfinding algorithms, the advantage of MHA* is that the guiding power of multiple heuristic functions is stronger, which leads to better search performance. The work from Islam [23] proves that MHA* is not only necessary, but also highly efficient for solving high-dimensional, constrained pathfinding problems. However, applying multiple heuristic functions increases the computation and memory consumption. Moreover, MHA* suffers from a serious calibration problem [24] because each heuristic function weighs differently in different scenarios. Without proper calibration, some useful heuristics might be ignored during the search, resulting the MHA* is being reduced to the ordinary A* search. To overcome

the calibration issue, Islam proposed dynamic MHA* [25] that generates heuristics dynamically on-the-fly when necessary.

2.2.4 AI-Based A* Enhancement Technique

MHA* shows that a single heuristic function is not enough to efficiently solve complex problems. It also points out the difficulties to design and calibrate heuristic functions for maximum performance. Researchers seek to apply ML approaches to learn efficient heuristic functions.

2.2.4.1 Deep Neural Network as Heuristic Function

Several works are proposed to use DNN to replace heuristic functions [26] [27]. Using multiple inadmissible heuristics as features, a DNN can combine the strengths of all heuristics and result in a single efficient heuristic. Another approach uses RL to train the neural network instead of using heuristics as features [28]. The advantage of the deep learning approach is that the features are no longer bounded by the performance of the given heuristics. Instead, the heuristic is learned from the given environment. However, most of the pathfinding applications demands high energy-efficiency because of the limited power budgets and computing resources, DNN and RL could be too computational intense to be practical.

2.2.4.2 Genetic Algorithm Versus Reinforcement Learning

Recent researches show that Genetic Algorithm (GA) [29] provides better results with faster execution time compared to RL. GA is considered the most biologically accurate evolutionary model. It is a search-based optimization technique based on the principles of evolution and natural selection. Instead of learning with gradients like RL, GA evolves the solution over iterations. It starts with a competition between a population of randomly initialized solutions. These solutions then go through generations of selections, crossover recombination, and mutation to produce offspring solutions. This process is repeated until it yields the fittest solution that fulfills the given requirements. Deep Neuroevolution [30] proved that GA can be a competitive alternative to RL, and GA has advantages over RL on simplicity and energy efficiency. However, evolving a complex heuristic function can take some time in the preprocessing phase.

2.3 Summary

In this chapter we have reviewed some basic concepts of about pathfinding problems and prior researches on pathfinding algorithms that are related to our work. We also reviewed several AI-based enhancement techniques for A* algorithm.

Algorithm	Pros	Cons
A*	Guaranteed optimal solutions. Performs provably minimal number of state expansions required to guarantee optimality.	Long searching time. High memory usage.
WA*	Trades off optimality for speed. Usually faster than A* in many applications. Low memory usage.	Optimality is not guaranteed. Could be slower than A* for some problems.
ARA*	Dynamically adjusting the weight for the best suboptimal solution. Best suboptimal solution from a given search time.	Optimality is not guaranteed. Could be slower than A* for some problems. High memory usage.
MHA*	Guaranteed optimal solutions. Capable to handle complex problems with multiple heuristic functions	Extremely high memory usage. Depended on parallel computing. Long searching time.
PDBs	Faster computation. Guaranteed optimal solutions	Precomputation required. High memory consumption.

Table 2.1: Comparisons between previous heuristic search algorithms including A*, WA*, ARA*, and MHA*

Table 2.1 summarizes the search algorithms discussed above, along with their strengths and weaknesses. Each algorithm has its advantages in various domains, which it is not trivial to determine whether one is better than the other. For instance, A* guarantees the optimality of the solutions, however, the time and space complexity are high. There are many time bounded appli-

cations that cannot provide enough time for A* search to obtain an optimal solution. MHA* has the similar properties with A*; it is able to provide optimal solutions, but it pays a high cost on both computational power and memory usage. Both WA* and ARA* achieve better search performance, but they cannot guarantee to provide optimal solutions. We aim to improve the efficiency and accuracy of A* search using a different approach by seeking solutions in an AI perspective.

3. EVOLUTIONARY HEURISTIC A* SEARCH [1]

A well-designed heuristic function is essential for the A* search to obtain solution without overly exploring the state. Researchers focus on resolving these challenges by either trading solution quality for faster computation times or using computational resource intensively to compute optimal solution. Neither of the approaches directly resolve the heuristic function design problem. Aine et al. [22] presented the multi-heuristic search framework (MHA*) that uses multiple inadmissible heuristic functions as a complementary role to the admissible heuristic function to simultaneously explore the search space. While this approach is simple and powerful, the algorithm is resource-intensive. To reduce the computational time, Pohl [13] proposed the original idea of the weighted A* (WA*) for faster computation. However, the solution path discovered by the WA* may not be optimal. While both methods require accurate heuristic functions for optimal performance, they can be difficult to manually design due to the planning problem complexity.

In this chapter we present the Evolutionary Heuristic A* search (EHA*), which uses Genetic Algorithm (GA) to automatically design, calibrate, and optimize Multi-Weighted Heuristic functions (MWH) to maximize heuristic search performance. Compared to designing heuristic functions that are admissible and consistent, one can easily compute approximate heuristics and rely on EHA* to adjust heuristics until the most accurate heuristic function is generated. First EHA* generates a certain number of agents, each of which contains random heuristic functions with randomly initiated weight for each heuristic. During the optimization process, each agent competes with each other and reproduces better children. When the optimization is completed, the algorithm returns the optimized heuristic function set.

We first describe the EHA* properties, which proves the optimality of MWH, the complete-

[1] Reprinted with permission from "Evolutionary Heuristic A* Search: Heuristic Function Optimization via Genetic Algorithm." by Ying Fung Yiu, Jing Du, and Rabi Mahapatra, 2018. 2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), pp. 25-32, DOI: 10.1109/AIKE.2018.00012, Copyright ©2018 IEEE; Reprinted with permission from "Evolutionary Heuristic A* Search: Pathfinding Algorithm with Self-Designed and Optimized Heuristic Function." by Ying Fung Yiu, Jing Du, and Rabi Mahapatra, 2019. International Journal of Semantic Computing, 13(01), pp.5-23, DOI: 10.1142/S1793351X19400014, Copyright ©2019 World Scientific Publishing Co Pte Ltd

ness of EHA*, and the performance of the optimization technique. Then we present experimental results for applying the proposed algorithm to different types of grid-based pathfinding benchmarks, provided from Nathan R. Sturtevant [31]. These experiments demonstrate the simplicity and effectiveness of EHA*, especially for complicated planning problems where it is difficult to design precise heuristic functions to solve. Our experiment evaluation metrics includes the number of iterations to optimize the heuristic functions, the maximum size of the priority queue, and the accuracy of the heuristic functions. Our contributions of this paper are summarized below:

1. We propose Evolutionary Heuristic A* search (EHA*) to automatically optimize heuristic functions and calibrate their weight values. EHA* minimizes the amount of time on heuristic functions design.

2. EHA* is capable to design complex Multi-Weighted-Heuristic functions (MWH) with great performance.

3. EHA* guarantees completeness and optimality.

4. Optimal solution paths can be found by EHA* with minimal computational cost.

3.1 EHA* Algorithm Overview

One of the most important features of EHA* is that it optimizes heuristic functions through GA. GA was first introduced by Holland [29], which was inspired by biological evolution. The purpose of GA is to solve both constrained and unconstrained optimization problems based on competition and natural selection. Over successive generations, the population evolves toward an optimal solution. The algorithm first generates random parameters for the agents to obtain initial solutions. After solutions are found, the cost of each solution is evaluated and ranked. The child with lowest cost is considered as an elite individual and its chromosome will be passed to the next generation as an elite child. Besides the elite child selection method, GA uses the individuals in the current generation to reproduce children via mutation and crossover.

Mutated children are generated by applying random changes to a single individual in the current generation to create a child. Mutation does not generally advance the search for a solution, but it does provide insurance against the development of a uniform population incapable of further evolution [29]. The rest of the children are generated by crossover. The purpose of crossover in GA is to test new parts of target regions rather than testing the same string over and over again in successive generations [29]. Both methods are essential to GA because they create diversity of a population and extract the best genes from different individuals to increase the possibility for GA to optimize the solution. By applying GA to A* search, the heuristic function design process can be more efficient and precise than the traditional methods. In the following section, we will analyze the methodology of EHA* algorithm, the design of the gene pool, and the heuristic function optimization models.

We start by presenting the optimization models for the heuristic function in EHA*. An agent contains a Multi-Weighted-Heuristic (MWH) function that is formed by four heuristics selected from the gene pool. The gene pool contains multiple simple heuristics, each provides different ways to estimate a cost from the current state to the goal state. Inadmissable heuristics are allowed to be included in the gene pool. The inadmissible heuristics play a complementary role that only helps in a specific situation. By combining the heuristics into a complex heuristic function, it becomes more informative and is able to provide a more accurate estimation compared to a single heuristic function. Each heuristic in the gene pool estimates the cost differently. For more complex problems, more heuristics are needed to be added into the gene pool to handle different situations. We have eight heuristics in the gene pool for our experiment to reduce the optimization complexity due to the many combinations of MWH.

3.1.1 Multi-Weighted-Heuristic (MWH) function

Figure 3.1 shows the MWH function, denoted as $H(n)$, is the sum of the weighted heuristic, with the formula $H(n) = w*h(n)$. By combining multiple weighted heuristics together, the MWH becomes more informative and accurate where each heuristic estimates the cost $c(N, G)$ between the node N and the goal G in a different domain. Unlike the traditional WA* search, the weight

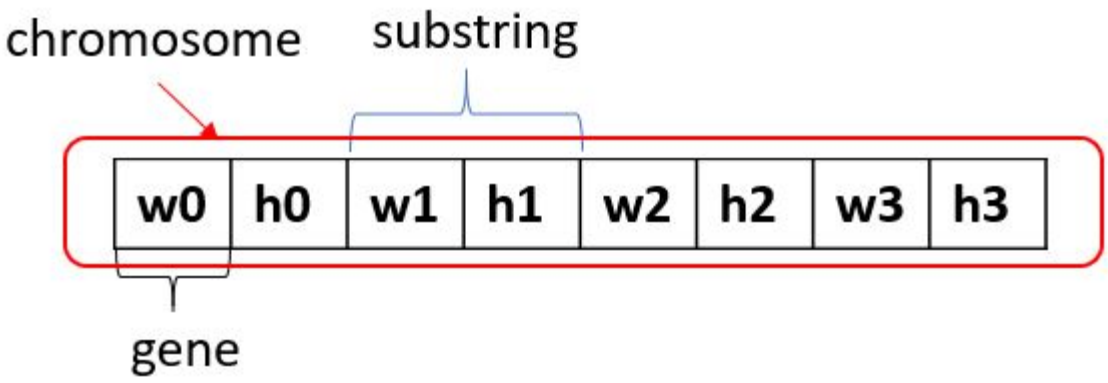


Figure 3.1: Chromosome of an agent. The chromosome is a numerical representation of a MWH function. Each heuristic function contains multiple weight and heuristic. As the figure shows, a gene is a single parameter, whether a weight value or a heuristic index. A substring is a bundle of a heuristic index and a corresponding weight value.

value in MWH serves the purpose to prioritize each heuristic. However, it is time consuming to design MWHs to choose the correct combination of heuristics and the appropriate ratio of weights. We present three EHA* models that differently optimize MWH: Base Model, Parallel Islands Model, and Re-Initialization Model. The implementation detail of the characteristics from each model to optimize the heuristic function is presented in the following section.

3.1.2 EHA* Implementation Models

Algorithm 1 Pseudocode of EHA*

Input: Population Size s, Chromosome Length l
Output: Best Chromosome c
procedure EHASTAR(s, l)
    $p = InitPopulation(ps, l)$ ▷ Population initialization
    while $!StopCond()$ **do** ▷ Stop after n generation
        $results = AStar(p)$ ▷ Use the population to run EHA* search
        $p = EvalSol(p, results)$ ▷ Sort the population based on the results
        $c = p[0]$ ▷ Get the best candidate
        $p = GenNext$(p) ▷ Create next gen using Genetic Algorithm
    end while
    return c ▷ Return chromosome with the best performance
end procedure

Table 3.1: Gene pool and its heuristic functions. Each heuristic estimates the cost in a different way. By combining multiple heuristics to generate a multi-heuristic function, it can provide a more accurate estimation that leads to search performance boost

Index	Heuristic
0	Difference between X-coordinate
1	Difference between Y-coordinate
2	Sum of Manhattan distance
3	Sum of Euclidean distance
4	Sum of Diagonal distance
5	Sum of Manhattan distance, ignore shallow water
6	Sum of Euclidean distance, ignore shallow water
7	Sum of Diagonal distance, ignore shallow water

We choose to use 20 agents in our experiment for the convenience of experiment setup. The number of agents can be changed depends on different problems. The Base Model uses only selection, mutation, and crossover between n agents to optimize the heuristic function. The pseudocode for the Base Model of EHA* is illustrated in Algorithm 1. Each agent uses different MWH functions simultaneously to independently search the solution path. Therefore, each search has its own priority queue and different h values. The default setting of these MWH functions are initialized randomly by picking four heuristics and four weights from the gene pool shown in Table 3.1. After all the solutions are found, EHA* ranks the solutions based on the solution path length and the priority queue size. The top two agents will be selected as the elite individuals and the rest of the population are formed with nine mutated children and nine crossover children. Algorithm 2 presents the psudocode of children generation of the Base Model. The ratio is used to maximize the optimization speed by mutating or crossovering most of the agents. Mutated children are reproduced from any of the agents to avoid bias. Each mutated child inherits the mutated heuristic function from its parent. The mutated heuristic function is generated by randomly replacing one of the genes instead of a single bit from the chromosome, this provide higher flexibility for what the new value of the gene will be in the next generation.

Algorithm 2 Pseudocode of Children Generation of the Base Model

Input: Population p
Output: Population p
procedure GENNEXT(p)
    for $ind = 2 : p.size()$ **do** ▷ p[0:1] are Elites, no modification needed
        if $ind >= 3 \&\& ind <= 11$ **then** ▷ Mutated Children
            $p[random()].gene[random()] = random()$ ▷ randomly mutate a gene
        else if $ind >= 12 \&\& ind <= 20$ **then** ▷ Crossover Children
            parOne = p[$random()$]
            parTwo = p[$random()$]
            while $parOne == parTwo$ **do** ▷ Avoid parents are the same agent
                parTwo = p[$random()$]
            end while
            for $s = 0 : substr.size()$ **do**
                if $random()\%2$ **then** ▷ Choose substring randomly
                    $p[ind].substr[s] = parOne[random()].substr[random()]$
                else
                    $p[ind].substr[s] = parTwo[random()].substr[random()]$
                end if
            end for
        end if
    end for
    return p ▷
end procedure

Algorithm 3 Pseudocode of Children Generation of PIM

Input: Population p, Generation g
Output: Population p
procedure GENNEXT(p)
    for $ind = 2 : p.size()$ **do** ▷ p[0:1] are Elites, no modification needed
        if $g\%50$ **then** ▷ For every 50th generations, inter-islands crossover
            parOne = p[$random()$]
            parTwo = p[$random()$]
            while $parOne == parTwo$ **do** ▷ Avoid parents are the same agent
                parTwo = p[$random()$]
            end while
            for $s = 0 : substr.size()$ **do**
                if $random()\%2$ **then** ▷ Choose substring randomly
                    $p[ind].substr[s] = parOne[random()].substr[random()]$
                else
                    $p[ind].substr[s] = parTwo[random()].substr[random()]$
                end if
            end for
        else
            if $ind >= 3 \&\& ind <= 11$ **then** ▷ Mutated Children
                $p[random()].gene[random()] = random()$ ▷ randomly mutate a gene
            else if $ind >= 12 \&\& ind <= 20$ **then** ▷ Crossover Children
                parOne = p[$random(withinIsland)$]
                parTwo = p[$random(withinIsland)$]
                while $parOne == parTwo$ **do** ▷ Avoid parents are the same agent
                    parTwo = p[$random(withinIsland)$]
                end while
                for $s = 0 : substr.size()$ **do**
                    if $random()\%2$ **then** ▷ Choose substring randomly
                        $p[ind].substr[s] = parOne[random()].substr[random()]$
                    else
                        $p[ind].substr[s] = parTwo[random()].substr[random()]$
                    end if
                end for
            end if
        end if
    end for
    return p ▷
end procedure

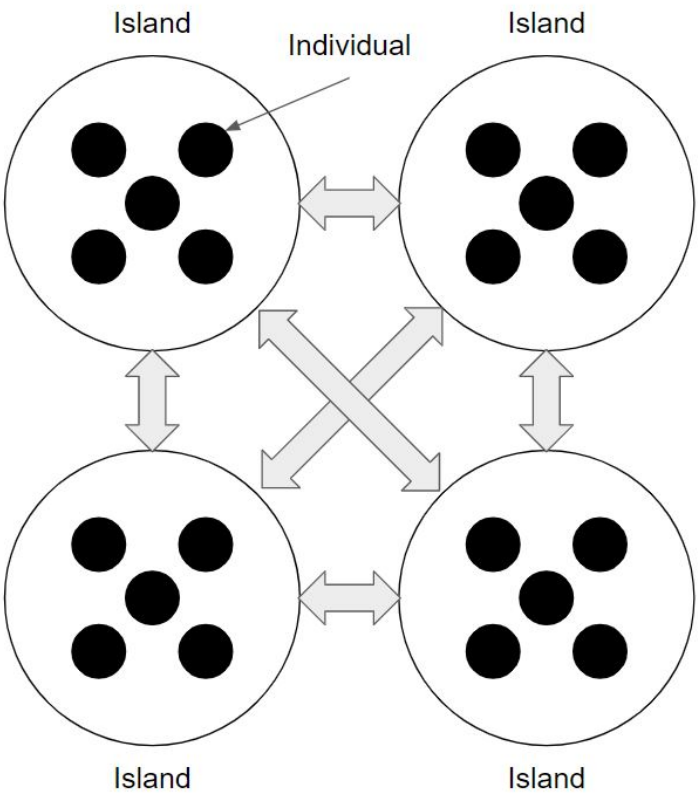


Figure 3.2: Graphical representation of Parallel Islands Model

Crossover children are reproduced based on the following rules: one child is identified as the elite crossover child, as it is generated from the elite parents, while the rest are generated from random parents to avoid trapping in the local minima. A crossover child is reproduced by randomly selecting two substrings from each parent. A substring is a bundle of a heuristic index and its corresponding weight value. It is essential to use a substring instead of a gene to reproduce a crossover child that inherits the advantages from the parents. After generations of selections, mutations, and crossovers, the heuristic function will evolve to an optimal formation that is able to provide the most accurate h value for an A* search. While the Base Model is able to optimize the heuristic function, it has difficulty of escaping the local minima in complex optimization problems. To demonstrates there are improvement for the Base Model, we present two additional models, the Parallel Islands Model and the Re-Initialization Model, that increase the degree of randomness during reproduction to reduce the optimization time.

When a search problem faces many local minima or plateau, EHA* should allow a weak chromosome to be inherited for several generations instead of eliminating it in an early stage. However, a weak chromosome has little or no chance to survive over generations in the current Base Model. We propose to embed the Parallel Islands Model (PIM) into EHA* algorithm, which increases the search diversity and the ability to escape the local minima. In PIM, populations are equally distributed into g groups as Figure 3.2 illustrated. Individuals within each group crossover reproduce with each other during normal operations, while the inter-islands crossover reproduction only happens every N^{th} generations. In the regular reproduction phase, each island has one elite child and the rest are mutated or crossovered. In every fiftieth generation, other than two elite children, the rest of the children are reproduced by inter-islands crossover. Consequently, a new generation is formed with two elite children, and eighteen inter-islands crossover children. All parents used for crossover are randomly chosen from the previous generation. Algorithm 3 presents the pseudocode of the children generation of PIM.

We also present a special case of PIM, named Elite Island Model (EIM), where one of the islands is formed by only elited children. Figure 3.3 presents the idea of elite island. We believe the optimization process will be improved by grouping the best evolved agents in each island. In EIM, we split the agents into five groups with four agents in each island. The top four agents will be placed in the elite island. The rest of them will be reproduced via mutation and crossover and redistributed to the rest of islands randomly. The advantage of both the PIM and EIM enhance the opportunity to optimize the heuristic function when there are many local minima. Yet, the Base Model, PIM, and EIM rely on the quality of the initial population which can affect the efficiency of the models. To reduce the impact of the initial population, new agents should join into the competition pool every N^{th} generation.

Re-Initialization Model (RIM) is designed to minimize the impact caused from the initial populations. By flushing the old populations other than the elites and replacing new populations to the competition pool, RIM increase the search diversity which avoids trapping in local minima. It is essential to combine RIM with EIM in order to increase the survival rate of the new agents. Same

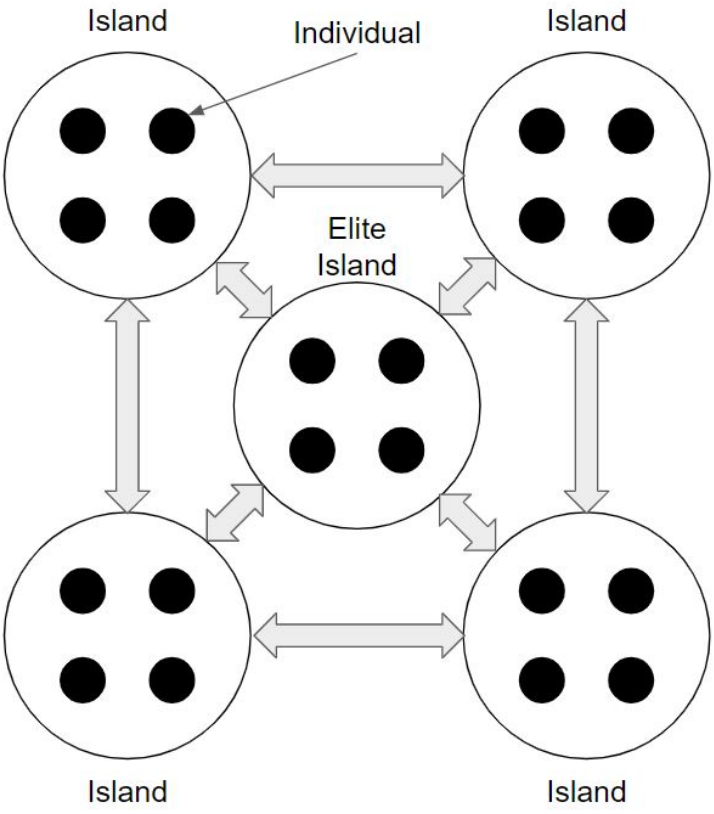


Figure 3.3: Graphical representation of Elite Islands Model

Table 3.2: Population distributions for each EHA* model of a population of 20. For the PIM, EIM, and RIM, the population distribution changes for every 50th generation.

	Base	PIM		EIM		RIM	
		Reg	50N Gen	Reg	50N Gen	Reg	50N Gen
Elite	2	4	2	4	4	4	2
Mutated	9	8	0	8	0	8	0
Crossover	9	8	18	8	16	8	0
New Agent	0	0	0	0	0	0	18

as EIM, elite island is formed every fiftieth generation. The difference is the rest of the agents will be replaced by new agents instead of reproducing thought mutation or crossover. This only happens in every N^{th} generation, and the rest of the settings remains the same as PIM.

Table 3.2 summarizes each model implementation and configuration. The distribution of agents is based on the total population of twenty. In the experiments, we focus on the performance of EHA* to optimize the weighted heuristic functions, the accuracy of the heuristic functions, and the search performance of EHA*. Grid-based pathfinding benchmarks are used for evaluation because these maps present different heuristic search features including branching factor, solution length, and number of states.

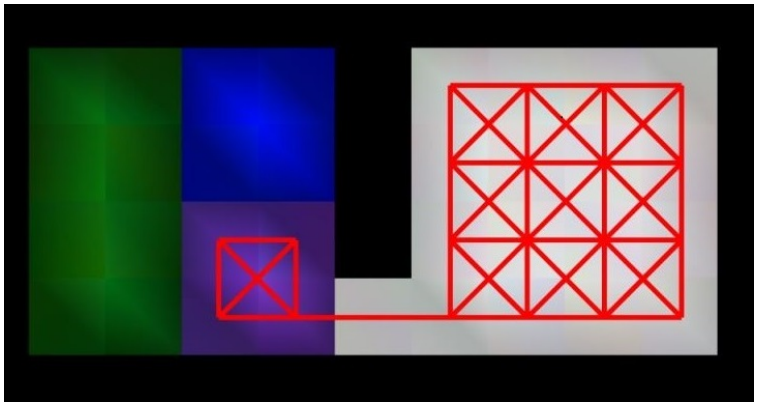
3.1.3 Benchmarks

We are using using grid-based pathfinding benchmark [31] which includes muliple maps to evaluate EHA*. The benchmark sets contain maps with high diversity including large scale maps from videos game, room maps, and maze maps. The variousness of the map types provides high difficulties for algorithm evaluations. We will introduce the map files format, types, and the properties.

The map file format used in the repository was originally developed by Yngvi Bjornsson and Markus Enzenberger at the University of Alberta [31]. As Figure 3.4a illustrated, the map format starts with the type, which is always octile, the height, the width, and the map data. All grids in a map are either blocked or unblocked. However, the textual representation of the maps is a more than just binary. We modified the setting that the cost of walking on shallow water is doubled

```
type octile
height 49
width 49
map
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTT............TTTT.TTT...TTTT.TTTT............TT
TT.............TTT........TTT..TTT.............TT
T.................................................T
T.................................................T
T.................................................T
T.................................................T
T.......................TT........................T
T......................TTT........................T
T......................TTT........................T
T.................................................T
T.................................................T
T.................................................T
```

(a)



(b)

Figure 3.4: Sample map textual (a) and graphical (b) representation. The lines from (b) show the connectivity between each grid cell.

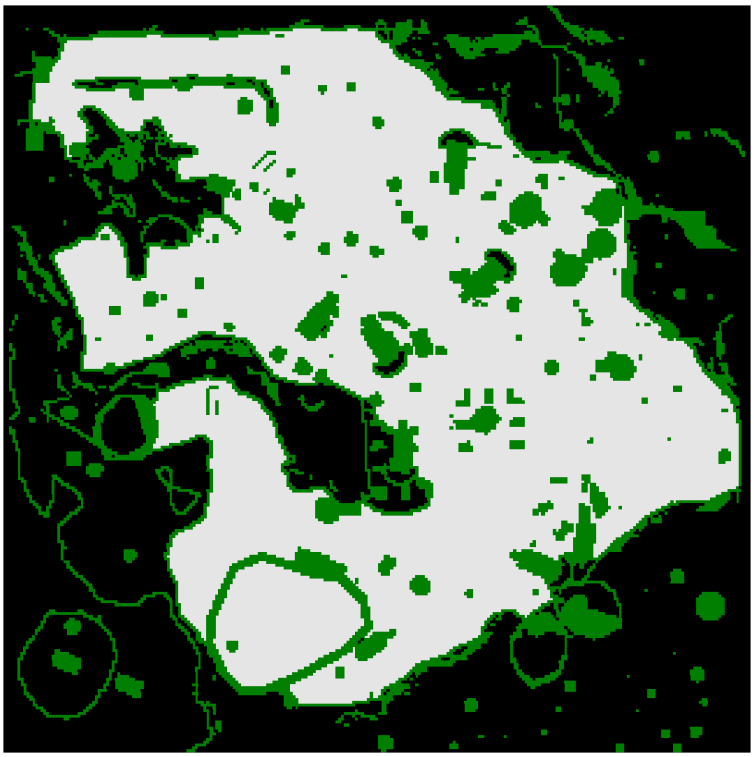


Figure 3.5: Sample 2D grid map from a video game named Dragon Age: Origins

compared to normal ground, meaning it takes a cost of 2 units to across the shallow water grid when taking the ground costs 1.

Normal ground is represented by a period ('.'), and shallow water is represented by the 'S' character. These are the only walkable types of grid. The rest of the grids are different types of blocks, including trees ('T'), water ('W') and out of bounds ('@'). Figure 3.4 illustrates a sample of the textual representation of the map in part (a) and the graphical representation in part (b). Note that the benchmarks designer assumed that diagonal moves are only possible if the related cardinal moves are both possible. That is, it is not possible to take a diagonal move between two obstacles, nor take a diagonal move to cut a corner [31]. The assumption was made under the impression that the diagonal grids do not have enough space for the subject to pass through.

The benchmark contains maps from 1) video games including Baldur's Gate II (BG), Dragon Age: Origins (DAO), and Starcraft (SC); 2) original room maps that has large local minima to increase the difficulty; 3) and mazes. Figure 3.5 illustrates a sample map from DAO. In our ex-

periment, we chose the largest maps that has the size of 512 by 512. The largest map has 137375 walkable states, while some maps only have a few hundred walkable states [31]. Therefore, the benchmark contains different levels of difficulty to evaluate the efficiency of a pathfinding algorithm.

We randomly selected 10 maps from the benchmarks, and generated 20 problem sets from each map. There are two types of problem sets, learning set and testing set for each map. The purpose of splitting into two problem sets is to train and test EHA* in two unrelated problems, which shows EHA* will not be overfitted. The problem sets are randomly selecting two points as the starting point and the goal, where the Euclidean distance of the two points must be larger than a preset value M. This condition is set to prevent these two points are being too close and decrease the difficulty. Detail of the experiment setup will be explained in the beginning of the following section.

3.2 Results and Discussion

We have selected 10 maps for problem set generation that has the size of 512 by 512, where 2 of each are from DAO, BG, SC, mazes, and original room maps. For each map, 20 problem sets are generated by getting starting and ending points that are far apart. We consider the problem set is valid when the L1 distance must be over 500 and the L2 distance must be over 400 between two points. We established these rule to ensure the solution path length for each map will not be too short that greatly reduces the difficulty to solve the problem. In the experiment, we use EHA* with different configurations for comparison.

The chromosome length of the EHA* can be changed when it comes to different problems. We have fixed the weight value to be an integer that ranges between 0 to 9 and there are 8 different heuristic to use in the gene pool. We compares the chromosome length from 2 to 10 genes, that is, the shortest heuristic function is 1 weight multiple by 1 heuristic, and the longest heuristic function is the sum of the 5 weights multiple by 5 heuristics. We can see the shortest heuristic function is the representation of the Weighted A*. With a special case of the weight equals to 1, it becomes an A* search algorithm.

The purpose of chromosome length comparison is to discover the optimal point between the performance versus training time trade off. Increasing the chromosome length provides several advantages, it provides higher flexibility, more combinations, and more inclusive. The weight can be zero so that it is equivalent to a short MWH. On the contrary, it greatly increases the training time. This is a classic cost-performance trade off issue that optimization problems often faces.

While the search performance is usually the most critical metric to evaluate the efficiency of a search algorithm, our approach mainly focuses on achieving optimal search performance. We evaluated the performance of the EHA* algorithm in the following evaluation metrics. First, the heuristic optimization time is measured in terms of the number of generations EHA* needs to assemble the most accurate heuristic function. Second, we evaluate the accuracy of the heuristic function in EHA* by observing the maximum size of the priority queue. This represents the number of nodes visited by the search agent. Third, by comparing the solution path length to the existing heuristic search algorithm, we can evaluate the optimality of EHA*. Finally, by changing the chromosome length, we can observe how much the factor affects the three metrics above. The result figures present the average results from each map and the no-solution caused by out-of-time or out-of-space does not appear on the figures.

3.2.1 Solution Path Length

In this section, we present the experimental results for the solution path length. We compare the results between each model, with different chromosome configurations. Figure 3.6 illustrates the result of the chromosome length versus the solution path length. All models are able to construct the most accurate heuristic function. Therefore, there is no difference in solution path length and the priority queue size between each model. To simplify the figure presentation, we only shows the result of the Base Model. Moreover, the solutions has no difference between each chromosome configuration. The reason of this phenomenon is that most of the heuristics in the gene pool are admissible, that guarantees EHA* to find the optimal path solution when enough amount of time and space are given. In reality, however, a less informative heuristic function might not be capable to find a solution with bounded resource. We found that in our experiment, a MWH with length

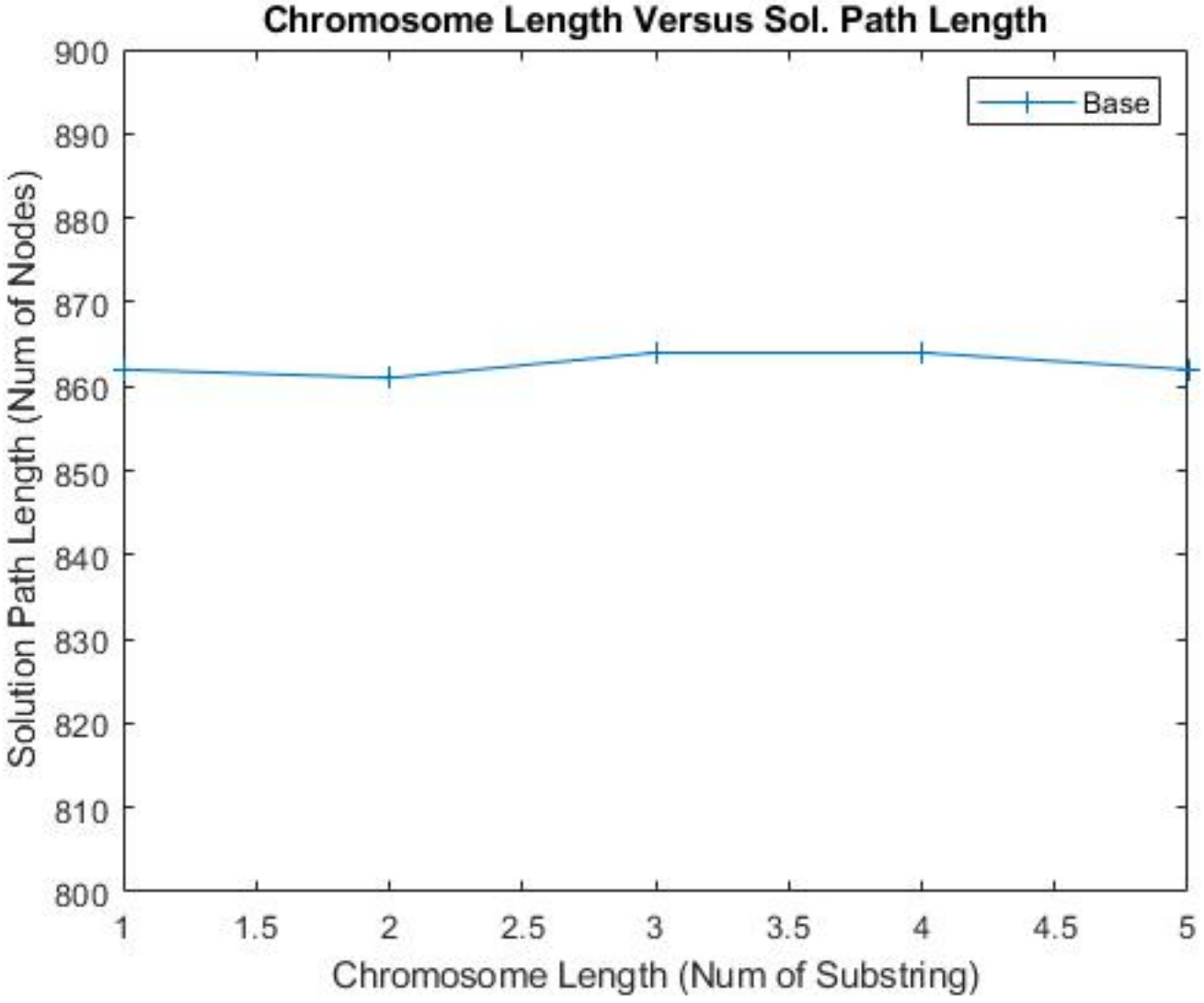


Figure 3.6: Chromosome Length vs Solution Path Length

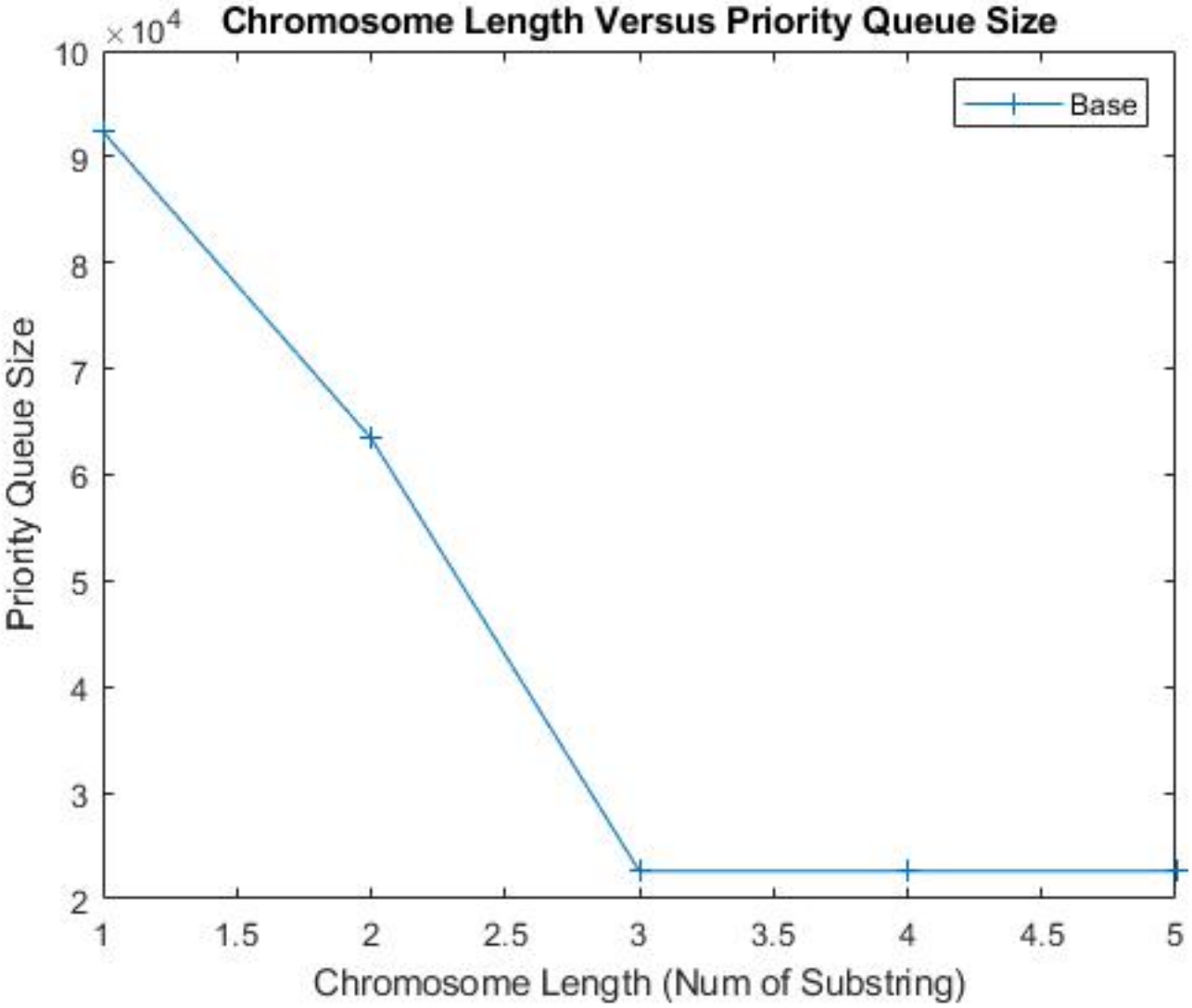


Figure 3.7: Chromosome Length vs Priority Queue Size

less than or equal to 2, does not always find the solution because the priority queue is overflowed.

3.2.2 Priority Queue Size

Compared to solution path length, the priority queue size comparison is obviously more interesting. Figure 3.7 shows that as the chromosome length increases, the priority queue size greatly decreases. The improvement stops when chromosome length reaches 3, meaning the longer chromosome does not make the MWH more informative for this benchmark. A more informative

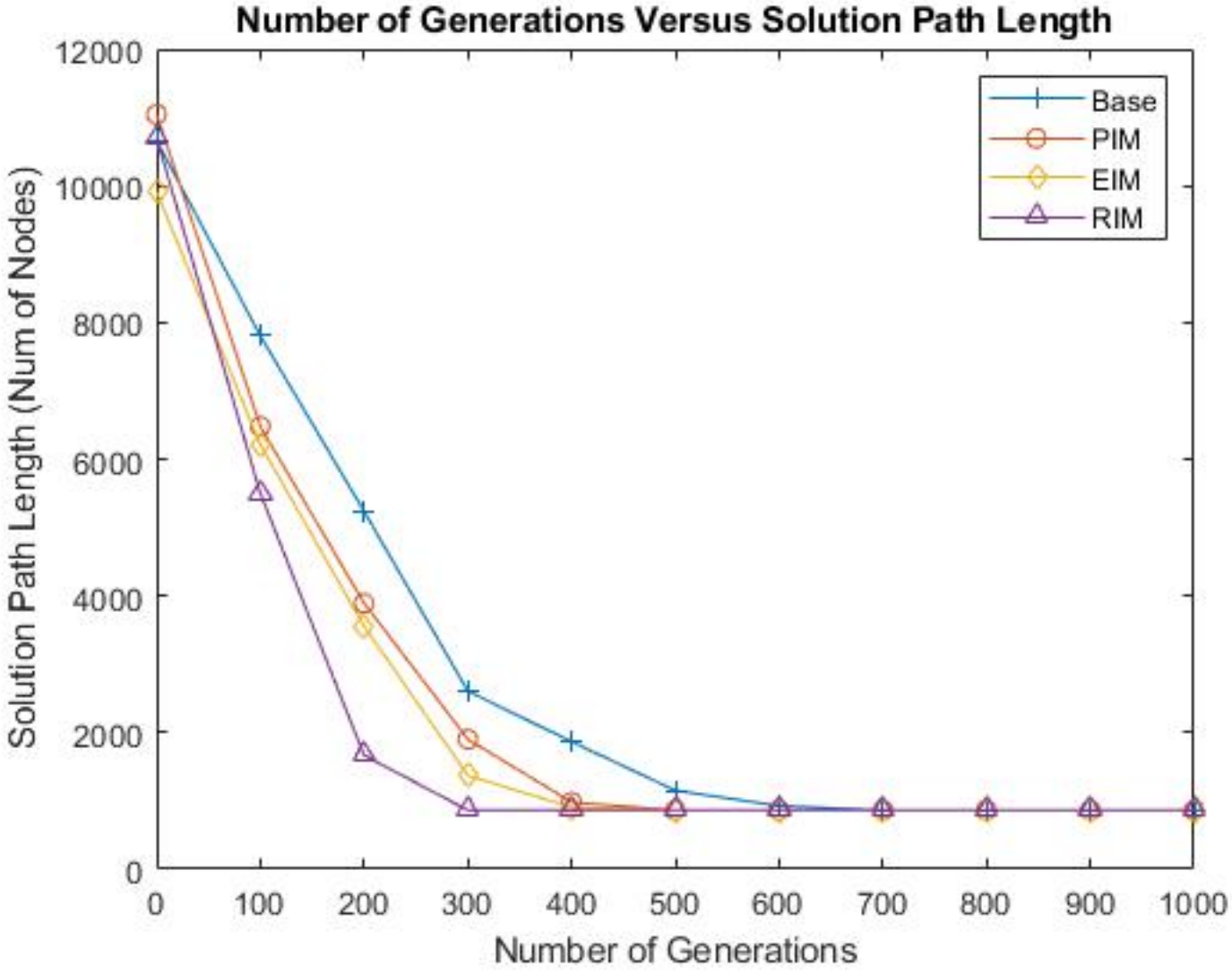


Figure 3.8: Number of Generations vs Solution Path Length

heuristic function helps EHA* to find the optimal solution in a shorter time, with less memory consumption. Same as the solution path length section, all the models show the same result because all of them are capable to generate the same MWH for the benchmark.

3.2.3 Optimization Time

Figure 3.8 shows the effects of each model act on the heuristic function optimization. The base model gradually improves the solution quality until it reaches the optimal; the PIM has a massive

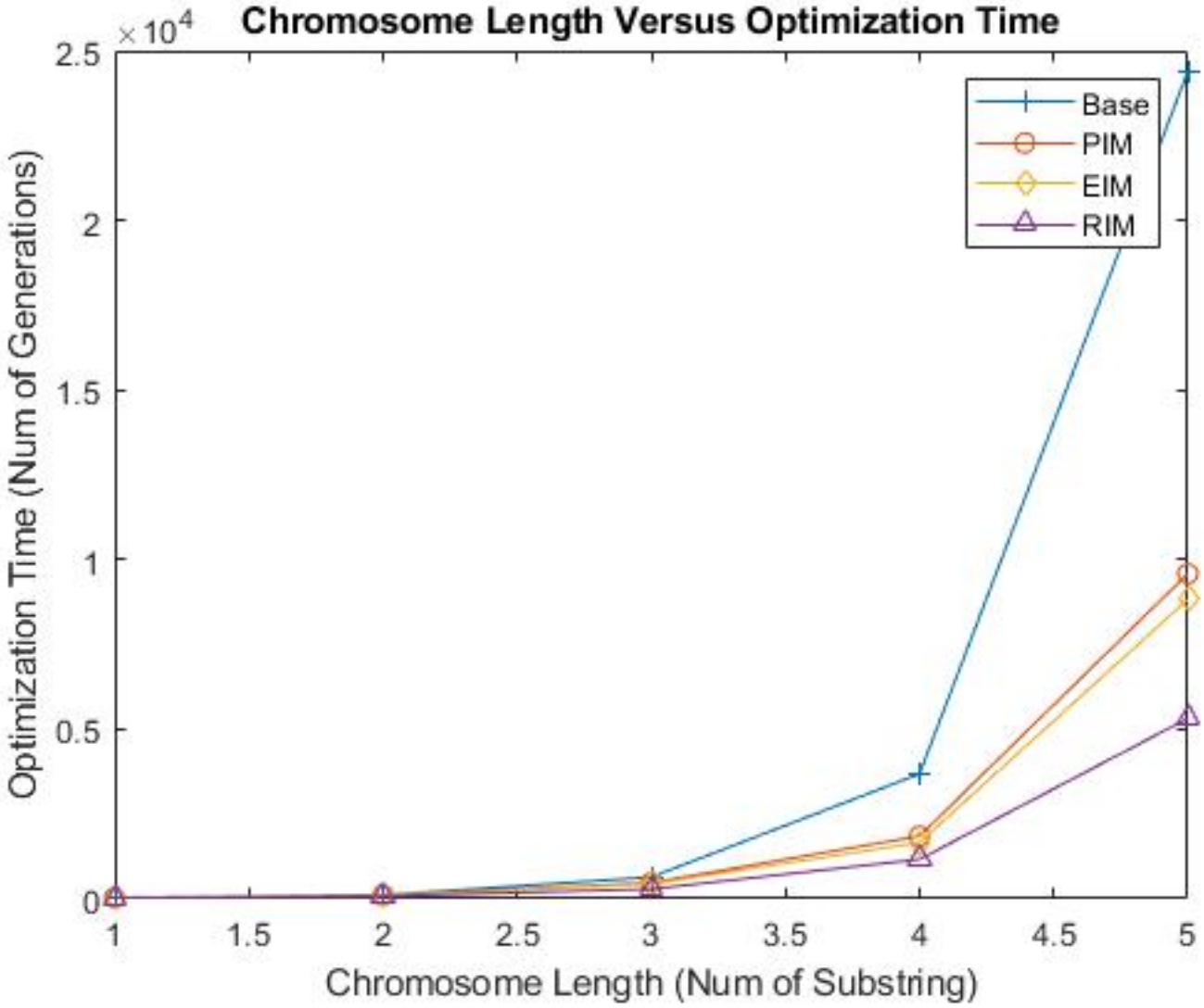


Figure 3.9: Chromosome Length vs Optimization Time

improvement in a short period of time; the EIM behave similarly to the PIM, with a slightly faster optimization; the RIM provides the fastest optimization in all models because it has the highest ability to escape local minima in an optimization problem by eliminating the old agents.

Figure 3.9 explained the relationship between the chromosome length and the optimization time. The optimization time exponentially increases as the number of parameters in the heuristic function increases. The chromosome length of EHA* can be increased by two factors: the number of parameters in the MWH and the number of choices in the gene pool and the weight value.

We have fixed the number of choices in the gene pool to 8 heuristics and the weight value to 10 integers. As the number increases, for example, more heuristics in the gene pool, or higher weight value is allowed, the number of combinations for MWH exponentially increases. We conclude that he increment of the chromosome length directly affects the optimization time for MWH.

Our experiment results demonstrated that EHA* has the following contributions to improve search performance and efficiency. First, EHA* is able to obtain the optimal heuristic function when the gene pool is inclusive enough. However, increasing the gene pool size leads to longer search time. Second, EHA* is able the design a very informative MWH that saves computational time and space. That is, EHA* is capable to design the most accurate heuristic function by combining multiple simple heuristics from the gene pool. Third, while the heuristic function has a maximum size of four heuristic combined, EHA* has the flexibility of eliminating the redundant heuristics by setting the weight to zero when a complex heuristic function is not needed for the search problem. Fourth, the methodology of EHA* can be applied to other heuristic search algorithm to minimize the complexity of algorithm design. EHA* has the capability to automatically choose both the most accuracy weight and heuristic using genetic algorithm.

3.3 Summary

In this chapter presented a self-evolving heuristic search algorithm named Evolutionary Heuristic A* which has the capability to design and optimizes a Multi-Weighted-Heuristic function. EHA* overcomes the difficulty to design high complexity heuristic functions. The experimental results show EHA* has the ability to optimize a complex heuristic function within a reasonable amount of time. EHA* is tested against previous works in different benchmarks to prove that EHA* balances the solution optimality, the memory space occupation, and the optimization time. The simplicity and effectiveness of EHA* show the potential to solve complex planning problems in many domains such as high dimensional space pathfinding [32], videogames [33], and robotics [34]. The future researches can be focused on hierarchical genetic algorithm [35], hierarchical pathfinding [36], pathfinding for dynamic environments [37], and a scalable architecture design on GPU [38] and FPGA [39]. The chromosome length increases as the heuristic function

becomes more complicated. The increment of the chromosome length exponentially increases the optimization complexity. HGA that can optimize a function partially is needed to reduce the optimization time; however, it requires a high amount of computational resources. A reconfigurable GPU or FPGA accelerator may achieve high performance computing with low energy consumption. A large-scale, efficient optimization method may efficiently design heuristic functions for complex search problems such as pathfinding for a dynamic environment. A different approach to solve a complex problem is to scale down the search size by using hierarchical pathfinding algorithm, where a search problem is divided by grids or clusters.

4. REGIONS DISCOVERY ALGORITHM [1]

Before we explore the optimization techniques of A* search algorithms, we should investigate the challenges when it comes to large map pathfinding problems. First, A* search guarantees solution optimality, it consumes a large amount of memory and CPU power to evaluate all possible options. Second, A* often faces many symmetric paths in a large map and makes the algorithm take a tremendous amount of time and memory to evaluate paths with the same cost. Third, A* depends on the heuristic function to determine the search behavior, which limits its ability to handle various environments. To overcome these challenges, researchers proposed different techniques including 1) better heuristic functions design [22], 2) preprocessing techniques [40], and 3) map abstraction techniques [41] to improve the A* search performance and reduce the computational resources consumption.

When these techniques are applied, we have to compromise among computation speed, solution optimality, and memory consumption. Depending on the objective of the pathfinding problems, different optimization techniques can be applied. One of the most common techniques being used is hierarchical pathfinding. The advantages of hierarchical pathfinding approaches are 1) vast problem complexity reduction, 2) no requirement of domain-specific knowledge, and 3) speed-up of search performance. The biggest drawback of existing hierarchical pathfinding algorithms such as Hierarchical Pathfinding A* (HPA*) [18] and Hierarchical pathfinding for Navigation Meshes A* (HNA*) [20] [21] is that they do not provide optimal solutions. As the environment complexity increases, suboptimal pathfinding algorithms might not function well.

Many existing graph decomposition approaches would divide graphs into many equally-sized areas. While evenly dividing a graph can be done quickly and efficiently, it might not be the best partitioning technique because it can cause pathfinding algorithms to return non-optimal solutions. In this chapter we present a novel approach named Regions Discovery Algorithm (RDA) in this

[1] Reprinted with permission from "Regions Discovery Algorithm for Pathfinding in Grid Based Maps." by Ying Fung Yiu and Rabi Mahapatra, 2020. 2020 Second International Conference on Transdisciplinary AI (TransAI), pp. 84-87, DOI: 10.1109/TransAI49837.2020.00018, Copyright ©2020 IEEE;

paper to reduce search space while maintaining the solution optimality. Unlike most of the hierarchical approaches, we do not create abstract maps to change the representation of the search space. There is no new vertex added to the graph nor precomputed path between regions. Our approach simply identifies and removes regions that are not going to be included in any possible solution paths. Therefore, RDA can reduce search complexity, provide flexibility when it comes to dynamic search environments, and guarantee optimal solutions under any circumstances.

We apply RDA to the grid-based pathfinding benchmark [31] which includes multiple two dimensions grid-based maps. This benchmark set is chosen because of its simplicity and diversity. It contains many maps with different characteristics including large scale maps from video games, room maps, and maze maps, it is also easy to use and simple to modify. The variousness of the map types provides a more objective result for algorithm evaluations. To evaluate RDA by the performance gain, memory consumption, and the solution optimality, we compare it with other popular pathfinding techniques including A*, Weighted A*, and HPA*.

The main contributions of this paper includes a new map abstraction approach that: 1) is domain independent and able to automatically partition grid-based graphs into small regions based on local features, 2) can be combined with existing A* optimization techniques to further improve the search performance, 3) guarantees optimal solutions.

4.1 Regions Discovery Algorithm Overview

The goal of RDA is to cluster search spaces with little overhead. By adopting the advantage of PDBs and hierarchical approaches, our approach improves search performance and guarantees optimal solutions. The algorithm starts with creating an abstract map by dividing the map into many local clusters and adding extra vertices to connect each cluster.

Figure 4.1 illustrates an example of A* search with Manhattan distance as the heuristic function. The explored area is marked in pink, and the solution path is marked in green. When finding the shortest path between two given nodes, A* search would explore almost the entire map before it finds an optimal solution. This is because the heuristic function can only estimate the distance between the given tile to the goal without any knowledge of the obstacles. The algorithm spends

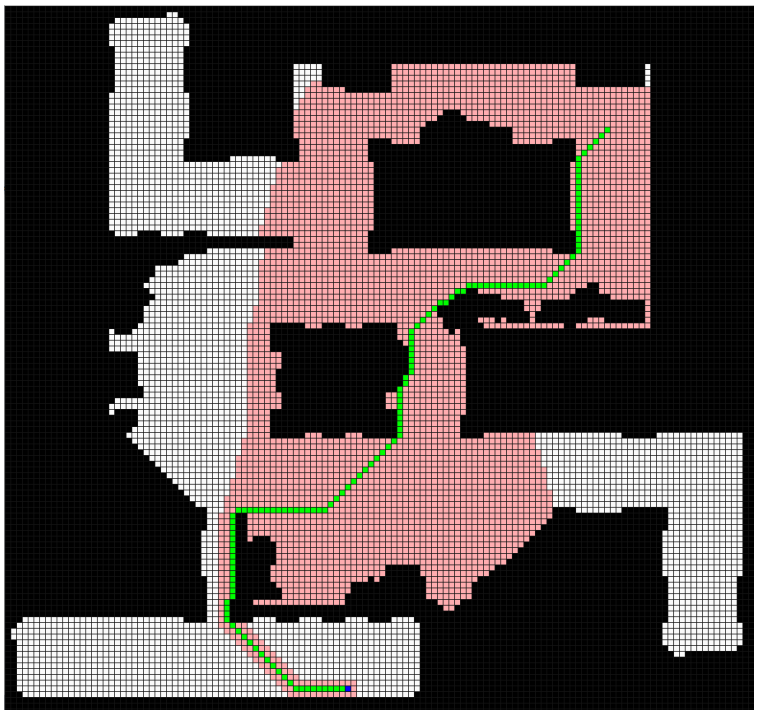


Figure 4.1: Traditional A* search with Manhattan distance as the heuristic function.

lots of time exploring the area that clearly cannot lead to the goal. The objective of this paper is to identify and exclude these areas before the search, so that it would not waste so much effort on searching through the unnecessary areas. Moreover, the identification and exclusion process of the algorithm should not be too aggressive to remove areas that are likely to be parts of an optimal solution.

For instance, HPA* [18] [19] introduces a preprocessing technique to cluster the map into subgrids using no domain specific knowledge. While the clustering method is fast and simple, cutting the map into equally-sized subgrids might not exploit the full advantage of reducing search space. Figure 4.2 shows the same map that is being clustered into 4 by 4 subgrids. In these subgrids, some of them are being divided by the obstacles into two zones that cannot be connected without going through to another subgrid. This is a major drawback to map abstraction and search space reduction for obvious reasons.

We introduce the Region Discovery Algorithm (RDA) that partitions maps into smaller regions

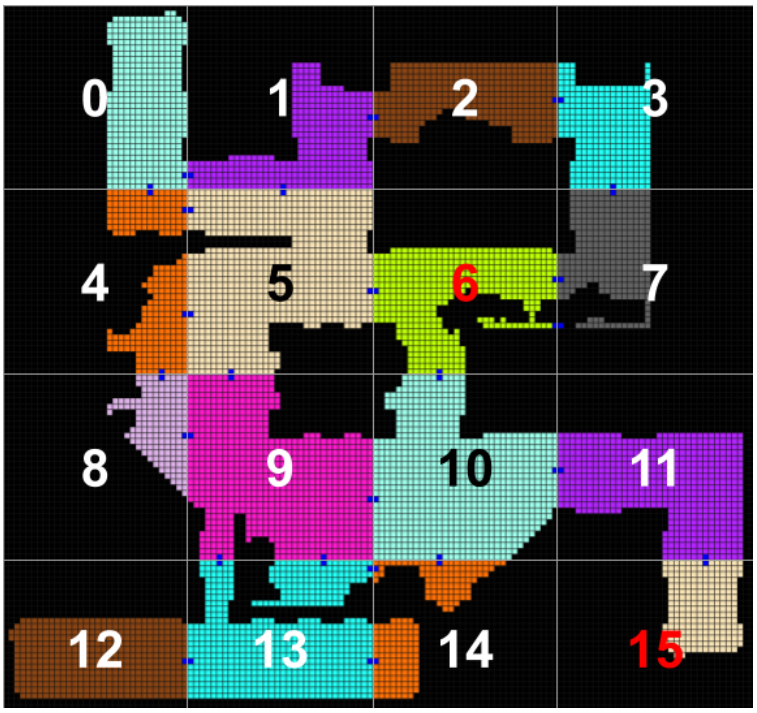


Figure 4.2: HPA* clusters the sample map into a 4x4 subgrids of equal size.

based on their local features so that the search agent can avoid areas that cannot possibly be part of a solution path. This is accomplished in the preprocessing phase by automatically clustering the map into many small clusters and then removing the useless areas. This preprocessing phase is an offline computation, and it only needs to be done once for each map.

4.1.1 Regions Discovery Algorithm

RDA automatically creates borders and divides a map into many clusters based on certain conditions. The algorithm does not require any specific knowledge about the map other than the map itself. It scans through the map and identifies what zones the tiles belong to. Figure 4.3 is the flowchart of RDA and it shows the basic principles of how it defines and clusters a zone. Consider the map as 2D arrays with N rows and M columns, the top left has the coordinate of (0, 0) and the bottom right has the coordinate (N, M). RDA starts from the top left and scans from left to right, top to bottom to find the first available node as the starting node for zone exploration. As Figure 4.3 illustrates, it keeps moving to the right until it hits the wall or enters a different zone. The

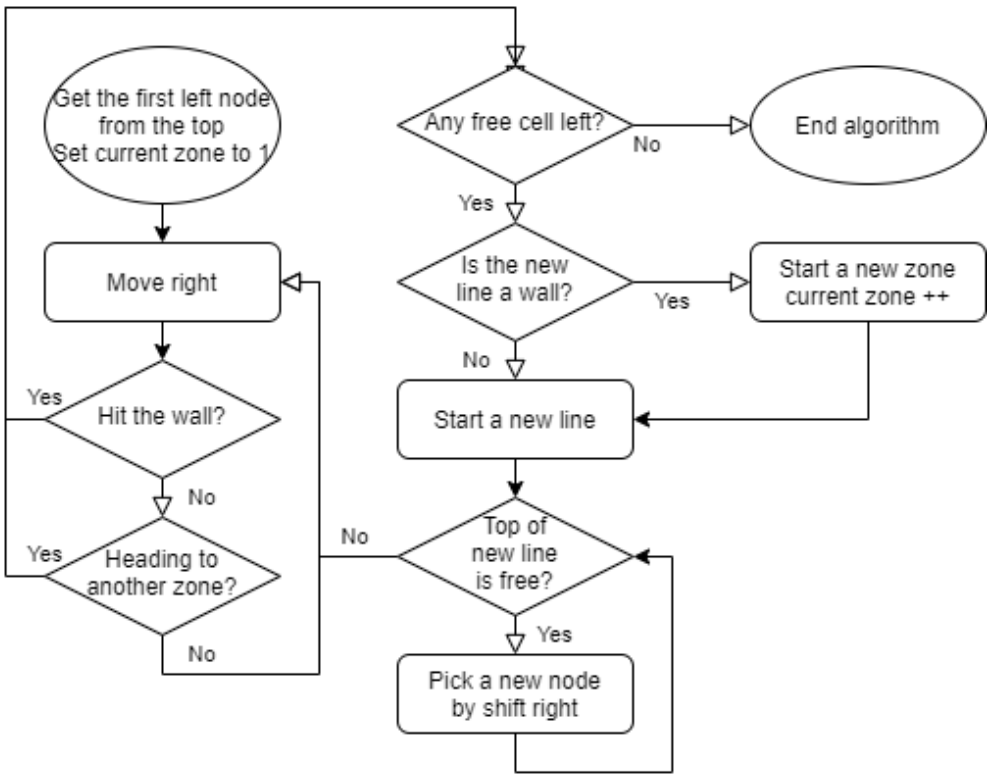


Figure 4.3: Flowchart of Regions Discovery Algorithm RDA. RDA automatically clusters the map into small areas based on the local features.

explorer monitors its surroundings and if it sees a free tile on its top right corner, it stops exploring and sets the right border m', meaning all new lines of the current zone cannot go beyond the m' column. To start with a new line, first we need to check if the top of the left-most tile is free. If so, it will start a different zone, thus it needs to move right until its upper tile is either a wall or the current zone. Once it has completed exploring the row, it needs a final check that at some point of the row, it must be attached from the current zone, otherwise it is not going to be part of the zone and RDA starts a new zone by repeating the procedure. This process repeats until every tile is assigned to a zone.

4.1.2 Lookup Tables

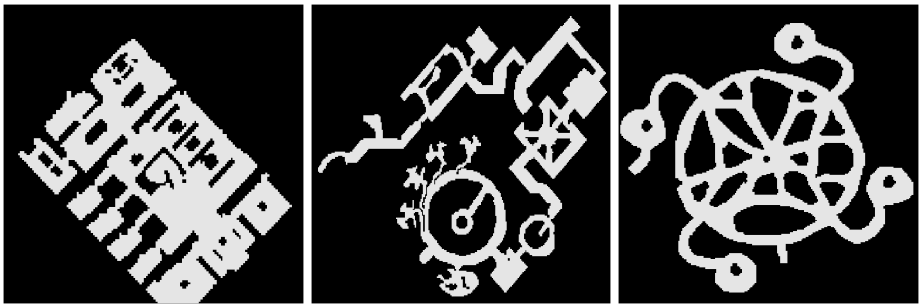
RDA creates two lookup tables simultaneously while discovering the zones. Both lookup tables are used for search space reduction. The first lookup table records the number of gates a zone has.

The algorithm determines the type of these zones depending on the amount of gates. A zone with one gate is a dead end zone. A two-gate zone is a hallway zone. Any zone that has over two gates is an interconnect zone. The second lookup table is a NxN binary table to show the connections between zones. It is a simplified version of an abstracted map, but without any extra node or vertex creation. If there is a connection between Zone A and Zone B, both d(A, B) and d(B, A) are set to 1, otherwise, both are set to 0. The first lookup table is used to reduce the search space by removing all the dead end zones that the start or the goal does not belong to. Then, an abstract path search is done on the second lookup table. The lookup table simply shows which zones are connected together. Therefore, the abstract path search only removes zones that are absolutely impossible to visit.

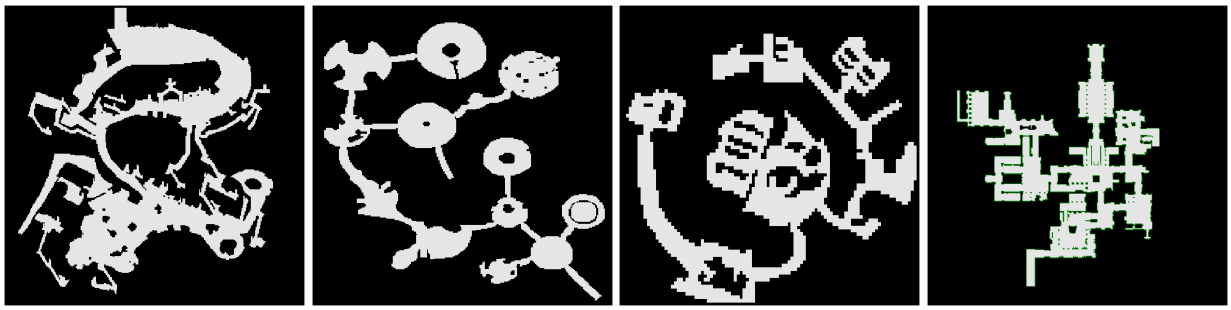
4.2 Experiment Design

We evaluate the performance of RDA on benchmarks for grid-based pathfinding [31] taken from Nathan Sturtevant's Moving AI Lab. This is a widely used benchmark in many of the literatures. The blackened tiles are the walls or obstacles that are untraversable and the white tiles are traversable. The benchmark does not only contain many types of maps including mazes, real cities, and video game maps, but also the maps that are large and complex enough to examine the bottlenecks of pathfinding algorithms. The largest maps in the benchmark can have up to hundred thousands of walkable terrains. We have chosen a set of 10 most complicated maps from the benchmark shown in Figure 4.4, with the size of all maps are scaled to 512 by 512. In each map, we run 100 searches with a similar optimal path length that are provided from the benchmark problem set. The searches are not randomly generated for two reasons. First of all, the problem set provides the optimal solution cost as a comparison and second, the lengths of all optimal paths are similar so that it is easier for comparison between different algorithms.

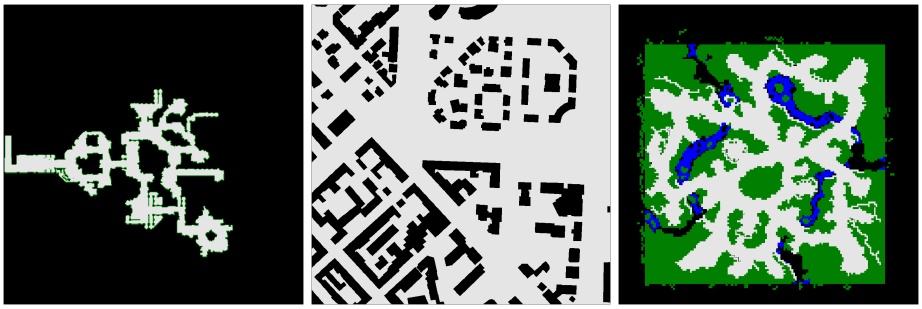
In the experiment, we assume that diagonal moves are only possible if the related cardinal moves are both possible. All grid-based maps use octile neighborhood relation, meaning the adjacency relationship in 4 straight and 4 diagonal directions. At any given time during the search, an agent that is allowed to move to one of its up to eight traversable neighbors. That is, it is not



(a) AR0012SR.map (b) AR0202SR.map (c) AR0205SR.map



(d) AR0300SR.map (e) AR0411SR.map (f) AR0705SR.map (g) hightown_a3.map



(h) lowtown_a2.map (i) Milan_1_256.map (j) tranquilpaths.map

Figure 4.4: Applied RDA to partition maps used in the experiment.

Table 4.1: Search space reduction after RDA is applied to the benchmark.

Map	AR0012SR	AR0202SR	AR0205SR	AR0300SR	AR0411SR	AR0705SR	hightown	lowtown	Milan	tranquilpaths
Original	78614	50514	66753	68964	58581	68690	40127	31456	47798	71032
w/ RDA	53174	30778	53274	56700	31035	48129	25548	23092	101423	48736
Percentage	67.64%	60.93%	79.81%	82.22%	52.98%	70.07%	63.67%	73.41%	75.69%	68.61%

possible to take a diagonal move between two obstacles, and it is not possible to take a diagonal move to cut a corner. Such an assumption is made because in a real game, all creatures occupy some volume of space and are not able to pass through blocked corners. All grid-based maps use octile neighborhood relation, meaning the adjacency relationship in 4 straight and 4 diagonal directions. At any given time during the search, an agent that is allowed to move to one of its up to eight traversable neighbors. The cost of vertical and horizontal movements is 1 and any diagonal movement has the cost of $\sqrt{2}$.

We design the experiment to compare RDA with the several popular pathfinding algorithms including A*, Weighted A*, HPA*. After RDA is applied to a map, we then use A* to search for the shortest path. To compare the performance between these algorithms, preprocessing time and memory consumption, execution, solution optimality, and the priority queue size are evaluated. All algorithms are using Manhattan distance as the heuristic function to avoid any bias. To the fairness of each algorithm, all experiments are single threaded to eliminate the advantage of parallel computing. All experiments were running on the same machine with Intel i7-4790k at 4.0 GHz, with 32.0 GB of RAM using python 3.6.9 on Windows Subsystem for Ubuntu.

4.3 Results and Analysis

RDA removes areas that are unnecessary to search. Depends on where the start and goal node are, the algorithm use both look up tables to determine what regions are needed to remove. For instance, all dead ends that start and goal nodes do not belong to are removed. This leads to a much smaller search space for the agent to handle, hence improves the search performance. Table 4.1 shows the average reduction of each map by RDA in our experiment. The results show that RDA works best with large dead end areas such as map B and E that can be entirely removed.

RDA scans through maps in a top to bottom, left to right manner. Such approach creates borders that are either vertical or horizontal. Therefore, RDA partitions boxy maps better than other non-rectangular maps. Moreover, RDA works particularly well when there are many dead ends in the map. This explains the performance increases compared to traditional A* search. The search space not only is significantly smaller compared to the original graph, the remaining search space is also simpler to search. With most of the local minima are removed by RDA, A* with even the simplest heuristic function can rapidly return optimal solutions. RDA can reduce the map size up to 47% in our experiment. Moreover, RDA does not create any cluster that is split by walls or obstacles compared to the clustering method used in HPA*. Therefore, it improves the efficiency and quality of abstract level maps. Table 4.2 shows our pathfinding experiment results of multiple pathfinding algorithms are being compared. In conclusion, our experiment results show that 1) RDA trades off little preprocessing time and memory for performance improvement, 2) RDA guarantees solution optimality, and 3) RDA is a simple but efficient algorithm that reduces search complexity for pathfinding problems.

Table 4.2 and 4.3 shows our pathfinding experiment results of multiple pathfinding algorithms are being compared. A* is our baseline algorithm. It guarantees solution optimality, but the execution time and memory consumption are the highest among all. Weighted A* is as simple as A*, but the execution time is roughly 70% lower than A*. However, the result shows that Weighted A* returns the worst solutions among all algorithms. HPA* has the advantage of fast computation thanks to paths precomputation. Because of the majority of pathfinding process has been completed in the preprocessing phase, the actual online search is reduced to minimal. In HPA*, only two of the clusters are being search online: 1) local search from start node to start zone border node, 2) abstract map search from start zone to local zone, and 3) local search from goal border node to goal node. However, the precomputation process is significantly long because many intra-edges for the abstract map are needed to compute. Since all the intra-edges are computed using A*, it leads to better suboptimal solutions compared to any version of weighted A*. Compared to HPA*, RDA has the advantage of shorter preprocessing time and memory consumption, simply

Table 4.2: Pathfinding results comparisons of each map

Map Name	Algorithm	Preprocessing Time (ms)	Execution Time (ms)	Priority Queue Size	Solution Length
AR0012SR	RDA	1888.23	1020.26	36754.68	546.944141
	A*	N/A	1560.720572	55378.7	546.944141
	WA*$_{w=2}$	N/A	368.5408	23815.6	589.8483
	WA*$_{w=10}$	N/A	123.9333932	15200.6	594.7211322
	HPA*$_{4x4}$	4033.344	226.1625	2636.471	557.1864
	HPA*$_{8x8}$	1618.797	48.801	260.8716	566.352781
AR0202SR	RDA	1364.93	293.8545	18872.27	543.5058726
	A*	N/A	540.6978728	35921.7	543.5058726
	WA*$_{w=2}$	N/A	358.5419397	27203	574.6321928
	WA*$_{w=10}$	N/A	407.4872408	23196.9	607.668946
	HPA*$_{4x4}$	2651.846	112.977	1497.65	558.1186
	HPA*$_{8x8}$	1420.28	41.543	188.114	575.337
AR0205SR	RDA	1405.35	813.8657	34160.31	545.863737
	A*	N/A	1074.1282	45622.3	545.863737
	WA*$_{w=2}$	N/A	358.1313	24154.4	565.4832
	WA*$_{w=10}$	N/A	125.0167544	15303.6	565.8932968
	HPA*$_{4x4}$	2469.307797	135.5081	1841.44057	554.238
	HPA*$_{8x8}$	1419.91	38.5437	172.654	561.3284
AR0300SR	RDA	1442.051	747.5585	33834.63	545.2014137
	A*	N/A	919.1134644	44192	545.2014137
	WA*$_{w=2}$	N/A	65.21228	11807.5	565.2996
	WA*$_{w=10}$	N/A	33.21919753	7560.8	579.4961538
	HPA*$_{4x4}$	3098.986	172.532	2562.543	550.124
	HPA*$_{8x8}$	1518.281	46.288	231.484	552.47106
AR0411SR	RDA	1372.69	263.9398	17923.79	544.7160098
	A*	N/A	685.2459741	36483.4	544.7160098
	WA*$_{w=2}$	N/A	535.2192	31572.8	570.3889
	WA*$_{w=10}$	N/A	623.8719573	33978	573.5202954
	HPA*$_{4x4}$	2124.628	131.518	1668.2343	558.5106
	HPA*$_{8x8}$	1315.535	36.689	91.448	561.3287

Table 4.3: Pathfinding results comparisons of each map

Map Name	Algorithm	Preprocessing Time (ms)	Execution Time (ms)	Priority Queue Size	Solution Length
AR0705SR	RDA	1429.677	602.5419	30318.49	544.0366158
	A*	N/A	883.6838537	45390.6	544.0366158
	$WA^*_{w=2}$	N/A	789.386	36104.4	569.574
	$WA^*_{w=10}$	N/A	529.3010241	29550.3	576.7480082
	HPA^*_{4x4}	2254.354	135.1494	1801.69428	556.609
	HPA^*_{8x8}	1413.316	37.4218	169.255	563.25211
ht_0_hightown_a3	RDA	1282.87	151.9275	17372.39	538.9687876
	A*	N/A	238.1366463	27077.4	538.9687876
	$WA^*_{w=2}$	N/A	320.8206	25388.2	567.8909
	$WA^*_{w=10}$	N/A	139.1065772	18966.9	570.3453603
	HPA^*_{4x4}	6919.537	177.2352	2591.493	552.451
	HPA^*_{8x8}	1559.72	46.9431	256.31208	560.23897
lt_0_lowtown_a2_n_c	RDA	1323.38	158.683	18141.24	485.7313201
	A*	N/A	218.0864544	24847.8	485.7313201
	$WA^*_{w=2}$	N/A	69.08735	11194.8	522.9277
	$WA^*_{w=10}$	N/A	20.52341847	6904.6	534.5561251
	HPA^*_{4x4}	2457.961	210.433	2734.592845	501.51
	HPA^*_{8x8}	1538.765	46.458	262.6916	506.347
Milan_1_256	RDA	569.92	28.1783	6247.383	348.0954903
	A*	N/A	42.06531859	9080.7	348.0954903
	$WA^*_{w=2}$	N/A	10.50281	4586.6	377.1334
	$WA^*_{w=10}$	N/A	5.365340042	3827.8	385.5358149
	HPA^*_{4x4}	2313.061	108.5433	1454.348	352.785
	HPA^*_{8x8}	1658.991	26.3910	85.193	356.1936
tranquilpaths	RDA	1683.31	201.683	13576.9	538.3277846
	A*	N/A	298.1428395	24105.7	538.3277846
	$WA^*_{w=2}$	N/A	57.20954	10072.3	630.4549
	$WA^*_{w=10}$	N/A	11.85008471	5439.9	610.7630586
	HPA^*_{4x4}	3981.162	261.1495	3461.14	549.529
	HPA^*_{8x8}	1328.492	51.3123	297.9323	552.2353

because there is no path precomputation used in our method. Moreover, RDA does not sacrifice solution optimality for speed up, unlike HPA*. Among all the A* variants, using RDA with A* search provides the best balance of preprocessing time, execution time, memory consumption, and solution optimality.

Weighted A* is known for fast searches, although it does not guarantee optimal solutions. Our results show that sometimes weighted A* search gets trapped in local minima a lot in some of the problems, causing it has poor performance. On the other hand, RDA removed many regions that are unnecessary to visit, resulting in a much smaller number of priority queue size compared to A* and weighted A* search. RDA provides a lot more stable result in performance boosting. Therefore we have faster search and better memory consumption with RDA compared to weighted A*. In conclusion, our experiment results show that 1) RDA trades off little preprocessing time and memory for performance improvement, 2) RDA guarantees solution optimality, and 3) RDA is a simple but efficient algorithm that reduces search complexity for pathfinding problems.

4.4 Summary

In this chapter we have presented a novel clustering algorithm named Regions Discovery Algorithm (RDA) for grid-based maps to speed up pathfinding and reduce search effort. RDA clusters and identifies regions based on their local features, which leads to better graph reduction results. RDA uses lookup tables to reduce search space instead of precomputed paths to 1) minimize time and memory consumption in proprocessing phase, and 2) guarantee solution optimality. Moreover, our approach is environment independent, simple, and effective. Our work can be combined with other pathfinding algorithms for further performance boosting. We have tested our algorithm using a widely used benchmark and compared with several popular pathfinding algorithms. Our results show that RDA has little overhead in preprocessing, while having significant performance gain in the pathfinding process to search for optimal solutions. We have only demonstrated RDA with A* search on in the experiment, but our approach could also be used with other pathfinding algorithms.

5. HIERARCHICAL EVOLUTIONARY HEURISTIC A* SEARCH [1]

Due to the difficulty of parameters hand tuning process, a lot of researchers attempted to apply Machine Learning (ML) methods, including Deep Neural Network (DNN) [26] [27] and Reinforcement Learning (RL) [28] to form heuristic functions based on the given environment. The advantage of using DNN that is trained by RL as a heuristic function is that the heuristic function learns about the environment and optimizes itself over iterations, instead of having a team of engineers to calibrate the parameters via trial and error. However, both DNN and RL are extremely computation and memory hungry that most of the pathfinding applications cannot afford to provide.

NeuroEvolution (NE) [42] shows that evolutionary algorithms can outperform RL in both execution speed and solution quality. In this chapter, we present a novel pathfinding algorithm named Hierarchical Evolutionary Heuristic A* (HEHA*) that combines the advantages of several pathfinding optimization techniques including 1) hierarchical pathfinding, 2) pattern databases (PDBs), and 3) Genetic Algorithm (GA). HEHA* partitions the map into many smaller regions based on local features, and each region has its own heuristic that is evolved to be the fittest. HEHA* creates an abstracted map only to remove unnecessary search space, meaning all searches are done at the local level to ensure solution optimality. No precomputed path is done during the preprocessing phase. We make the following contributions:

1. We propose the Hierarchical Evolutionary Heuristic A* search (HEHA*) to automatically design the fittest heuristic functions for regions in the map.

2. We evaluated HEHA* on different types of map and proved that HEHA* yields a better search performance compared to other pathfinding algorithms.

3. HEHA* overcomes the assumption that hierarchical pathfinding cannot guarantee solution

[1] Reprinted with permission from "Hierarchical Evolutionary Heuristic A* Search." by Ying Fung Yiu and Rabi Mahapatra, 2020. 2020 IEEE International Conference on Humanized Computing and Communication with Artificial Intelligence (HCCAI), pp. 33-40, DOI: 10.1109/HCCAI49649.2020.00011, Copyright ©2020 IEEE;

optimality.

We apply HEHA* to the grid-based pathfinding benchmark [31] which includes multiple two dimensional grid-based maps. This benchmark set contains many large scale maps with different characteristics, and it is simple to use. The variousness of the map types not only yields more objective results for algorithms evaluation, but it also proves that HEHA* can always provide the most accurate heuristic function for different scenarios. To evaluate HEHA* by the performance gain, memory consumption, and the solution optimality, we compare it with other popular pathfinding techniques including A*, Weighted A*, and HPA*.

5.0.1 HEHA* Overview

As the complexity of search problems increases, the demand of better heuristic functions increases as well. However, having a single heuristic function to capture all the complexity could be difficult to achieve. Dividing search problems into smaller and simpler subproblems can reduce the search effort of pathfinding algorithms, hence reducing the heuristic function complexity. Moreover, partitioning search graphs opens up the opportunity to apply different heuristic functions based on local features, so that the pathfinding algorithm explores the minimal number of states when searching for an optimal path.

HEHA* can be separated into a preprocessing phase and an online pathfinding phase. In the preprocessing phase, HEHA* clusters the given search graph based on local features and creates an abstract graph. The abstract graph is created by a graph partitioning algorithm inspired by Regions Discovery Algorithm (RDA) [3]. The abstract graph created by RDA transformed the search graph into a navigation mesh (NavMesh) graph. Unlike the usual hierarchical pathfinding approach, HEHA* does not precompute and store any path to speed up the search. Instead, HEHA* stores the abstract graph as a lookup table with the estimated values between borders of regions. The advantages of this approach are twofold: it leads to higher accuracy of the heuristic values and guarantees optimality. The abstract graph can be treated as an assistant of the heuristic function. The purpose of the abstract graph is to reduce search spaces that are not part of the solution. Unlike

traditional hierarchical approaches, HEHA* doesn't eliminate any possible solutions.

After the abstract graph is created, HEHA* evolves heuristic functions for different regions. HEHA* categorizes regions based on the number of borders they have. Regions with similar characteristics should use the same heuristic function. This approach reduces the evolving time and memory consumption. After heuristic functions are evolved, the online pathfinding can begin.

5.0.2 Map Abstraction

There are many ways to create an abstract map. The most straightforward approach is to evenly divide a map into many smaller maps and connect them by creating new edges. Hierarchical Pathfinding A* (HPA*) [18] is one of the most efficient examples of map abstraction. HPA* first partitions a map into an NxN grid map, picks border nodes from borders in each subgrid, and then calculates the costs of intra-edges between these border nodes. Figure 5.1 shows an example of how HPA* creating an abstract graph from a map. While this is the fastest and simplest approach to divide a map, it might not be the best performance booster to pathfinding problems.

Our previous work RDA suggests that a map should be partitioned based on its local features. RDA breaks down a map into many small areas with different shapes and characteristics. Some are rooms with only one exit, some are hallways with two exits, while others are lobby-like areas with many exits. An abstract map created by RDA can improve the pathfinding speed by efficiently reducing search space. In this paper, we propose to modify RDA so that it provides a higher map reduction rate, faster pathfinding speed, and more accurate heuristic estimations.

In the original RDA, there are two lookup tables to store the map partition information. The first lookup table records the number of gates a zone has. The second lookup table is a NxN binary table to show the connections between zones. The original RDA eliminates areas that cannot be in the solution. For instance, any room or dead end that does not contain either the start or the goal can be eliminated. In the improved RDA, we keep the lookup table 1 and modify the table 2 to improve the search space reduction performance. HEHA* creates an abstract by choosing the midpoints of borders as abstract nodes and estimated distances between abstract nodes as abstract vertices.

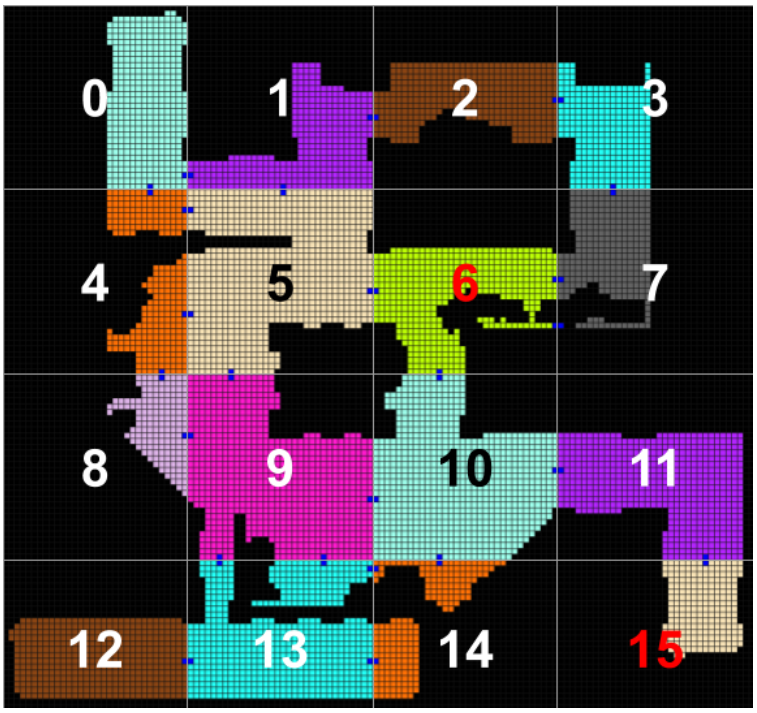


Figure 5.1: HPA* clusters the sample map into a 4x4 subgrids of equal size. The entrance tiles are marked as blue and are identified as inter-clusters connections.

The advantage of using estimated values is that it creates abstract maps faster than HPA*. No pathfinding is done in the preprocessing phase. The abstract map is used as a more accurate heuristic function to estimate the distance between start and goal. This approach reduces the time to search in local minima. Instead, all local minima are eliminated during abstract search, leaving a relatively smaller search area for A* to search. This approach highly reduces the computation time, complexity, and memory consumption. The abstract search relieves the responsibility heuristic functions have in pathfinding. After a map is partitioned, HEHA* starts to evolve heuristic functions for each type of region.

5.0.3 Heuristic Functions Evolution

HEHA* applies GA to evolve heuristic functions based on local features of search spaces to solve pathfinding problems. The algorithm starts with a pool of agents, each of them try to solve the problem on their own. The pool of agents is called a population. Each agent is represented

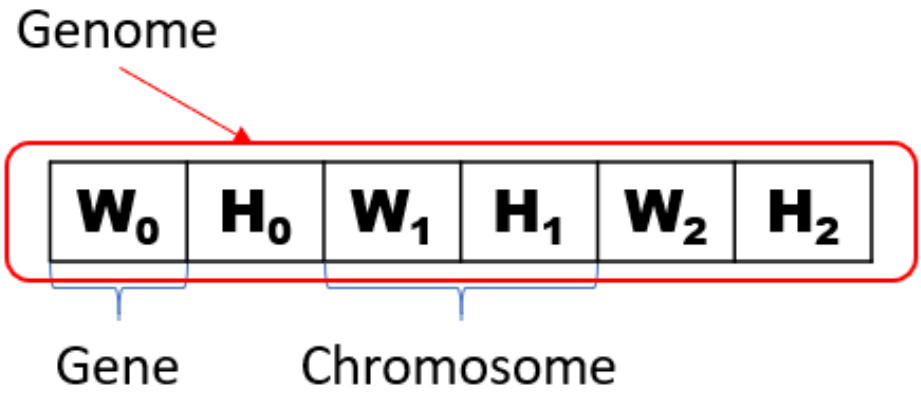


Figure 5.2: Structure of a genome used in HEHA*.

in a numerical form that we call it a genome. A genome is a combination of genes, which is the basic building block of an agent. In HEHA*, a gene can either be a weight value or a heuristic index. Figure 5.2 shows the structure of a genome. The genome has at most three weights and three heuristic indices.

The flow of heuristic functions evolution starts from population initialization. Each agent is assigned with a weight value of 1 and a random heuristic index. After initialization, agents solve the pathfinding problem individually with the given heuristic functions. The formula of the heuristic function is a sum of weighted-heuristic-functions: $\sum_{i=0}^{2} W_{\mathrm{n}} * H_{\mathrm{n}}$. Note that the sum of weights must be equal to one, otherwise the heuristic function might overestimate the distance between current nodes to the goal. Overestimation might lead to non-optimal solutions. The heuristic index represents the corresponding heuristic function from a preset table we called a gene pool. A gene pool contains commonly used heuristic functions that may or may not be admissible shown in Table 5.1. By combining the search power of different heuristic functions, HEHA* increases the search ability and accuracy of the heuristic function.

After agents have solved the given problem, HEHA* evaluates performance of agents via solution optimality and memory consumption, and then ranks them accordingly. HEHA* evaluates the solution for every region to ensure every heuristic function guarantees completeness and optimality. The purpose of evaluation is to create the next generation of population. HEHA* evolves these

Index	Heuristics
0	Difference of X Coordinate
1	Difference of Y Coordinate
2	Manhattan Distance
3	Euclidean Distance
4	Diagonal Distance
5	Weighted $A^*_{w=2}$
6	Weighted $A^*_{w=10}$
7	lean to left wall
8	lean to right wall
9	in favor to move up/downward first

Table 5.1: Gene Pool

agents via selection, crossover, and mutation. In the selection phase, the best agents are chosen to create the next generation. The chosen agents are either unchanged to the new generation, or crossover between two agents to create new offspring. On top of crossover, the new offspring has a small probability to mutate, meaning one of the genes is changed randomly. Crossover exchanges a pair of genes we called chromosome, which is one heuristic index with its corresponding weight, from each parent randomly. After exchanging chromosomes, the weight will be recalibrated to ensure the sum of weights equals to one. After the new generation is created, repeat the fitness evaluation process until the preset criteria are met. In our case, the only criteria is the maximum number of iterations.

The principle of GA is having the population to compete with each other and select the fittest individual. However, when the new generations join the population, they might not be mature enough to be competitive with the older generation. We do not want to prematurely eliminate the individuals with potentials to grow, but instead they should be allowed to be protected until optimized. We propose an island model that isolates agents to increase their survival rate. Therefore, HEHA* ensures that the new agents are protected for several iterations to be optimized.

5.0.4 Online Search

In HEHA*, the hierarchical path searching is used to eliminate search space. Best first search is used to find the top N shortest paths in the abstract map. Any regions that are not included in the top N shortest paths are eliminated. In our experiment, we observe how different values of N affect the performance and results. We want to make sure HEHA* does not eliminate areas that the optimal solution needs. After the search space reduction, HEHA* searches the remaining graph using evolved heuristic functions based on the type of regions.

5.1 Experiment and Results

We start this section by describing the benchmark we use in our experiment. Then we explain our experiment design, including the reasons for maps and problems choices, the algorithms comparison, the evaluation metric, and the platform we use for the experiment. This section ends with detail evaluation and analysis of HEHA*, showing the strengths and advantages of HEHA*.

5.1.1 Benchmark

We evaluate the performance of HEHA* using a benchmark introduced by Nathan Sturtevant's Moving AI Lab [31]. This is a widely used benchmark for grid-based pathfinding problems in many of the literatures. The blackened tiles are the walls or obstacles that are untraversable and the white tiles are traversable. The map format starts with the type, which is always octile, the height, the width, and the map data. In our experiment, we have converted the representations of all traversable tiles to 0 and untraversable tiles to -1. The benchmark not only contains many types of maps including mazes, real cities, and video game maps, the maps are large, diverse, and complex enough to examine the bottlenecks of pathfinding algorithms. The largest maps in the benchmark can have up to hundred thousands of walkable terrains.

5.1.2 Experiment Design

We use the same 10 maps from the last chapter. Each map we run 100 searches with the cost of at least 500 that are provided from the benchmark problem set. Unlike the experiment setup from

other literatures, we do not randomly generate the start and the goal for the following reasons. First of all, the benchmark problem set provides the optimal solution cost as comparison. Second, it is more objective to compare all these different algorithms with all problems on the same scale. There are several rules set by Sturtevant [31] we decided to follow in the experiment. We assume that diagonal moves are only possible if the related cardinal moves are both possible. That is, it is not possible to take a diagonal move between two obstacles, and it is not possible to take a diagonal move to cut a corner. Such an assumption is made because in a real game all creatures occupy some volume of space and are not able to pass through blocked corners. Moreover, octile neighborhood relation, meaning the adjacency relationship in 4 straight and 4 diagonal directions is used in all the maps. A search agent that is allowed to move to one of its up to eight neighbors at any given time during the search as long as it is traversable. The cost of vertical and horizontal movements are 1 and any diagonal movement has the cost of $\sqrt{2}$, which approximately equals 1.414.

We design the experiment compares HEHA* with our previous works, EHA* and RDA, and several popular pathfinding algorithms including A*, Weighted A*, HPA*. To compare the performance between these algorithms, preprocessing time and memory consumption, execution, solution optimality, and the priority queue size are evaluated. All algorithms in the control experiment are using Manhattan distance as the heuristic function to avoid any bias. For Weighted A*, we examine two weight values, 2 and 10. For HPA*, we implemented two versions of clustering, a 4x4 clustering and a 8x8 clustering to observe how different numbers of clusters would affect the results. Moreover, we observe the heuristics designed by HEHA*. The advantage of parallel computing is not a concern in this paper, even though it would benefit HEHA* the most. To the fairness of each algorithm, all experiments are single threaded and running on the same machine: it has an Intel i7-4790k CPU at 4.0 GHz, with 32.0 GB of DDR3 RAM. The algorithms are implemented using python 3.6.9 and executed on the Windows Subsystem for Ubuntu.

5.1.3 Results and Analysis

As the de facto pathfinding algorithm, A* guarantees optimal solutions. However, A* does not scale well and it performs poorly when the map is large and complex. Our experiment result

Table 5.2: Search space reduction from HEHA*

Map	AR0012SR	AR0202SR	AR0205SR	AR0300SR	AR0411SR	AR0705SR	ht_0_hightown_a3	lt_0_lowtown_a2_n_c	Milan_1_256	tranquilpaths
Original	78614	50514	66753	68964	58581	68690	40127	31456	47798	71032
HEHA*	17062.29	10872.58	13856.66	11725.88	10878.12	13116.58	7288.64	6209.13	9515.67	15731.49
Percentage	21.70388226	21.52389437	20.75810825	17.00290006	18.56936549	19.09532683	18.16392952	19.73909588	19.90809239	22.1470464

Table 5.3: Pathfinding results comparisons of each map, part 1.

Map Name	**Algorithm**	**Preprocessing Time (ms)**	**Execution Time (ms)**	**Priority Queue Size**	**Solution Length**
	HEHA*	6861.15	84.786	3664.496	546.944141
	EHA*	22170.5	941.538	38253.1585	546.944141
	RDA	1888.23	1020.26	36754.68	546.944141
AR0012SR	A*	N/A	1560.720572	55378.7	546.944141
	$WA^*_{w=2}$	N/A	368.5408	23815.6	589.8483
	$WA^*_{w=10}$	N/A	123.9333932	15200.6	594.7211322
	HPA^*_{4x4}	4033.344	226.1625	2636.471	557.1864
	HPA^*_{8x8}	1618.797	48.801	260.8716	566.352781
	HEHA*	4965.514	73.271	2752.25	543.5058726
	EHA*	15321.713	434.91	25482.219	543.5058726
	RDA	1364.93	293.8545	18872.27	543.5058726
AR0202SR	A*	N/A	540.6978728	35921.7	543.5058726
	$WA^*_{w=2}$	N/A	358.5419397	27203	574.6321928
	$WA^*_{w=10}$	N/A	407.4872408	23196.9	607.668946
	HPA^*_{4x4}	2651.846	112.977	1497.65	558.1186
	HPA^*_{8x8}	1420.28	41.543	188.114	575.337
	HEHA*	6053.066	81.103	3269.639	545.863737
	EHA*	19210.219	526.5292	29739.6914	545.863737
	RDA	1405.35	813.8657	34160.31	545.863737
AR0205SR	A*	N/A	1074.1282	45622.3	545.863737
	$WA^*_{w=2}$	N/A	358.1313	24154.4	565.4832
	$WA^*_{w=10}$	N/A	125.0167544	15303.6	565.8932968
	HPA^*_{4x4}	2469.307797	135.5081	1841.44057	554.238
	HPA^*_{8x8}	1419.91	38.5437	172.654	561.3284
	HEHA*	6292.245	87.338	3810.19	545.2014137
	EHA*	19742.14	376.21	31123.4436	545.2014137
	RDA	1442.051	747.5585	33834.63	545.2014137
AR0300SR	A*	N/A	919.1134644	44192	545.2014137
	$WA^*_{w=2}$	N/A	65.21228	11807.5	565.2996
	$WA^*_{w=10}$	N/A	33.21919753	7560.8	579.4961538
	HPA^*_{4x4}	3098.986	172.532	2562.543	550.124
	HPA^*_{8x8}	1518.281	46.288	231.484	552.47106
	HEHA*	4988.316	75.979	3072.418	544.7160098
	EHA*	16123.06	580.519	27250.32	544.7160098
	RDA	1372.69	263.9398	17923.79	544.7160098
AR0411SR	A*	N/A	685.2459741	36483.4	544.7160098
	$WA^*_{w=2}$	N/A	535.2192	31572.8	570.3889
	$WA^*_{w=10}$	N/A	623.8719573	33978	573.5202954
	HPA^*_{4x4}	2124.628	131.518	1668.2343	558.5106
	HPA^*_{8x8}	1315.535	36.689	91.448	561.3287

shows that A* is indeed returning optimal solutions, but it searches almost the entire map to find the shortest path. Compared to A*, Weight A* speeds up the search by sacrificing solution optimality. In most of the cases, increasing the weight leads to faster search and worse results, but there are exceptions where increasing the weight is not the best idea. Our results show that when there are many local minima, higher weight value does not help to speed up performance. HPA* uses a simple partitioning method to evenly divide a map into smaller areas, which is similar to HEHA*. By precomputing and storing all intra-edges, HPA* becomes very fast during online search. However, HPA* pays the price of high precomputing time and memory consumption. Moreover, HPA* does not guarantee solution optimality.

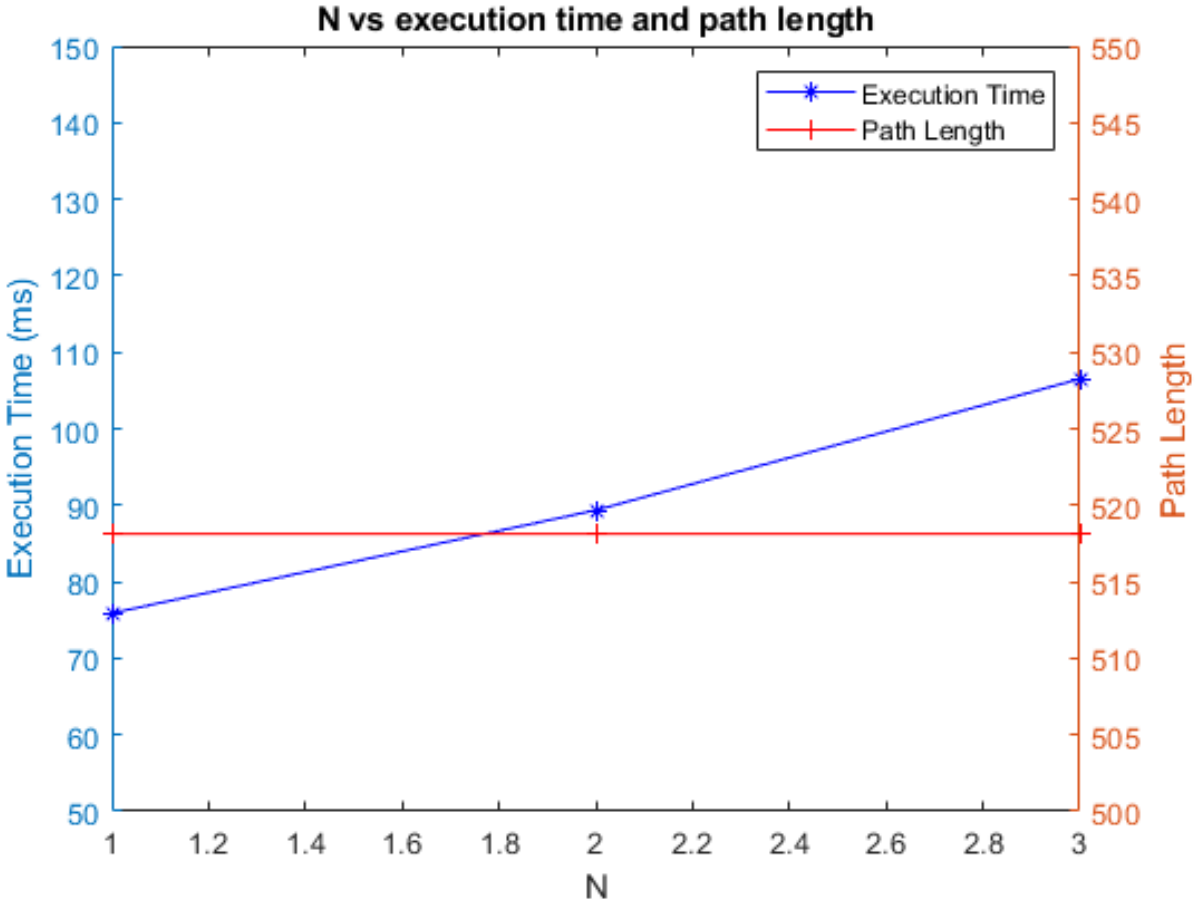


Figure 5.3: The impact of different N values to HEHA* on execution time and optimality

Table 5.4: Pathfinding results comparisons of each map, part 2.

Map Name	Algorithm	Preprocessing Time (ms)	Execution Time (ms)	Priority Queue Size	Solution Length
	HEHA*	6059.493	77.334	3101.623	544.0366158
	EHA*	19968.81	689.55	33519.35	544.0366158
	RDA	1429.677	602.5419	30318.49	544.0366158
AR0705SR	A*	N/A	883.6838537	45390.6	544.0366158
	$WA^*_{w=2}$	N/A	789.386	36104.4	569.574
	$WA^*_{w=10}$	N/A	529.3010241	29550.3	576.7480082
	HPA^*_{4x4}	2254.354	135.1494	1801.69428	556.609
	HPA^*_{8x8}	1413.316	37.4218	169.255	563.25211
	HEHA*	403.827	90.418	4267.793	538.968787
	EHA*	13087.46	136.281	18197.8	538.968787
	RDA	1282.87	151.9275	17372.39	538.9687876
ht_0_hightown_a3	A*	N/A	238.1366463	27077.4	538.9687876
	$WA^*_{w=2}$	N/A	320.8206	25388.2	567.8909
	$WA^*_{w=10}$	N/A	139.1065772	18966.9	570.3453603
	HPA^*_{4x4}	6919.537	177.2352	2591.493	552.451
	HPA^*_{8x8}	1559.72	46.9431	256.31208	560.23897
	HEHA*	3691.22	87.482	3837.206	485.7313201
	EHA*	11743.038	150.806	18852.259	485.7313201
	RDA	1323.38	158.683	18141.24	485.7313201
lt_0_lowtown_a2_n_c	A*	N/A	218.0864544	24847.8	485.7313201
	$WA^*_{w=2}$	N/A	69.08735	11194.8	522.9277
	$WA^*_{w=10}$	N/A	20.52341847	6904.6	534.5561251
	HPA^*_{4x4}	2457.961	210.433	2734.592845	501.51
	HPA^*_{8x8}	1538.765	46.458	262.6916	506.347
	HEHA*	4828.71	39.538	1631.533	348.0954903
	EHA*	14295.96	38.19	5254.6431	348.0954903
	RDA	569.92	28.1783	6247.383	348.0954903
Milan_1_256	A*	N/A	42.06531859	9080.7	348.0954903
	$WA^*_{w=2}$	N/A	10.50281	4586.6	377.1334
	$WA^*_{w=10}$	N/A	5.365340042	3827.8	385.5358149
	HPA^*_{4x4}	2313.061	108.5433	1454.348	352.785
	HPA^*_{8x8}	1658.991	26.3910	85.193	356.1936
	HEHA*	6427.044	62.4913	2241.528	538.3277846
	EHA*	20913.48	217.349	19215.535	538.3277846
	RDA	1683.31	201.683	13576.9	538.3277846
tranquilpaths	A*	N/A	298.1428395	24105.7	538.3277846
	$WA^*_{w=2}$	N/A	57.20954	10072.3	630.4549
	$WA^*_{w=10}$	N/A	11.85008471	5439.9	610.7630586
	HPA^*_{4x4}	3981.162	261.1495	3461.14	549.529
	HPA^*_{8x8}	1328.492	51.3123	297.9323	552.2353

While we modified RDA and applied in HEHA* to improve the search space reduction rate, our results show that the graph partitioning process in HEHA* performs nearly as fast as RDA. On the other hand, HEHA* reduces much more search space compared to RDA while maintaining solution optimality. Table 5.2 shows how much HEHA* outperformed RDA in terms of search space reduction. HEHA* performs a lot better because RDA applies a relatively conservative approach to reduce search space, while HEHA* reduces search space more aggressively. Moreover, Figure 5.3 shows the impact of N to HEHA* in terms of execution time and solution optimality. HEHA* have the flexibility to increase the search space if needed, although our results do not show any difference in terms of solution optimality.

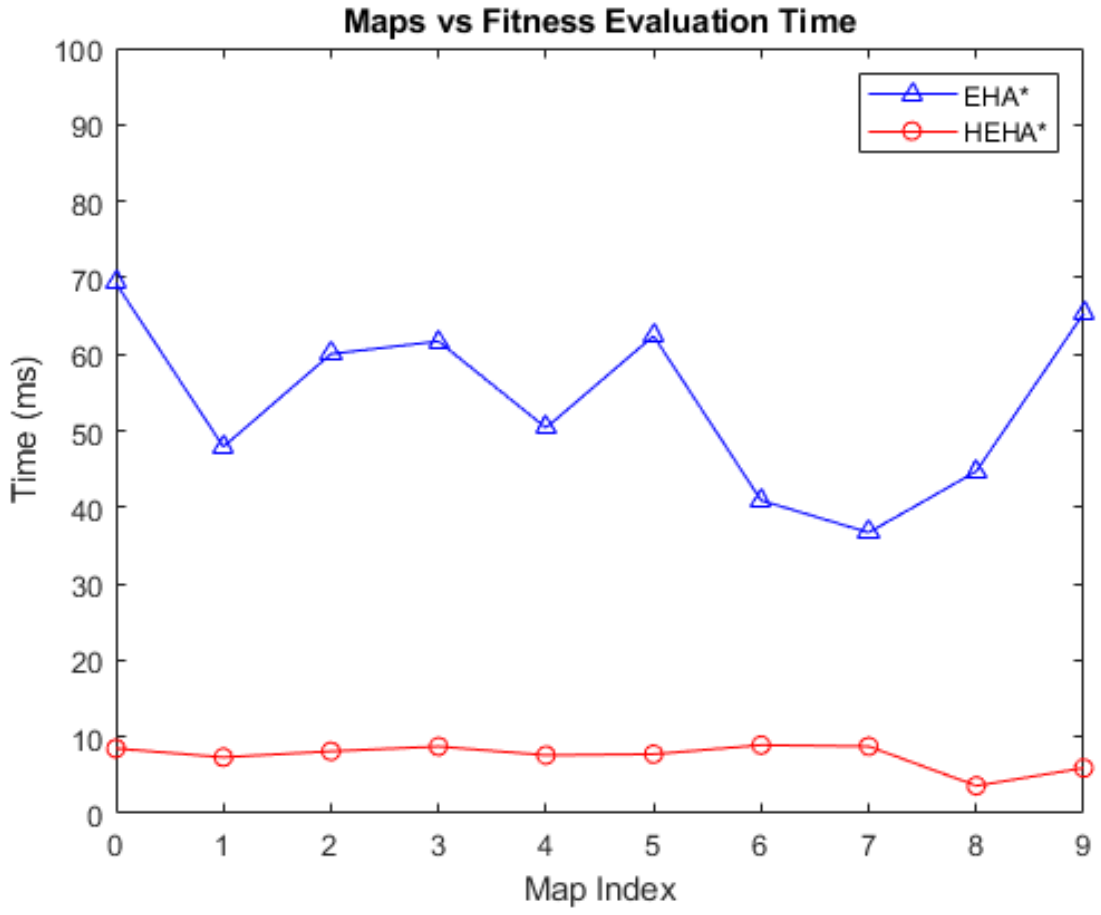


Figure 5.4: Comparisons of heuristic function evolution time in one generation between EHA* and HEHA*.

After graph partitioning, HEHA* evolves heuristic functions based on local features in the second half of the preprocessing phase. Figure 5.4 shows that HEHA* completes the optimization faster compared to EHA* for the following reasons: 1) HEHA* can evolve heuristics for different regions in parallel, and 2) even without parallelism, the heuristics in HEHA* are far more simpler compared to EHA*. HEHA* has simple heuristic functions because heuristics used in HEHA* are applied to smaller and simpler regions, while EHA* requires one heuristic function to explore the entire map. In terms of overall performance, Table 5.3 and 5.4 shows the comparisons among all algorithms we used in the experiment, and HEHA* outperforms all algorithms in almost every aspect.

5.2 Summary

In this chapter, we presented a novel hierarchical pathfinding algorithm named Hierarchical Evolutionary Heuristic A* (HEHA*). HEHA* clusters search space and evolves heuristic functions based on local features of regions. Our experimental results show that HEHA* proves the need of 1) multiple heuristics in pathfinding and 2) evolutionary heuristic function. Compared to the A* variants we chose in our experiment, HEHA* show better overall search performance. HEHA* balances the solution optimality, the memory space occupation, and the optimization time. By applying Genetic Algorithm on designing heuristic functions based on local features, HEHA* shows better search performance compared to traditional A* search and hierarchical A* search. HEHA* reduces search spaces in abstract search and applies multiple heuristic functions in local search to achieve maximum performance. HEHA* shows promising results in our experiment that we believe it has the potential to solve complex planning problems in other domains, especially for pathfinding problems with uncertainties, such as multi-agents pathfinding (MAPF) problems. Our ideas for future work includes MAPF version of HEHA*, and hardware accelerator for the preprocessing speed-up.

6. MULTI-AGENT PATHFINDING WITH HIERARCHICAL EVOLUTIONARY HUERISTIC A*

Multiagent pathfinding (MAPF) problem is an important topic to various domains including video games, robotics, logistics, and crowd simulation. The goal of a pathfinding algorithm for MAPF is to find the shortest possible path for each agent without collisions with other agents. Search is among the most fundamental techniques for problem solving, and A* is the best known heuristic search algorithm. While A* guarantees to find the shortest path using a heuristic function, it cannot handle the large scale and many uncertainties in MAPF. The main challenge of MAPF is the scalability. The problem complexity grows exponentially as both the size of environments and the number of autonomous agents increase, which becomes more difficult for A* to compute results in real time under the constraints of memory and computing resources. To overcome the challenges in MAPF, distributed approaches are introduced to reduce the computational time and complexity. Contrast to centralized approaches, which use a single controller to determine every move of all agents, distributed approaches allow each agent to search for its own solution. Distributed MAPF algorithms need to refine solutions for all agents that are collision-free. The algorithm should lead agents to take another path, or standby on the same node at the moment, to avoid conflicts between any two paths. Under the circumstances, an optimal solution is no longer simply finding the shortest path for each agent. Instead, it should contain a collision-free path for every agent, with the lowest makespan, which is the number of time steps required for all agents to reach their target. However, minimizing the makespan and the sum of cost for all agents is a NP-hard problem. Given MAPF problems often require to be solved in real time with limited resources, minimizing only the makespan is a more practical approach.

In this chapter we improved our previous work [4] and propose a multi-agent version of the Hierarchical Evolutionary Heuristic A* (MA-HEHA*) search algorithm to overcome the MAPF challenges, i that is inspired by the traffic control system in this paper. MA-HEHA* combines the power of the hierarchical pathfinding, pattern database, multi-heuristic-functions, and evolution-

ary algorithm to maximize the search quality and performance. MA-HEHA* first decomposes a search graph into many small areas based on structure of the environment for search complexity reduction. Moreover, it identifies areas that are possible bottlenecks for MAPF problems such as narrow hallways, so that agents may choose a different route or wait for the area to be cleared to avoid collisions with others. Second, MA-HEHA* creates an abstracted map to increase search performance while maintaining the solution quality. The last feature of MA-HEHA* is evolutionary heuristic functions for A* search. MA-HEHA* designs and evolves heuristic functions for each region to maximize the searchability. These heuristic functions provide guiding power for agents to behave properly during search to reach the destination in the shortest time while avoiding any collision.

To achieve accurate search and high scalability, a MAPF algorithm must fulfill the following requirements: 1) it is capable to compute collision-free paths for all agents; 2) it can provides an accurate priority decision mechanism to ensure solution optimality; and 3) it should maintain the successful rate to obtain a solution as the number of agents increases. In this paper, we proposed a novel hierarchical pathfinding technique named Multi-Agent Hierarchical Evolutionary Heuristics A* (MA-HEHA*). Our contributions in this paper are: 1) we propose MA-HEHA* that can identify bottleneck areas to reduce collisions in abstract search; 2) our algorithm evolves heuristic functions by itself to avoid potential conflicts during local search; 3) we prove that MA-HEHA* maintain high successful rate when the scalability is high; 4) we evaluate MA-HEHA* on different types of MAPF problems to show its effectiveness. Our experiment results show that our MA-HEHA* can efficiently solve large scale MAPF problems compared to traditional MAPF approaches.

To evaluate MA-HEHA* and show that it can overcome the challenges, we choose a widely used grid-based pathfinding benchmark created by Moving AI Lab [31] [43]. The benchmark provides a large amount of various maps extracted from video games and real cities. We modified the benchmark by implementing three scenarios introduced from above to evaluate MA-HEHA*. In the experiment, we compared MA-HEHA* with a distributed version of A* [12]. Algorithms

are compared among execution time, memory consumption, solutions quality, and successful rate as the scale goes up.

6.0.1 Multi-Agent Pathfinding Problems

Multi-agents pathfinding (MAPF) problems are commonly found in many different domains including video games, robotics, logistics, and crowd simulation. We define MAPF as a set of n agents $\{A_{1...n}\}$ to find the shortest path from a set of starting nodes $\{S_{1...n}\}$ to a set of destinations $\{D_{1...n}\}$ in a graph G(V, E) [43] [44]. To solve MAPF, a search algorithm needs to compute a set of paths for all agents that no path at any point collides with each other. Moreover, the search algorithm needs to minimize the overall cost to be considered optimal. Therefore, the complexity of MAPF increases exponentially as the environment size and the number of agents increase. MAPF is considered as a dynamic search problem because they have multiple agents searching for a solution simultaneously. Agents are required to either re-plan new routes or avoid conflicts on the fly, which increases the difficulty and computation time to solve the problem.

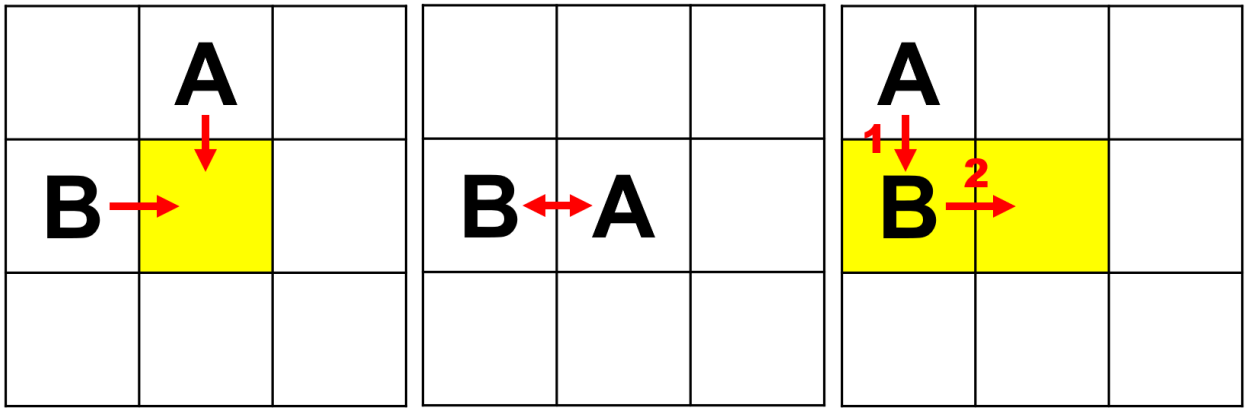
Based on observations, MAPF problems can result in three different scenarios. These scenarios are not mutually exclusive, meaning they can be found together in the same problem. The first scenario is called the herding problem. In the herding problem, all agents start at the same zone and they all move to one goal zone. The second scenario is called the scramble crossing problem. In the scramble crossing problems agents are placed in different locations and each of them has their own designated goal. The last scenario is called the midpoint problem. The objective of the midpoint problem is to gather all agents together at a mid-zone of which the travel time for all agents to arrive is minimal. Each of them has its own characteristics and requires different tactics to solve, thus a specific tailored pathfinding algorithm for MAPF problems is needed to compute optimal solutions accurately.

These different scenarios show that MAPF problems have many challenges including crowd control, traffic jam avoidance, collision avoidance, and midpoint computation. To overcome these challenges, different methods for a search algorithm are required. Prior researches proposed many techniques to improve A* algorithm for MAPF problems. There are centralized computing [45],

distributed computing [46] [47], spatial distributed approach [48], hierarchical pathfinding [49], and heuristic improvement [50] [51] [52] [53]. Centralized approach is one of the most popular approaches for MAPF problems but it usually lacks scalability. Distributed approach allows agents to perform individually but it does not guarantee the quality of solutions [44]. Moreover, it is tricky to ensure that private information is not shared among agents, yet still provides computational efficiency. Hierarchical approach allows agents to process information in a smaller area, but the trade off is extra preprocessing time and memory consumption. A more complex and informative heuristic function is a straightforward solution. It processes more information and uncertainties to accurately estimate the path. The downside is that having an overcomplicated heuristic function may reduce the performance. In the end, it is difficult to compare these approaches because each of them focuses on different challenges that MAPF problems provide.

A Multi-Agent Path Finding Problem (MAPF) problem is defined as a set of n agents $\{A_{1...n}\}$ to find the shortest path from a set of starting nodes $\{S_{1...n}\}$ to a set of destinations $\{D_{1...n}\}$ in a graph G(V, E). Time is considered to be discretized into time steps, and each agent is allowed to perform one action in every time step. Agents have two types of actions: wait and move. A wait action means that an agent stays in its current vertex for one time step, and a move action means that an agent moves from the current vertex to an adjacent vertex. A path of an agent is a sequence of actions it chooses to move from start to finish. A path is considered valid if it does not have any conflict with other agents. There are three types of conflicts [43]: vertex conflict, edge conflict, and following conflict. Figure 6.1 illustrates the different types of conflicts. A vertex conflict occurs between any two agents when they plan to move to the same vertex at the same time step. When any two adjacent agents swap their current positions, an edge conflict occurs, meaning both agents attempt to travel via the same edge at the same time step. Following conflict occurs when one agent is planning to move to a node that was occupied by another agent in the previous time step.

To solve MAPF problems, pathfinding algorithms must search for a valid path. A set of valid paths is formed as each agent contributes to a valid path. As a result, the set consists of k paths. To evaluate the solution optimality, two objective functions are used. The primary objective is



(a) Vertex conflict (b) Edge conflict (c) Following conflict

Figure 6.1: Different types of conflicts in MAPF problems

to minimize the makespan which is the latest arrival time of an agent to the target. The second objective is to minimize the flowtime, or sum-of-costs, which is the total cost of all agents to reach their target locations. To compare the quality of solutions, the makespan must be considered as the primary criteria.

MAPF is an NP-hard problem [54] because the search space grows exponentially as the number of agents increases. As a result, traditional pathfinding approaches have troubles to keep up the performance when the problem scale increases. Moreover, MAPF is a dynamic pathfinding problem. The environment changes as agents move on the search space. Agents become obstacles to each other and they need to avoid each other during search. The dynamic environment increases the uncertainty of the problem that makes the algorithm more difficult to accurately estimate the path cost. A more practical approach to solve MAPF is to maintain the minimum makespan while relaxing the flowtime.

6.0.2 Prior Research Works

A* is the de facto pathfinding algorithm in many domains including MAPF problems. A MAPF problem can be solved by both the centralized approach or the distributed approach. The centralized approach sees a MAPF problem as a k-agents-placement problem. Therefore, the action A* has to determine is to place k agents into their adjacent vertices for every time step. Each action

is valid only when there is no conflict between any agents. The advantage of the centralized A* is that the algorithm has the complete knowledge of the environment to guarantee that every move is conflict-free. However, A* for MAPF suffers from scalability issues. The size of the search space grows exponentially with the number of agents, meaning the approach is computationally intensive.

Since the MAPF problem is more complicated than traditional pathfinding problems, A* needs to have a more complex heuristic function to minimize the number of expanding states. Multi-label A* Algorithm (MLA*) [45] introduced a new centralized h-value-based heuristic that plans tasks for all agents each step to eliminate the restructuring process. It is a central controller that requires positions of all agents to compute. MLA* reduces all invalid moves in the search tree to shorten the execution time. However, the search space can still be too large to be practical.

Instead of relying on a centralized system to plan every action for every robot, decentralized approaches not only are more efficient, they are more suitable for parallel computing. Although decentralized approaches usually do not guarantee completeness and optimality, they provide much higher scalability to be practical for real world applications. Decentralization approaches divide pathfinding in three phases: 1) independently compute paths for agents regardless of conflict, 2) decide the priority of the agents for plan coordination, and 3) restructure final solutions to resolve conflicts between agents [46]. Phase one is equivalent to a traditional single agent pathfinding problem, hence reducing the computational complexity. Phase two is an optimization problem to minimize the number of conflicts so that less restructuring is needed. Phase three is simply a search for alternative routes for agents with lower priority to resolve conflicts. Compared to centralized heuristic functions that have complete knowledge of the environment, distributed heuristic functions have limited information about other agents. Therefore, distributed approaches provide higher flexibility because each agent has its own heuristic function.

Although decentralized approaches reduce computational intensity and complexity, they often still require to search the same space multiple times. Hierarchical approaches like MA-NHA* [49] maximize data reuse by decomposing the search space into many regions and precomputing intra-

edges within regions. Because lots of subpaths are shared among agents in MAPF problems, precomputed paths vastly reduce online execution time and memory consumption. Moreover, with the search space divided into many regions, path restructuring can be done locally without searching the entire search space every time. Thanks to the paths precomputation and hierarchical approach, MA-NHA* often provides much faster computation compared to other techniques, even though it does not guarantee optimal solutions.

6.1 Multi-Agent Hierarchical Evolutionary Heuristic A* (MA-HEHA*)

6.1.1 Algorithm Overview

MA-HEHA* is inspired by the traffic control system that agents follow certain rules based on the environment conditions to avoid collisions. We present a new version of our previous work HEHA* [4] to adopt the multi-agent setting. In the original HEHA*, the algorithm serves two purposes, it decreases the problem complexity by decomposing the search graph into many smaller areas, and it evolves heuristic functions based on the local features of these smaller areas. Similar to HEHA*, MA-HEHA* has two phases, the offline preprocessing phase and the online pathfinding phase. Figure 6.2 illustrates the procedures of MA-HEHA*.

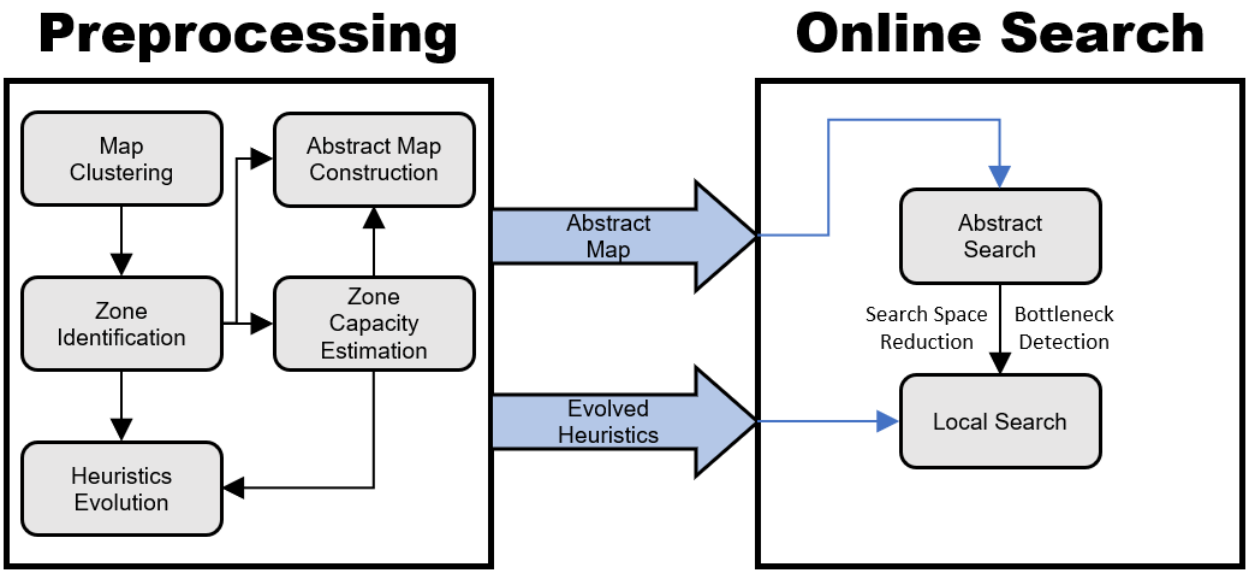


Figure 6.2: MA-HEHA* algorithm overview

In the offline preprocessing phase, MA-HEHA* clusters a search graph and creates an abstract graph based on local features for hierarchical path searching. Moreover, MA-HEHA* categorizes the decomposed areas into different levels based on how likely conflicts occur in the area. The partitioning algorithm in HEHA* is inspired by our previous work named Regions Discovery Algorithm (RDA) [3]. Unlike traditional hierarchical pathfinding approaches, MA-HEHA* does not precompute and store any actual path speed up search performance. Instead, MA-HEHA* uses estimated distances between borders of regions as abstracted edges. This approach reduces the preprocessing time and provides accurate search. After the abstract map is created, MA-HEHA* categorizes regions into several types based on the number of borders they have. For each region type, MA-HEHA* evolves a specific heuristic function that maximizes the search performance and develops the ability to avoid collisions. Regions with similar characteristics share the same heuristic function. This approach vastly reduces the time and memory consumption during the evolutionary process.

The online pathfinding phase is separated into two processes, which are the abstract search and local search. MA-HEHA* first searches the abstracted graph for each agent. After abstract paths are computed, MA-HEHA* decides the priority of agents based on the estimated cost of each agent to travel from its starting node to the goal node. The agent that has the longest travel distance has the highest priority, therefore it goes first to ensure minimal makespan. Moreover, MA-HEHA* identifies potential conflict zones and notifies involved agents. Once the abstract search is completed and all related information is sent to agents, each agent then searches independently on the local level for the final solution. During the local search, if there are any potential conflicts between agents, agents with lower priority have to yield for higher priority agents.

6.1.2 Map Abstraction

MA-HEHA* clusters the search space into many regions with different shapes based on its local features. The decomposition algorithm is modified based on our previous work Regions Discovery Algorithm (RDA) [3]. MA-HEHA* analyzed more information of the abstracted map to prevent conflicts among agents. MA-HEHA* categorizes each area based on the likeliness of

having conflicts. For instance, areas with narrow pathways are more likely to have conflict when many agents are trying to get through. MA-HEHA* identifies these areas and divides them into five levels, from lowest to highest level. Agents work differently based on different levels of likeliness to conflict. When the first agent enters a bottleneck area, it broadcasts and warns other agents. If any other agent plans to enter the same area, the heuristic function will determine the most appropriate action, either waits or detours, to avoid any potential conflict.

6.1.3 Heuristic Function Evolution

MA-HEHA* applies Genetic Algorithms (GA) to evolve heuristic function based on local features of regions decomposed from the previous phase. MA-HEHA* optimizes heuristic functions for different types of regions. A set of agents, a.k.a population, has to solve different problems in different regions repeatedly to evolve heuristic functions. Based on the principle of survival of the fittest, these agents compete and evolve over generations. Each agent has a heuristic function that is represented in a numerical form called a chromosome. A chromosome is a combination of genes, which is the basic building block of a heuristic function. A gene is either a weight value or a heuristic index. The chromosome is formed by the sum of weighted heuristic function: $\sum_{i=0}^{n} w_{\mathrm{i}} * w_{\mathrm{i}}$. By combining the search power of different heuristic functions, MA-HEHA* increases the search ability and accuracy of the heuristic function.

Figure 6.3 shows the flow diagram of the evolutionary procedure. The heuristic function evolving process starts with population initialization. Each agent is assigned with a basic heuristic function from the gene pool with the weight value of 1 to solve the given problem. After solutions are obtained, MA-HEHA* evaluates and ranks agents based on the optimality of their solutions. After evaluation and ranking, the next generation of population is created based on the evaluation result. HEHA* evolves these agents via selection, crossover, and mutation. In the selection phase, the best agents are chosen to create the next generation. The chosen agents are either unchanged to the new generation, or crossover between two agents to create new offspring. On top of crossover, the new offspring has a small probability to mutate, meaning one of the genes is changed randomly. Crossover exchanges a pair of genes we named substring, which is one heuristic index with its

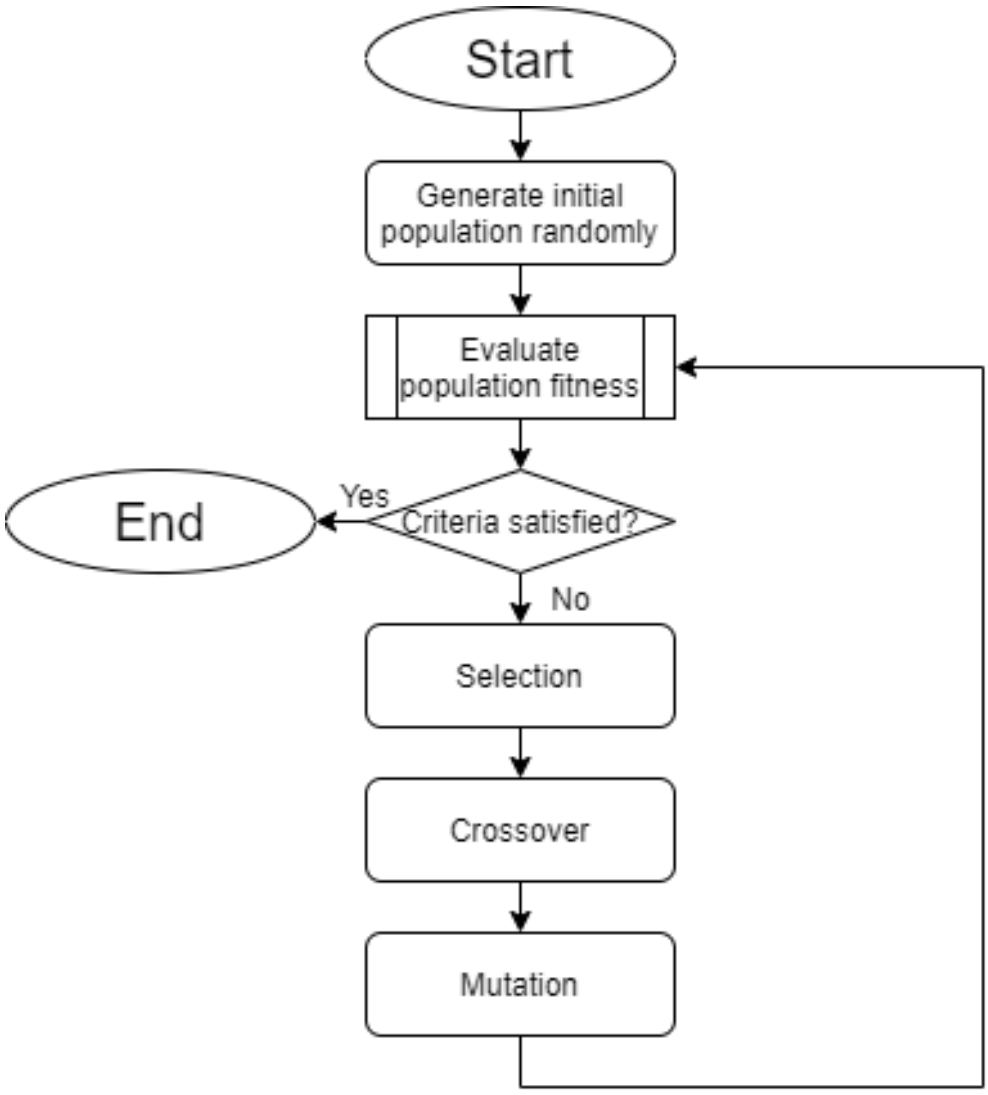


Figure 6.3: Genetic algorithm flow diagram

corresponding weight, from each parent randomly. After exchanging substrings, the weight will be recalibrated to ensure the sum of weights equals to one. After the new generation is created, repeat the fitness evaluation process until the preset criteria are met. In our case, the only criteria is the maximum number of iterations.

While GA is an efficient method to optimize heuristic functions, the algorithm can trap in local minima if not handled correctly. There is a need to protect the underprivileged new generations to avoid prematurely eliminating the individuals that have potentials. To achieve this goal, we

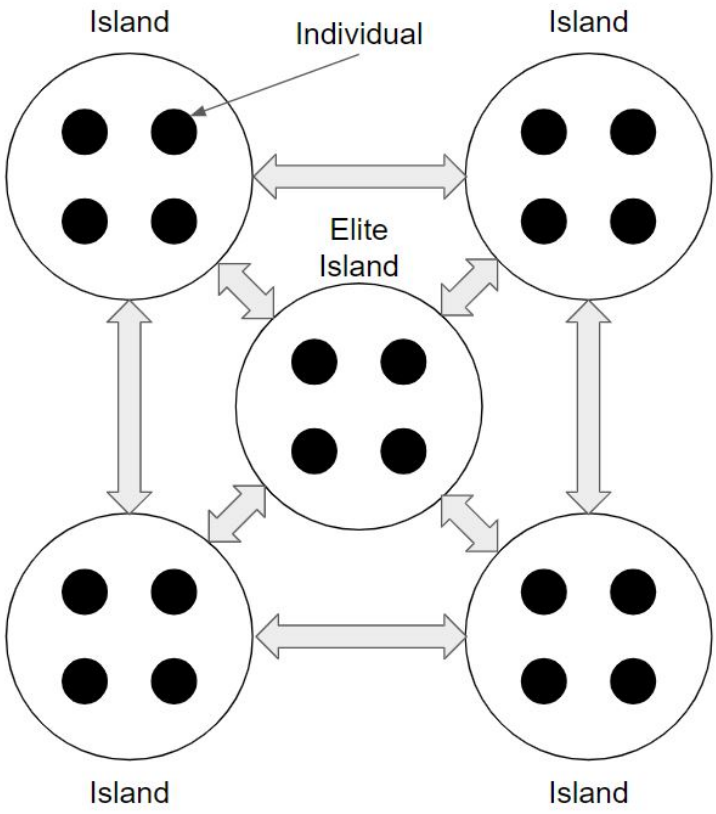


Figure 6.4: Genetic algorithm island model implementation.

propose to implement an island model that isolates agents to maximize their survival rate until they are ready. The island model separates agents into multiple batches. Agents only compete within agents until they are fully grown. Therefore, MA-HEHA* ensures that the new agents are protected for several iterations to be optimized. Figure 6.4 shows the general concept of our GA implementation. In our experiement, we divided our agents into n groups after the first iteration. One of the islands is an elite-only island. For every m iterations, inter-islands reproduction is allowed to generate new species. The best candidates from each island are moving into the elite island, while the rest will be removed from the competition.

6.1.4 Abstract Search

After the preprocessing phase is completed, MA-HEHA* moves on to the online pathfinding phase. The pathfinding phase is separated into two processes, the abstract search and the local

search. Abstract search serves two purposes: to determine priorities of agents and reduce search space for each agent. First MA-HEHA* has each agent searches on the abracted map to get estimated distance. Then MA-HEHA* decides the priority of agents based on the estimated distances. In traditional distributed approaches, priority decision doesn't happen until all paths are found. Thus path refinement is needed and that leads to multiple searches for each path until all paths are collision-free. The advantage of our method is that the estimation is highly accurate, the priority decision can depend on estimated distances instead of actual paths. With agent priorities decided before the local search begins, agents with lower priorities can now avoid conflicts during search by giving ways to higher priority agents.

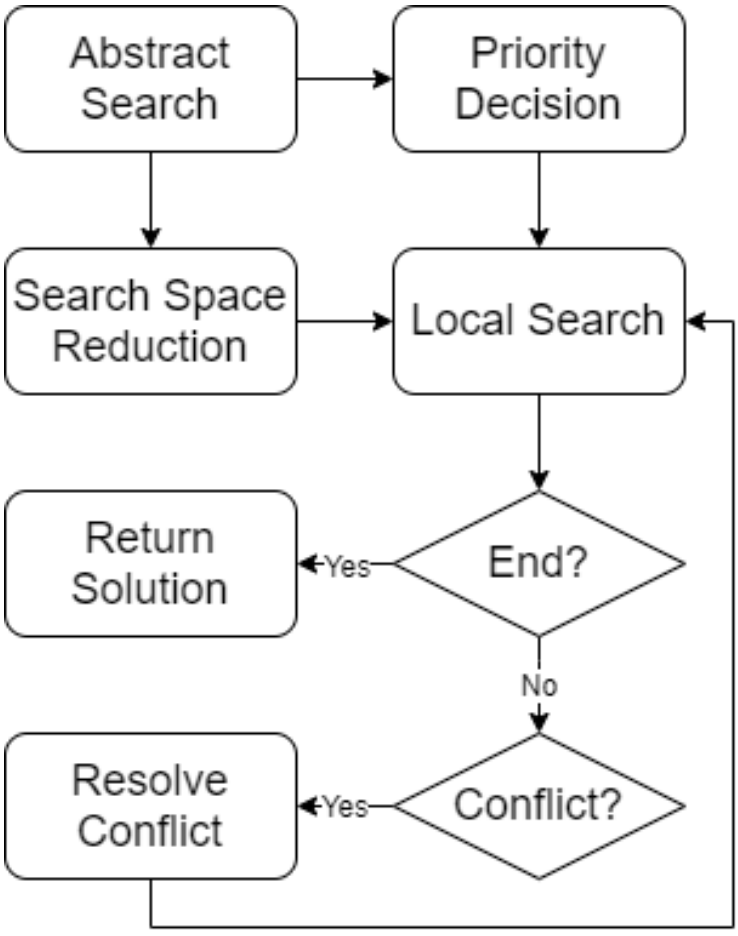
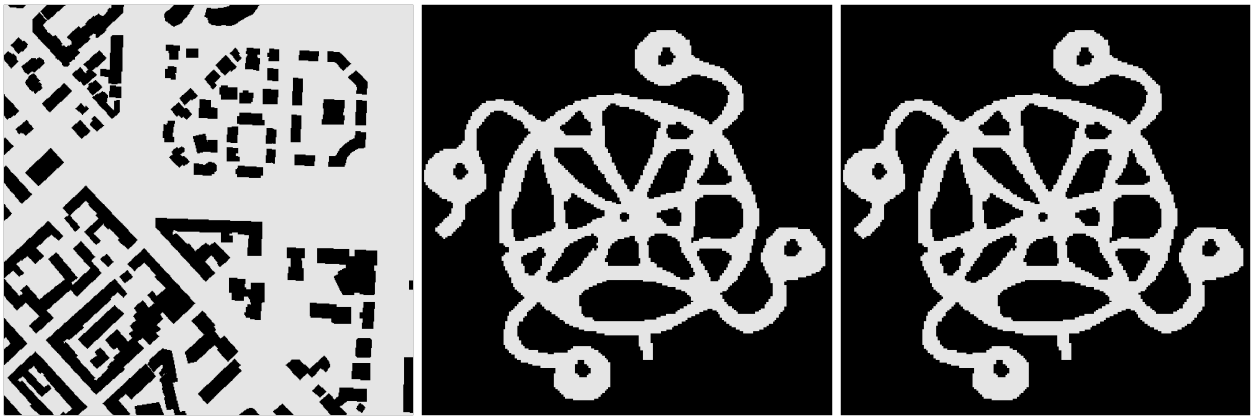


Figure 6.5: Online pathfinding procedure of MA-HEHA*.

6.1.5 Local Search

During the local search, each agent independently and simultaneously searches for the optimal solution. Based on the traffic condition, the heuristic function guides agents differently. When agents enter a bottleneck area, they broadcast the entered region, the timestep-stamp, and the coordinate, so that other agents may choose to wait to enter the same region, or to take another route, depending on the given situation of the area. While the wait action does not increase the travel distance of an agent, it does cost extra time for an agent to reach the destination. In our experiment, we consider the cost of each wait is 1. To maintain the solution optimality with our best effort, agents are not allowed to take paths that cost more than the makespan when there are other options available. The local search ends when all agents find their collision-free paths. Figure 6.5 shows the procedure of the online pathfinding process, from abstract search to local search.

6.2 Experiment Design and Results



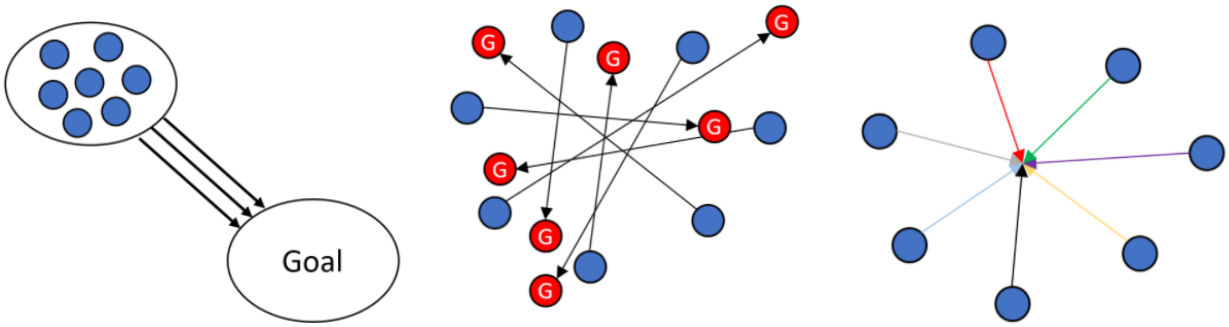
(a) Open Space (b) Narrow Hallways (c) Intersections

Figure 6.6: Different types of map for performance evaluation.

6.2.1 Experiment Design

We have modified a widely used grid-based benchmark that is introduced by Nathan Sturtevant's Moving AI Lab [31]. We chose 3 different types of map with the size of 512x512 to evalu-

ate MA-HEHA*. These 3 types are open space, narrow hallways, and intersections, as Figure 6.6 shows below. The blackened tiles are the walls or obstacles that are untraversable and the white tiles are traversable. We designed three scenarios that are commonly found in MAPF problems for each map. Our goal is to observe how does MA-HEHA* perform differently on each map with different scenarios as the number of agents increases. Figure 6.7 illustrates these 3 scenarios, which are the herding problem, the scramble crossing problem, and the midpoint problem. For each problem, we start with 10 agents and increase up to 100 agents.



(a) The Herding Problem

(b) The Scramble Crossing Problem

(c) The Midpoint Problem

Figure 6.7: Different types of conflict happens in MAPF problems.

The herding problem is often seen in strategy video games [55]. All agents in the herding problem start at the same zone and move to the goal zone. The main challenge of the herding problems is that some regions are not wide enough to fit all agents at once, causing them to wait or detour frequently. The second scenario is scramble crossing, agents are placed in different locations and each of them has their own designated goal. In the scramble crossing problem, agents often collide with each other. Pathfinding could be unpredictable which increases the computation time. The last scenario is the midpoint problem. Agents in the midpoint problem are required to cooperate to find a mid-zone to meet so that the makespan is minimized. The challenge of the midpoint problem is that the goal zone is not given. Agents need to estimate a mid-zone that is fair for every agent, meaning the travel distance of each agent is similar. Otherwise the makespan

increases because some agents have to travel much further.

6.2.2 Evaluation Metrics

To objectively evaluate the MA-HEHA* algorithm, we would observe the following metrics with different number of agents: the successful rate of MA-HEHA* to compute optimal solutions and the number of conflicts as the number of agents increases. These two metrics show the scalability of a multi-agent pathfinding algorithm. In this experiment, a distributed version of A* are used to compare with MA-HEHA*. Each agent searches its own path, and restructures the path if there is any conflict in between. A* algorithm is chosen because it is one of the most popular pathfinding algorithms in many domains. It is practical to be applied on all types of pathfinding problems due to its simplicity and robustness. Most importantly, A* guarantees to return optimal solutions. The procedure of distributed A* for MAPF is described as follows:

1. Each agent searches an optimal solution independently.

2. A centralized controller decides the priorities of agents.

3. Refine each solution if it is conflicted with other agents.

6.2.3 Results

Figure 6.8a shows the results comparison between MA-HEHA* and A* of solving the herding problem in the open space map. Throughout the experiment, MA-HEHA* shows the capabilities of avoiding collisions during searches, resulting in less collisions to occur and maintain high success rates even when the number of agents increases. The biggest challenge in herding problem is to ensure agents in the back would not overtake agents in the front. In the open space map, MA-HEHA* accurately decides the agent priorities and efficiently leads agents to detour when needed. MA-HEHA* leads to less number of conflicts and higher success rate to find optimal solutions compared to A*. The high number of conflicts occurs in A* is unavoidable since each agent searches independently. As the number of agents increases, the complexity of refinement increases exponentially, thus leading to high failure rates.

Compared to the herding problem, encountering collisions in the scramble crossing problem happens more frequently. Inspired by the traffic control system, MA-HEHA* leads agents to lean right when they enter potential conflict zones. This move vastly reduces the number of conflicts compared to A* as Figure 6.8b presents. Even when multiple agents search in the same zone, conflicts rarely occur because the heuristic function guides agents to avoid taking center paths. This tactic works particularly well in open space areas. On the other hand, A* has a very low success rate because the number of conflicts is too high. Refinement is basically impossible to achieve with limited computation power and resources.

The midpoint problem is a little different from the first two problems. Conflict occurrences are far less compared to the first two problems. The algorithm requires to compute the midpoint so that every agent has a similar travel distance. MA-HEHA* and A* first scan the entire map and obtain estimated distances using the Euclidean metric between the chosen midpoint and all agents. The node with the lowest standard deviation of estimated distances is chosen to be the midpoint. The equation to calculate the standard deviation is $s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$. Compared to A*, MA-HEHA* scans in the abstract map thus leads to faster search and more accurate result. Figure 6.8c shows that MA-HEHA* has a very high success rate and low numbers of conflict.

Compared to the open space map, it is more difficult to solve the herding problem in the narrow hallways map, because the number of conflicts is higher when agents travel. While MA-HEHA* works relatively well throughout the experiment, the success rate drops significantly as the number of agents increases as Figure 6.9a shows. Conflicts occur more frequently and often there are no alternative routes for agents to take. Agents are forced to wait in their current positions until the traffic jam is cleared. The traffic jam causes the makespan to increase which fails to obtain optimal solutions.

MA-HEHA* solves the scramble crossing problem in the narrow hallways map almost as good as solving in the open space map. MA-HEHA* effectively guides agents to avoid collisions, showing promising results even when the scalability increases. On contrast, A* cannot handle the high number of collisions and causes failure during paths refinement. Figure 6.9b shows the perfor-

mance differences between MA-HEHA* and A*.

To solve the midpoint problem, an algorithm needs to determine the destination first. The standard deviation of estimated distances from A* is too high, thus the solution cannot possibly be optimal. Although the number of conflicts is low, but the high failure rate is caused by A* requiring recomputation too many times to optimize the solution. Compared to A*, MA-HEHA* uses the abstract map to determine the midpoint, which is far more accurate. No re-computation is needed to minimize the standard deviation. Our results in Figure 6.9c show that MA-HEHA* successfully obtain solutions even when the number of agents is high.

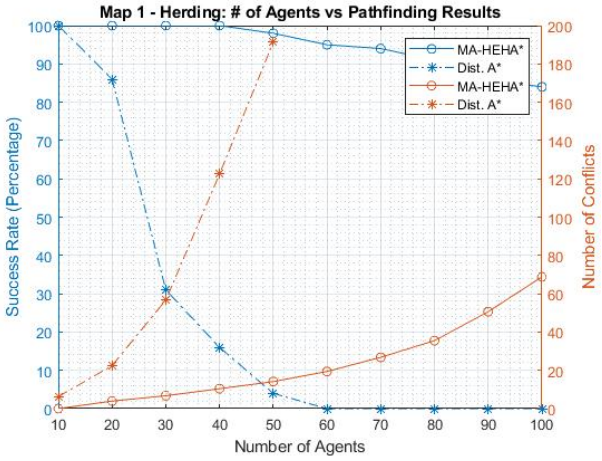
Solving the herding problem in the last map is more interesting than the last two maps. On one hand it has many narrow tunnels that potentially cause traffic jams. On the other hand there are many alternative routes to take if the heuristic function is intelligent enough to make the decision early on to avoid collisions. Results of MA-HEHA* shows in Figure 6.10a proves that MA-HEHA* is capable to do so thus it maintains a very low number of conflicts and high success rate.

Same as the herding problem, MA-HEHA* has no problem solving the scramble crossing problem in the intersections map. Although MA-HEHA* cannot always switch to alternative paths during abstract search, but the regional recomputation is simple and effective to avoid traffic collisions in narrow spaces. MA-HEHA* has a much lower computation time compared to A* since MA-HEHA* enables the opportunity to partially recompute the solution path. Figure 6.10b shows the results of MA-HEHA* solving the scramble crossing problem on map 3.

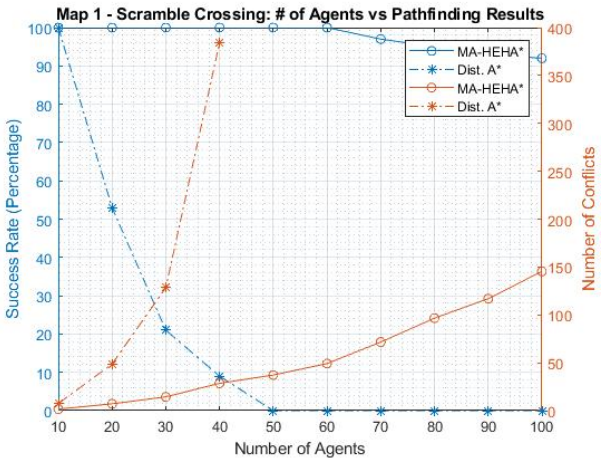
The midpoint problem shows how inaccurate a simple heuristic function like the Euclidean distance is. While it is an admissible solution, estimating the travel distance in a maze-like map using the Euclidean distance is unreliable. Figure 6.10c shows A* has a higher fail rate because of this particular reason. MA-HEHA* has the advantage of using hierarchical pathfinding for a simpler and more accurate estimation. This advantage leads to faster search and fewer errors when searching for an optimal solution.

6.3 Summary

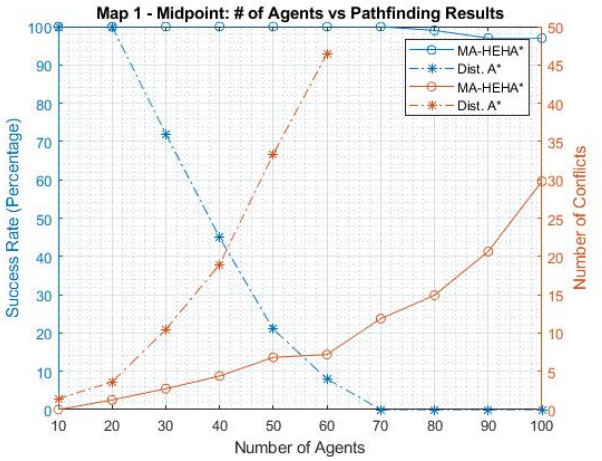
In this chapter have introduced a novel hierarchical approach named Multi-Agents Hierarchical Evolutionary Heuristic A* (MA-HEHA*) algorithm in this paper to improve search performance and scalability to solve MAPF problems. We evaluated MA-HEHA* with multiple common MAPF problems on different types of map to show the effectiveness of the algorithm. The contributions of MA-HEHA* are threefold: 1) MA-HEHA* decomposes a grid-based map in many small regions and identifies high traffic areas, 2) MA-HEHA* evolves heuristic functions based on local features to increase search performance and avoid potential collisions, and 3) MA-HEHA* minimizes the needs for path restructuring with our hierarchical approach. Our experimental results show that MA-HEHA* is capable to handle MAPF problems with many agents. Moreover, MA-HEHA* shows that it is distributed-system-ready: no centralized controller is needed for MA-HEHA* to operate.



(a) Map 1: Herding

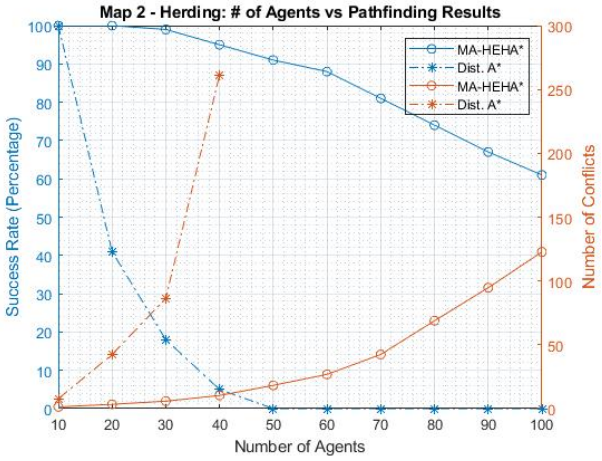


(b) Map 1: Scramble Crossing

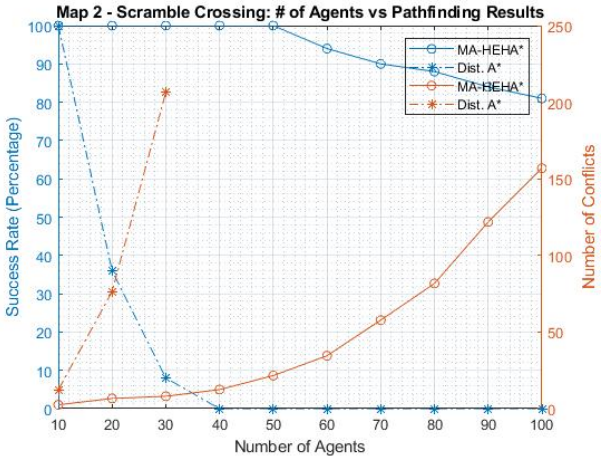


(c) Map 1: Midpoint

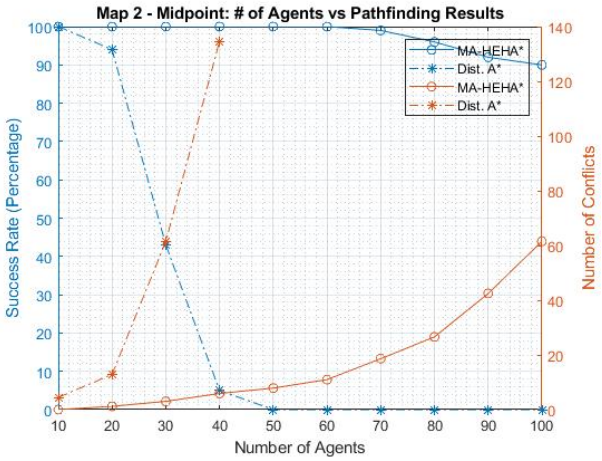
Figure 6.8: Results comparison of the open space map.



(a) Map 2: Herding

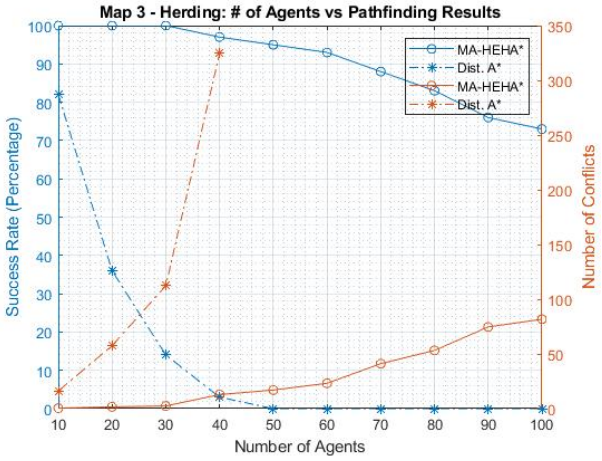


(b) Map 2: Scramble Crossing

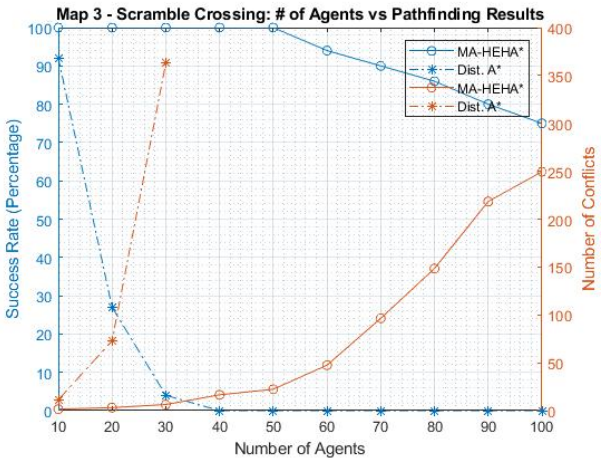


(c) Map 2: Midpoint

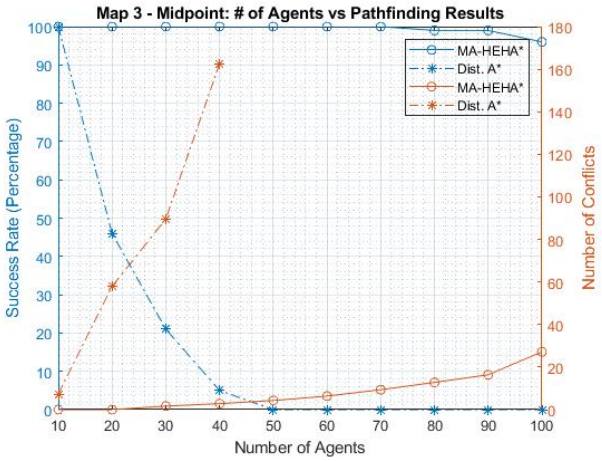
Figure 6.9: Results comparison of the narrow hallways map.



(a) Map 3: Herding



(b) Map 3: Scramble Crossing



(c) Map 3: Midpoint

Figure 6.10: Results comparison of the intersections map.

7. HEURISTIC FUNCTION EVOLUTION FOR PATHFINDING ALGORITHM IN FPGA ACCELERATOR

The Genetic Algorithm was proposed by Holland [29] that was inspired by Darwinism. The algorithm evolves computer programs in the process of competition and natural selection. Researchers have been successfully applied Genetic Algorithm to solve complex problems from different domains including the traveling salesman problem [56], VLSI partitioning [57], Machine Learning [58], Neuroevolution that uses the Genetic Algorithm for Reinforcement Learning tasks [30] [59] [60], and other optimization problems that have complex constraints [61] [62]. The Genetic Algorithm shows promising results solving these problems. In our previous work, Evolutionary Heuristic A* (EHA*) [1] [2], we proposed to combine the power of evolutionary process with A* search [12] to reduce user efforts on algorithm design.

However, the Genetic Algorithm requires a high degree of parallelism, intensive CPU operations, and high memory bandwidth that make the CPU hit the performance bottleneck. Consequently, researchers seek to overcome the bottleneck by implementing the algorithm onto domain specific hardware accelerators such as application-specific integrated circuit (ASIC), field programmable gate array (FPGA), and graphic processing units (GPU). The motivations for designing Genetic Algorithm hardware accelerators due to the fact that Genetic Algorithm has the potential to achieve high performance computing by utilizing deep pipelining and parallel evolution processing. Not only the reproduction and mutation process to generate offsprings can be pipelined for maximum throughput, but also this pipelining process can be parallelized by duplicating the process module and executing them in parallel .

In this chapter, we proposed a high-speed FPGA accelerator for EHA* (FEHA*) that accelerates the heuristic function evolution process. The contributions of this work are listed below:

- We identify the parallelism opportunities that are unique to the Genetic Algorithm. The observation provides better insights on designing an efficient architecture for the algorithm.

- We design a FPGA accelerator for the Evolutionary Heuristic A* that exploits parallelism to improve scalability and evolution speed.
- We implement the FPGA accelerator on a Terasic DE1-SoC Development Board [63], which uses the Altera SoC FPGA Cyclone V, and evaluate it against the optimized multi-threaded CPU version of EHA*.
- We evaluate and compare both software and hardware versions of EHA* on a widely used benchmarks for grid-based pathfinding [31]. We observe up to 8x speed up.

7.1 Background

In this section, we would like to introduce some concepts and prior researches that are related to our work which we hope that it will help readers to have a better understanding of the rest of the discussion.

7.1.1 The Genetic Algorithm

The Genetic Algorithm is introduced by John Holland in the 1970s that an algorithm mimics the processes of biological evolution in order to solve problems and to model evolutionary systems [64]. It is an optimization technique based on the principle of genetics and natural selection with random features, that performs an efficient and systematic search of the solution space. Similar to how living organisms evolve to adapt to the environment, computer programs can evolve in ways that resemble natural selection can solve complex problems even their creators do not fully understand.

One of the main reasons the Genetic Algorithm is suitable for many computational problems is because they expect computer programs to be adaptive [64]. They are expected to continue to perform well regardless of the changes in the environment. This often occurs in robotics in which a robot has to perform a task, such as finding the shortest path to the destination in a dynamic environment. Finding optimal solutions in these computational problems require searching through an enormous number of possible solutions. To speed up the search, an algorithm can benefit from

exploiting parallelism by simultaneously exploring the search space and testing many different possibilities.

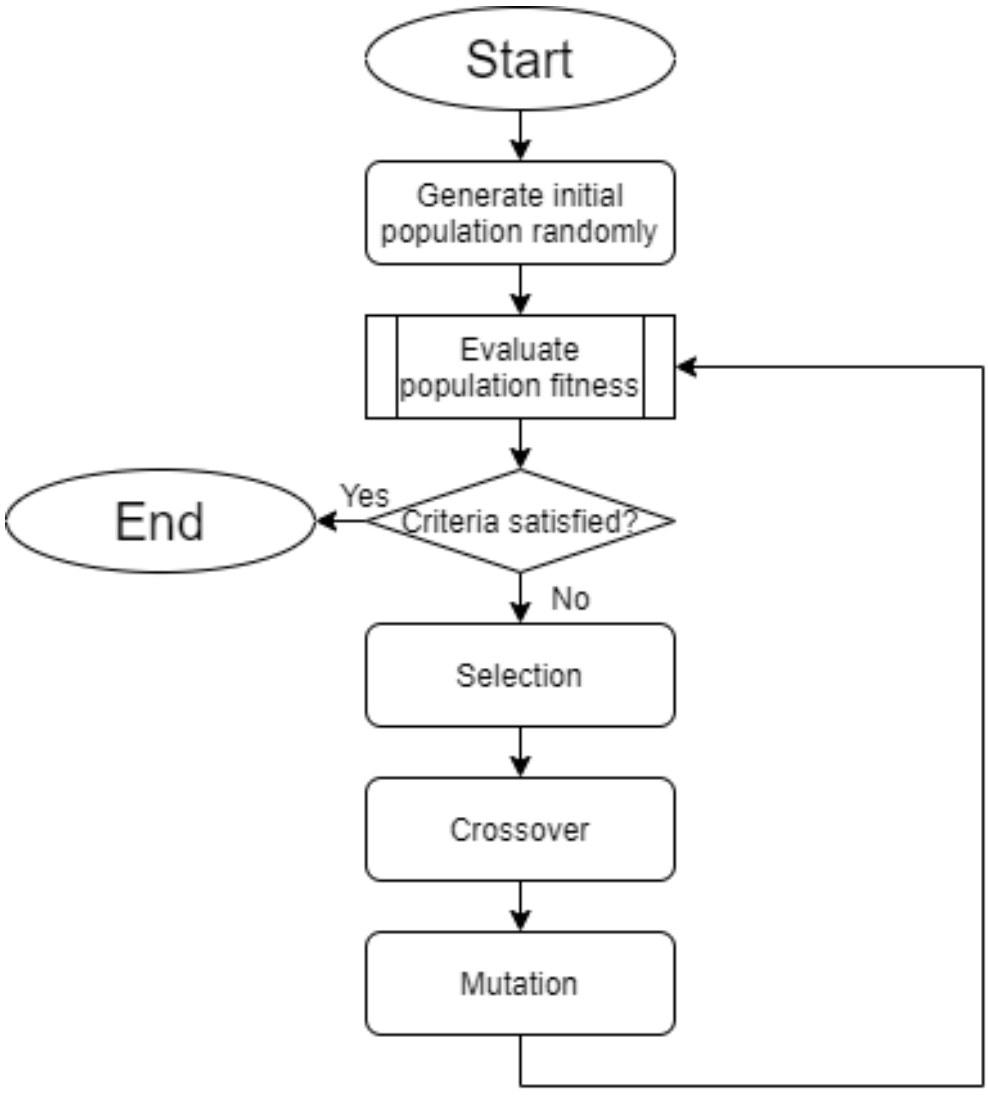


Figure 7.1: Genetic Algorithm procedure overview.

Figure 7.1 shows the procedure of the Genetic Algorithm. It begins with randomly generating an initial population of genomes. A genome is numerical representation of a solution of the given search problem. A common method of encoding them uses their integer representation. Each variable is first linearly mapped to an integer defined in a specified range, and the integer is encoded

using a fixed number of binary bits. The binary codes of all the variables are then concatenated to obtain a binary string [65]. Each genome is then evaluated by the fitness function to see how well the solution is to the target problem. Based on the fitness scores, genomes are ranked and selected to reproduce the next generation. Evolution begins in the reproduction process. Genomes that are representing better solutions to the problem are given higher probabilities to reproduce than genomes with lower fitness scores. The fitness of a solution is typically defined with respect to the current population average, or the population median [66]. After many iterations, the algorithm ends either because a good enough solution is found or it reaches the maximum number of generations.

However, the Genetic Algorithm suffers from low performance when implemented in software on CPUs. Parallel computing approach has been proposed to overcome the speed problem of Genetic Algorithm. Therefore, hardware accelerating approaches are introduced to help Genetic Algorithm to reach higher performance.

7.1.2 Hardware Accelerators Comparison

There have many proposed hardware accelerator implementation on GPUs [67] [68] [69] [70] [71] [72], FPGAs [73] [74] [75], and ASICs [76] [77] [78] [79] [80] [81] for the Genetic Algorithm. Some of them are general purpose Genetic Algorithm acceleratosr while some are problem specific. Hardware accelerators implementation can offer higher performance and energy efficiency compared to commodity processors when they are implemented correctly. However, programming these co-processing architectures, in particular FPGAs and ASICs, requires software developers to learn a whole new set of skills and hardware design concepts, and accelerated application development takes more time than producing a pure software version [82].

Compared to FPGAs and ASICs, GPUs have gained popularity as domain specific hardware accelerators because they are simpler to program using CUDA, an API that enables GPUs to be programmed with high level programming language such as C, C++, and Fortran [83]. Recently GPUs have expanded their territory from graphics processing to general purpose high performance computing, known as General-Purpose GPU (GPGPU). Using their large parallel computing ca-

pabilities they can be used to do other types of analysis competing with modern general-purpose multi-core CPUs [84]. While GPUs are powerful due to their large parallel computing capabilites, they might not be the best option to deploy on mobile devices due to their high energy consumption.

To maximize the power efficiency, researchers often turn their attention to ASICs. However, ASIC hardware accelerators cannot be reprogrammed once they are fabricated. Their lack of flexibility makes them unsuitable for algorithms they require to change constantly. Moreover, the cost of building new ASIC chips with state-of-the-art fabrication technologies is becoming increasingly high, especially for relatively moderate volumes [85]. Hence, programmable platforms such as FPGAs are more attractive compared to ASICs.

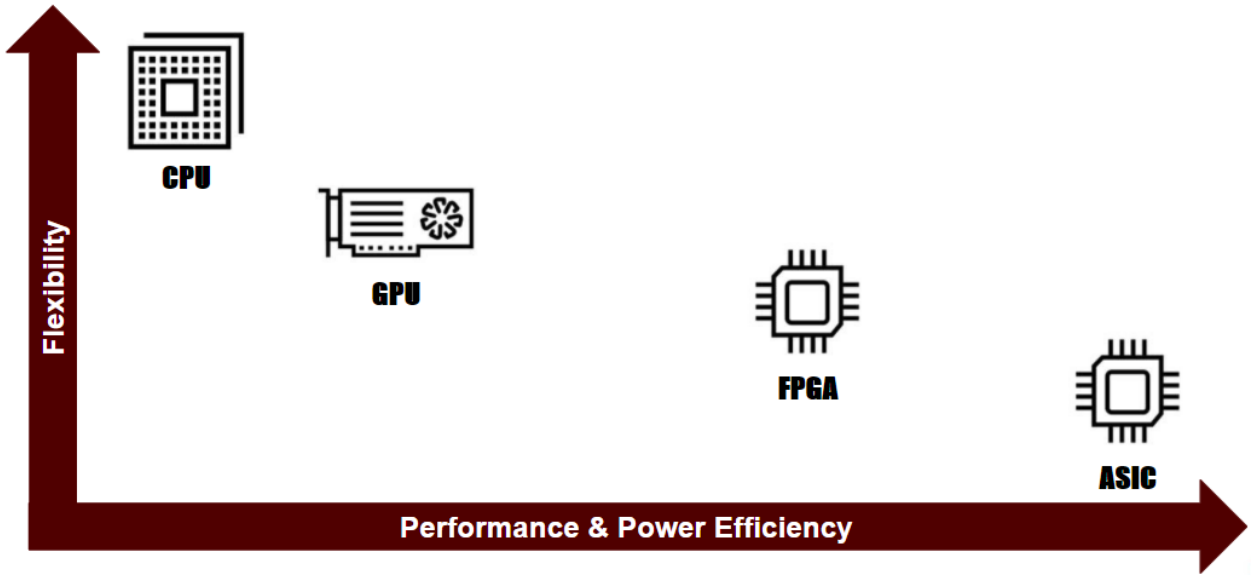


Figure 7.2: Power and performance trade-off for different hardware platforms

Due to the reconfigurability of FPGAs, they have become an alternative to ASICs for high performance computing accelerators. Field Programmable Gate Array is a semiconductor device composed of many logic blocks with configurable interconnections between these [86]. FPGAs operate with a lower clock frequency, have lower peak performances but can be hardware op-

timized for each specific application and, therefore FPGAs can achieve better performance efficiencies. Another important point is that FPGAs in general achieve higher power efficiencies compared to GPUs and CPUs [84]. Applications successfully implemented on FPGAs have typically contained high levels of parallelism and have often used simple statically-scheduled control and modest arithmetic. Successful computing applications on FPGA technology have applied the techniques including deep pipelined processing structures, extensive data-parallelism, simple data objects, and modest arithmetic [87].

There is no absolute answer about which hardware accelerator platform is the best since the answer depends on the application itself. Some parallel intensive applications should be implemented in a specific platform while others could be equally implemented in several platforms [84]. Figure 7.2 illustrates the trade-off comparison for CPUs, GPUs, FPGAs, and ASICs. Each platform has its pros and cons, thus it is up to the designer to determine which platform suits the application requirements.

7.2 The Evolutionary Heuristic A* Algorithm

In this section, we would like to provide the basic concepts and implementation detail of EHA* before we present the microarchitecture of the FPGA accelerated EHA*.

7.2.1 Genome Encoding

EHA* has a self-evolving heuristic function that improves itself by learning from the given environment. The heuristic function is called the weighted-multi-heuristics function. A weighted-multi-heuristics function is simply a sum of products of W and H. It combines the guiding power of multiple heuristic functions to maximize its pathfinding performance. To evolve the heuristic function, it must be converted into a numerical representation called genome, as Figure 7.3 shows. A genome in EHA* contains 6 genes, and these genes are either a weight value or a heuristic index. A weight value is ranged from 0.1 to 1. Heuristic indexes represent simple heuristics that are chosen by the user and stored in a gene pool. A pair of weight and heuristic indexes is called a chromosome. During a reproduction phase in Genetic Algorithm, either a gene or chromosome is

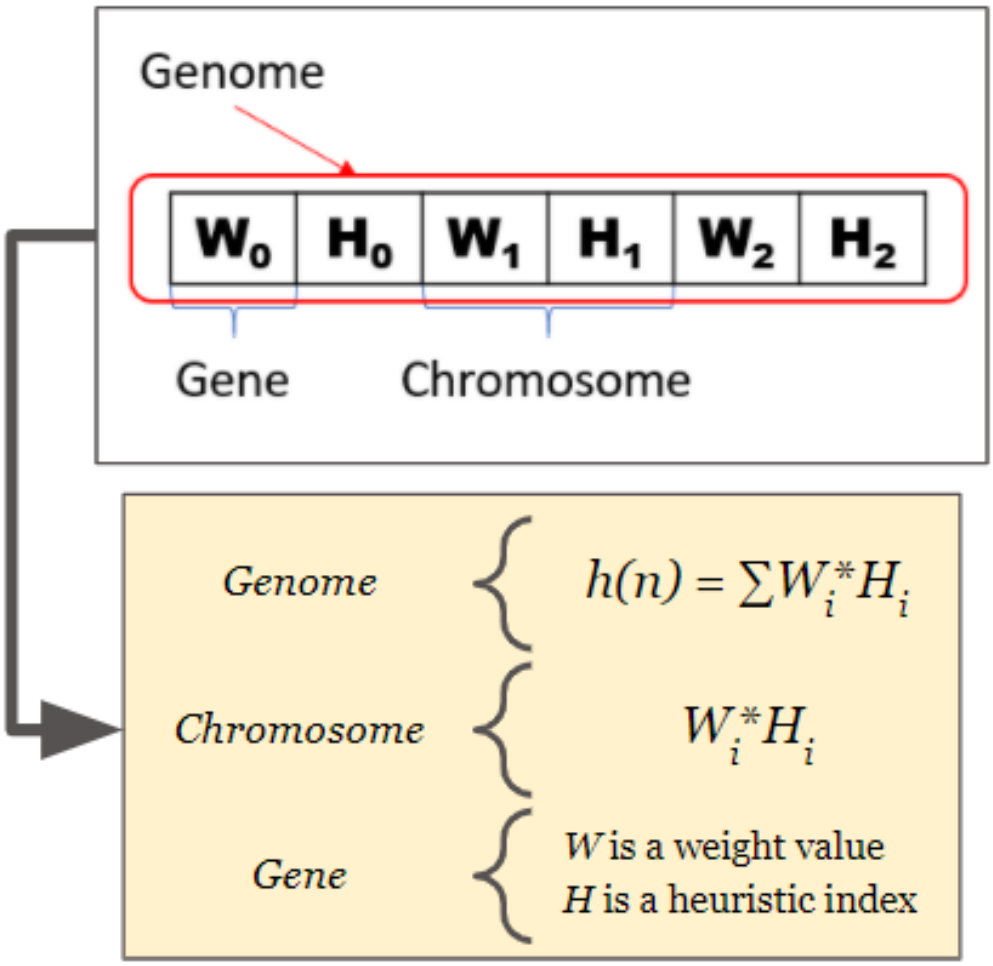


Figure 7.3: The binary representation of a multi-weighted-heuristics function.

updated as the result of evolution.

7.2.2 Offspring Generation

After each generation of genomes is reproduced and evaluated by the fitness function, the genomes need to compete with each other to survive. An individual with the best fitness score is considered as an elite and its genome will be passed to the next generation. The rest of the population needs to go through tournament selections and the winners will be chosen as parents to reproduce offsprings.

Tournament selection is one of the most popular selection methods in the Genetic Algorithm due to its efficiency and simple implementation. In tournament selection, n individuals are ran-

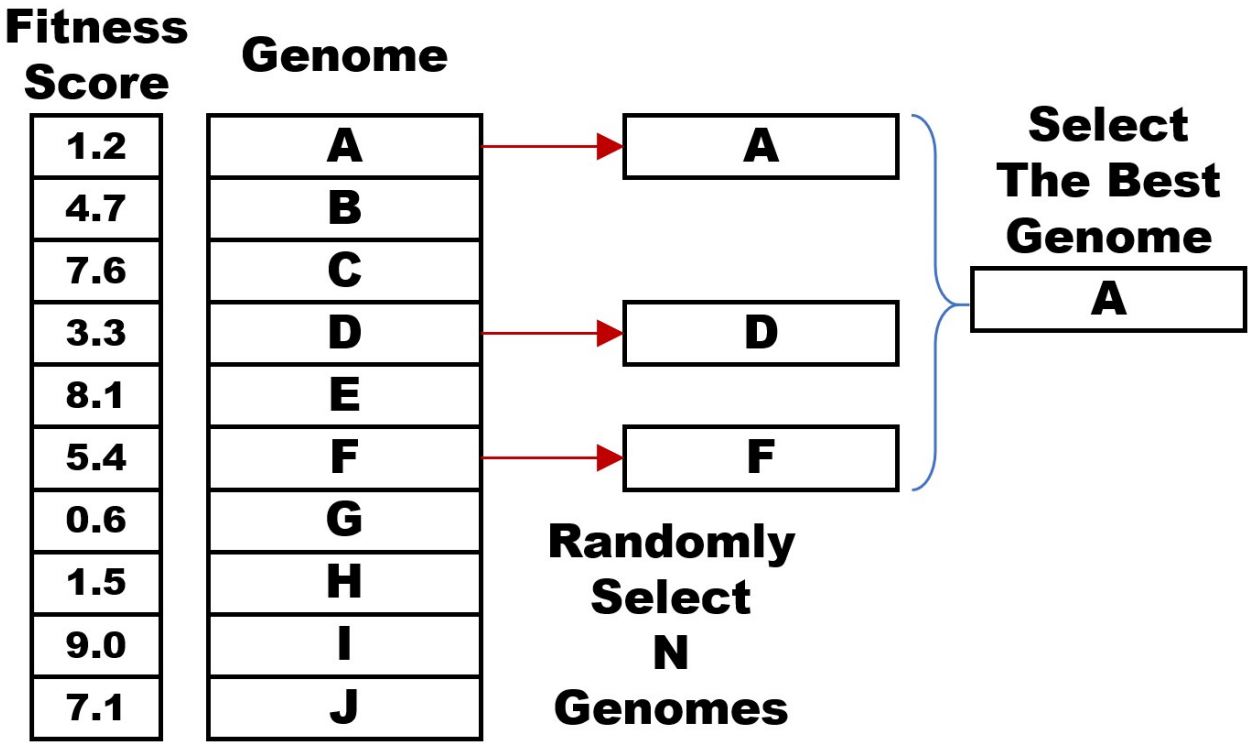


Figure 7.4: Demonstration of the tournament selection process.

domly selected from the population to compete against each other. The individual with the highest fitness score wins and is chosen to be one of the parents for the next generation population. To preserve population diversity, tournament selection randomly selects the competitors to avoid the population being dominated by a single species, even though this may slow down the evolution process. Figure 7.4 illustrates the mechanism and the procedure for tournament selection. The advantages of the tournament selection include efficient time complexity, high degree of diversity, and last but not least, it does not require fitness sorting which makes it very fast to compute.

After parents are chosen, offspring will be generated via the processes of crossover and mutation. Figure 7.5 shows the basic concept of crossover in EHA*. A pair of random chromosomes is swapped between two parents and one of the results is randomly chosen to be the offspring. After swapping chromosomes, adjusting weight values is needed to fulfill the constraint of the weight-multi-heuristic function, which is the sum of weight values of a genome must equal to one.

After crossover happened, there is a small probability for the offspring to mutate. As Figure

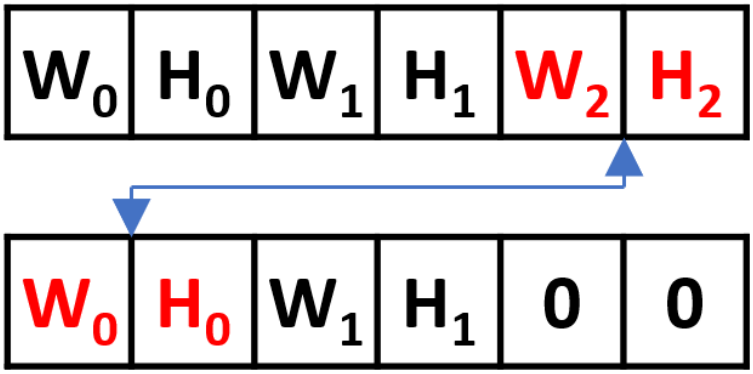
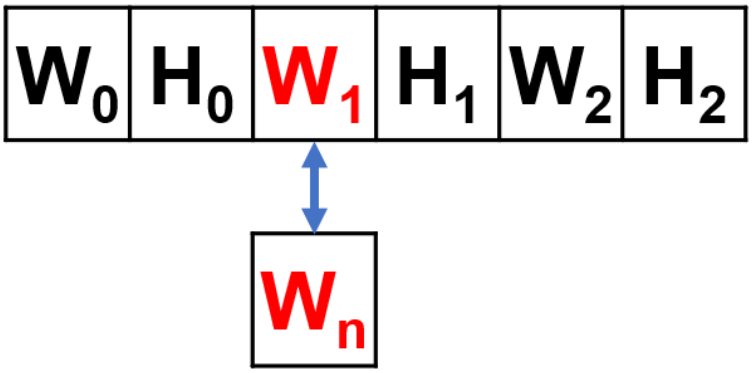
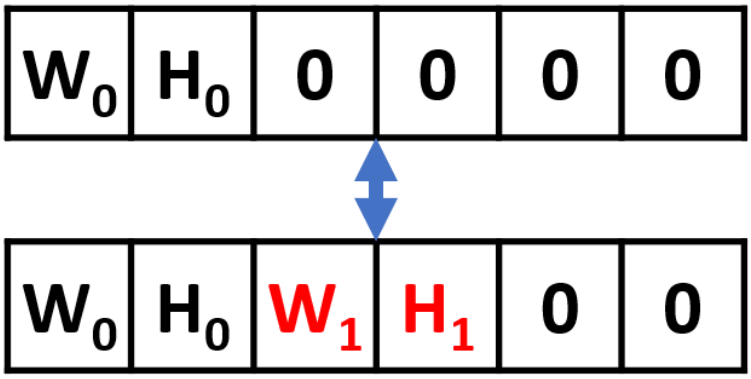


Figure 7.5: Demonstration of the crossover process between two parent genomes.



(a) Demonstration of the mutation process of a single gene mutation.



(b) Adding or deleting a chromosome from a genome.

Figure 7.6: Three types of mutation in EHA*.

7.6 shows, there are three types of mutation, which are gene mutation, additional chromosome, and chromosome deletion. The purpose of mutation is to provide more variation for a solution to avoid EHA* trapping in local minima when searching for an optimal heuristic function.

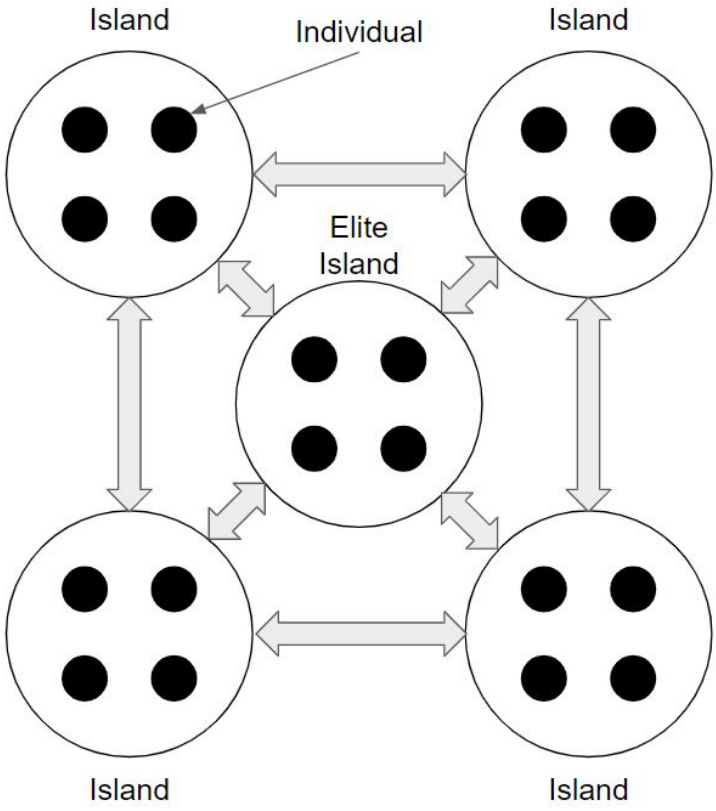


Figure 7.7: Genetic Algorithm island model implementation.

EHA* evolves heuristic functions by putting a population of genomes to compete with each other so that the fittest will survive at the end. However, offsprings with useful topological features can be prematurely terminated simply because the new feature has yet to be optimized thus leading to lower fitness score. EHA* is inspired by the concept of speciation and proposed an island model as Figure 7.7 shows to protect premature offsprings. The island model divides individuals to subgroups. Each subgroup only competes and reproduces within their own island until the inter-island iterations come up. Therefore, EHA* ensuring that the newborns are facing less competition

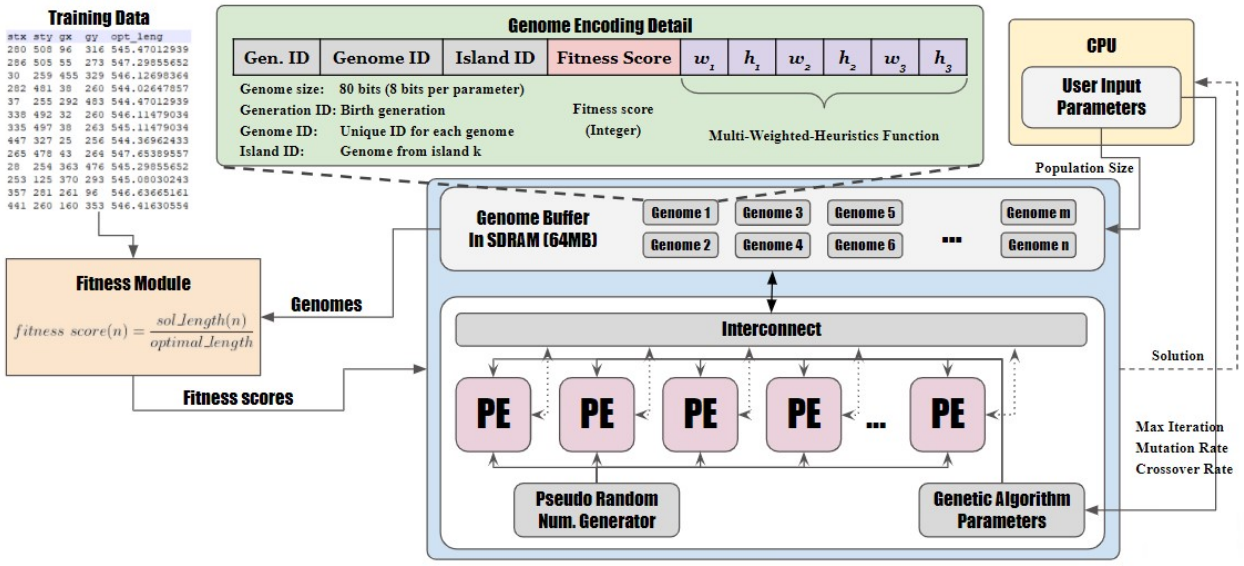


Figure 7.8: microarchitecture overview of the FEHA*.

and given enough time to optimize.

7.3 System Overview

In this section, we are presenting the proposed architecture to accelerate the Genetic Algorithm for EHA* to evolve heuristic function. Figure 7.8 presents the microarchitecture overview of our design. There are the CPU that simply provides an interface for user to enter the parameters for the accelerator, the evolution module that evolves genomes in parallel, the fitness module that evaluate genomes, the pseudo random number generator for the evolution process, and the genome buffer that stores the population for each generation. Moreover, Figure 7.8 illustrates the detail of the genome ecoding in hardware. The numerical representation of a genome contains information including a generation ID to record the birth generation of a genome, a genome ID that is unique to each genome in each generation as an identification for selection process, an island ID to show which island the genome belongs, a fitness score that belongs to the genome, and the heuristic function parameters.

7.3.1 Microarchitecture

After user provided the input parameters, FEHA* randomly generates an initial population and stores them in the genome buffer. Once a population is created, they will be fed to the fitness module to evaluate the quality of each genome in parallel. Equation 1 is used to evaluate each genome. The final fitness score of each genome is the average score of 10 computations. The fitness score will be updated to the genome buffer and wait for the processing elements (PE) to evaluate scores and start the reproduction process.

$$fitness\ score(n) = \frac{sol\_length(n)}{optimal\_length} \tag{7.1}$$

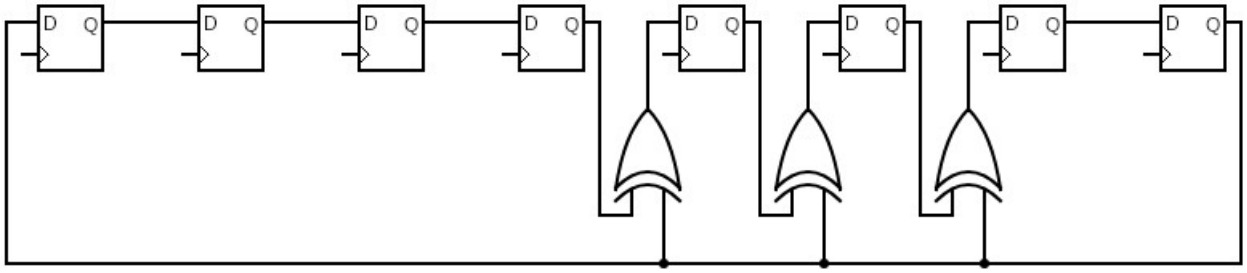


Figure 7.9: Logic circuit diagram of the Galois linear-feedback shift register.

The reproduction process of FEHA* that is executed in the processing elements (PEs) relies on a certain degree of randomness, but it does not necessarily require a true number generator. In our design, we chose to use a simple pseudo number number generator named the Galois linear feedback shift register (LFSR) [88]. The Galois LFSR is a simple logic circuit that can be used to generate pseudo random numbers simply by providing a seed value. The schematic of the Galois LFSR is presented in Figure 7.9. It is a bit shifter with extra XOR gates to generate a pseudo random sequence. In our implementation, the Galois LFSR is used to:

- generate an initial population at the beginning of the Genetic Algorithm.

- choose the parents for reproduction in our selection modules.
- choose the chromosome for swapping in our crossover modules.
- choose the gene or chromosome for mutation in our mutation modules.

7.3.2 Processing Elements

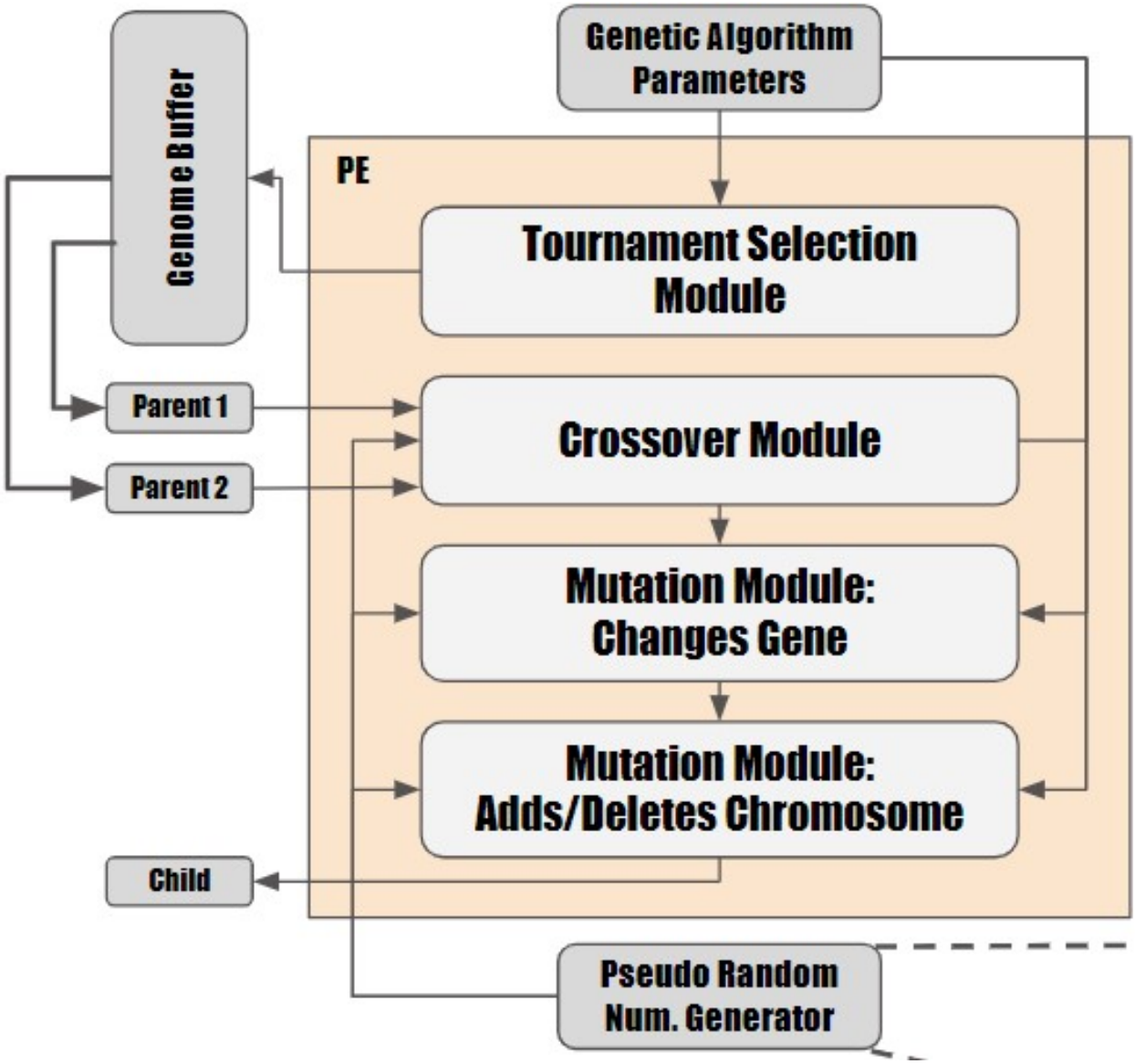


Figure 7.10: Processing element (PE) overview.

The processing elements (PEs) are the core component of the FEHA* evolution engine. A PE controls the selection module, crossover module, and three mutation modules. The three mutation modules are the gene mutation module, the chromosome addition module, and the chromosome deletion module. The modules in PEs are designed for pipelining for maximum throughput. Figure 7.10 shows the overview and dataflow of a PE. A PE starts with the selection module. Two selection modules, each randomly selects N genomes to enter the tournament. The genome with the highest fitness score among the selected genomes will be picked as parents to reproduce the new generation in the crossover module.

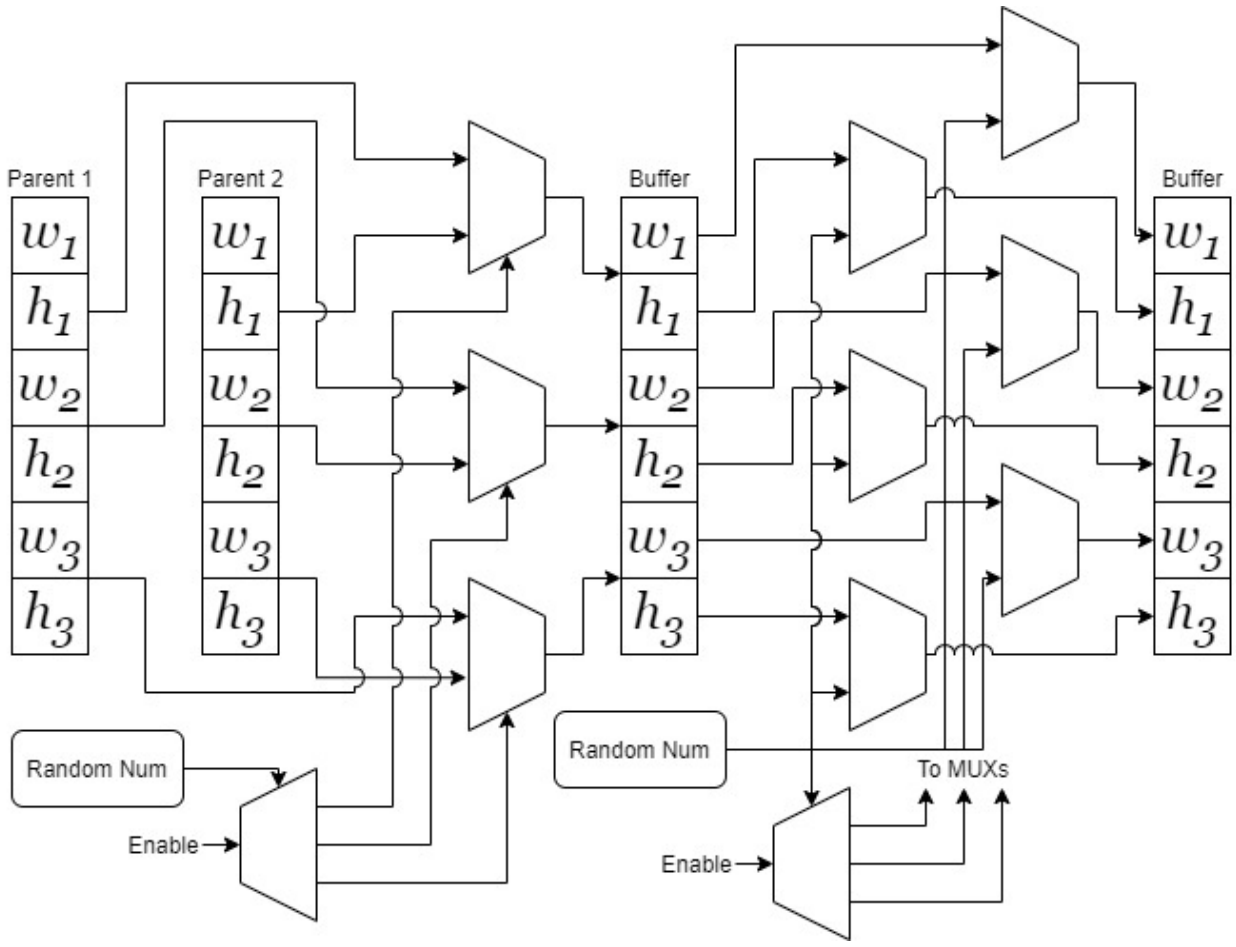


Figure 7.11: Logic circuit diagram of crossover and mutation module.

Crossover module receives two parents chosen by the tournament selection module and generates an offspring by randomly selecting a chromosome from parent 2 to randomly replace a chromosome from parent 1. The left hand side of Figure 7.11 shows the schematic of the crossover module. This process is simply completed by multiple multiplexers (MUXs). The output is stored in a register for the next pipeline process, which is the gene mutation module.

Unlike crossover, which has a high probability to happen, mutation modules have a much lower occurrence rate. The mutation rate starts relatively high and decreases over time as the organism becomes more complex. Just like in real life, mutation doesn't happen often and it happens slowly. Only one gene is changed for each occurence. The right hand side of Figure 7.11 shows how a gene is being updated. A gene is randomly chosen and mutated with a new value. Note that when a weight value is updated, the mutation module readjust the weight ratio so that the sum of weights equals to one.

After the gene mutation. The offspring goes into the next module, which is actually two separate processes named the chromosome addition module and the chromosome deletion module. As the name stated, these two modules serves the opposite purpose: adding a new chromosome and deleting an existing chromosome. Same as gene mutation, chromosome addition and deletion rarely happen. The addition module first checks and sees if there is any room to add a new chromosome. If so, then a new random chromosome is added to the offspring. The deletion module has a similar process: it checks if the offspring has more than one pair of chromosome (since the minimal number of chromosome is one), and then delete a pair of chromosome randomly. After these processes are completed, a newborn is generated and sent back to the genome buffer to replace the old genome that has the name genome ID number.

7.4 Experiment Design and Results

We start this section by illustrating the hardware platform we use for the experiment. Then we introduce our experiment design, including the benchmark we chose and the evaluation metrics. This section ends with detail evaluation and analysis of FEHA*, showing the strengths and advantages of FEHA*.

7.4.1 Experiment Setup

Genetic Algorithm Parameters Setting	
Max iteration:	50
Population size:	4, 8, 12, 16, and 20
Crossover rate:	0.9
Mutation rate:	Starts from 0.5 and decreases 0.1

Table 7.1: Parameters setting for EHA* and FEHA*

In our experiment, the optimized multi-threaded EHA* was implemented by python 3.6.9. The software was running on a computer with Windows Subsystem for Ubuntu. The computer has an Intel i7-4790k operating at 4.0 GHz, with 32.0 GB DDR3 RAM. For the hardware version, FEHA* is implemented on a Terasic DE1-SoC Development Board, which uses the Altera SoC FPGA Cyclone V. Table 7.1 lists the parameters setting including the maximum number of iterations, population size, crossover rate, and mutation rate for both EHA* and FEHA*.

7.4.2 Benchmark

We evaluate the performance of FEHA* using a widely used grid-based pathfinding benchmark introduced by Nathan Sturtevant's Moving AI Lab [31]. The experiment setup remains the same from chapter 3 to 6. The blackened tiles are the walls or obstacles that are untraversable and the white tiles are traversable. In our experiment, we have converted the representations of all traversable tiles to 0 and untraversable tiles to -1. The largest maps in the benchmark can have up to hundred thousands of walkable terrains.

We chose 10 maps that have a size of 512x512 tiles from different categories including open spaces, narrow hallways, and maze-like maps. Each map we run 100 searches with different starting nodes and goal nodes that are provided from the benchmark problem set. The benchmark assumes that diagonal moves are only possible if the related cardinal moves are both possible. That is, it is not possible to take a diagonal move between two obstacles, and it is not possible

to take a diagonal move to cut a corner. Such an assumption is made because in a real game all creatures occupy some volume of space and are not able to pass through blocked corners. Moreover, octile neighborhood relation, meaning the adjacency relationship in 4 straight and 4 diagonal directions is used in all the maps. A search agent that is allowed to move to one of its up to eight neighbors at any given time during the search as long as it is traversable.

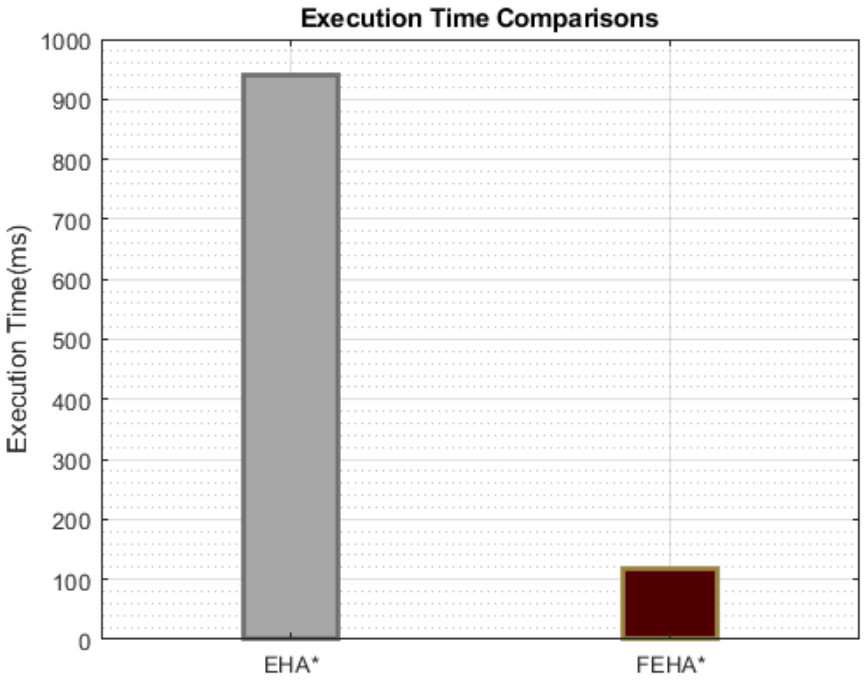
7.4.3 Results Comparisons and Discussion

To evaluate our FPGA accelerator, we focus on the execution time improvement under the benefit of parallelism, the latency to observe the speed up per process, and the logic element consumption to understand the scalability potential.

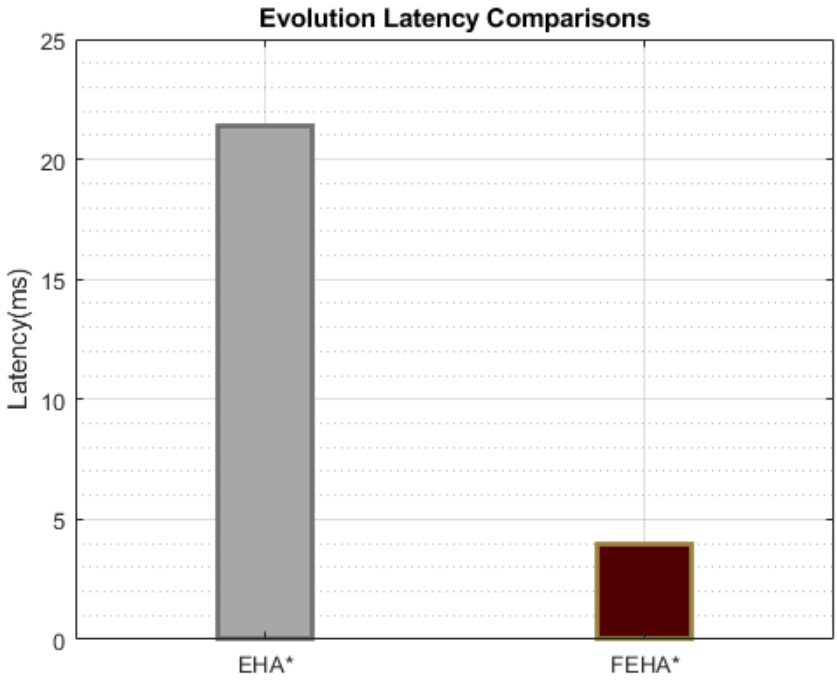
Without any surprise, FEHA* performs significantly better than the software version. Figure 7.12a shows the execution time comparison of the evolution process between the software and the hardware version of EHA*. Software EHA* is executed on a 4-core machine with 8-threads, while FEHA* has 20 PEs. The high degree of parallelism and deep pipeline process provide FEHA* to achieve much higher throughput compared to software EHA*. Although we have not evaluated FEHA* with a much higher number of population with more iterations to test the limitation of the accelerator, but we still achieve 8.3x speed up compared to the software implementation of EHA*.

Even without the benefit of pipelining and parallelism, simple operations of the process element help to speed up the evolution process. We observe 5.5x speed up in terms of latency shown in Figure 13b. We believe the latency can be further reduced by integrating all mutation modules into one. However, such design may compromise the throughput due to a longer clock cycle in one module.

Another important data to observe for a hardware accelerator is the area consumption of the development board. We observe the results in Table 7.2 shows almost linear growth of logic elements consumption as the number of PEs increases. The clock frequency states around 55MHz as the number of PEs increases. In conclusion, the linear growth and the stable clock frequency shows that it is possible to increase more PEs without any negative effects.



(a) Evolution process execution time comparison.



(b) Latency of generating one offspring comparison.

Figure 7.12: Performance analysis of EHA* and FEHA*.

Number of PEs	Clock (MHz)	Logic Elements (%)
4	56.26	3%
8	55.61	5%
12	55.61	9%
16	55.34	15%
20	54.87	23%

Table 7.2: Hardware utility analysis.

7.5 Summary and Future Works

In this chapter, we presented a FPGA accelerator architecture for the Evolutionary Heuristic A* (FEHA*) that exploits parallelism and pipelining to improve the computation speed. The accelerator includes initial population generators, pseudo random number generators, fitness evaluation modules, tournament selection modules, and reproduction operators. The performance of FEHA* is significantly better in terms of execution time and latency than the software implementation even with a low degree of parallelism. Our results show that FEHA* is capable to achieve a much higher level of parallelism due to the fact that only a small amount of logic elements is used in our experiments.

To unlock the full potential of this work, a further development and evaluation is needed. First of all, the evolution engine, a.k.a the processing element (PE) can be modified to become a general purpose Genetic Algorithm engine. To achieve this goal, other types of selection, crossover, and mutation methods are needed to be included in the modules. This includes but not limited to proportional roulette wheel selection, rank-based roulette wheel selection, single point crossover, two point crossover, swap mutation, inversion mutation, and scramble mutation. Area reduction and data path optimization is needed to balance the computing performance and area consumption. Moreover, the evolution engine should be evaluated with different types of optimization problems, especially problems with higher dimensions. To test the limitation of the FPGA accelerator, increase the population size and the maximum number of iterations are needed as well for a full scale evaluation.

8. CONCLUSION

In this dissertation, we study pathfinding problems in grid-based two-dimensional environments. We address four significant challenges that are commonly seen in pathfinding problems and pathfinding algorithms including (i) heuristic function design difficulty; (ii) large search space; (iii) high search problem complexity; and (iv) limited computation resources such as memory and preprocessing time. In this dissertation, we propose that self-evolving heuristic function for A* is needed to improve and optimize search performance for complex pathfinding problems. Our objectives throughout this dissertation are to:

1. Find an optimal solution in the minimum amount of time and resources
2. Reduce the engineering effort of designing heuristic functions for A* algorithm.
3. Reduce pathfinding problem complexity.

To overcome the said challenges, in Chapter 3, we address the first challenge and proposed EHA*. In Chapter 4, we illustrate the second challenge and proposed RDA. In Chapter 5 and 6 we present the first three challenges and proposed HEHA*, which combines and improves the strengths of EHA* and RDA. In Chapter 7, we address the last challenge and propose a FPGA accelerator for EHA* to reduce resource consumption.

8.1 Dissertation Summary

In this dissertation, we first present a self-evolving heuristic search algorithm named Evolutionary Heuristic A* which has the capability to design and optimize a Multi-Weighted-Heuristic function. EHA* overcomes the difficulty to design high complexity heuristic functions. We have evaluated EHA* on a widely used two-dimensional grid-based pathfinding benchmark. The experimental results show that EHA* has the ability to optimize a complex heuristic function within a reasonable amount of time. EHA* is tested against previous works in different benchmarks to

prove that EHA* balances the solution optimality, the memory space occupation, and the optimization time. Our experiment results show that EHA* is 35% faster and consume 39% less memory compared to A* with the Euclidean distance as the heuristic function.

We then propose a novel clustering algorithm named Regions Discovery Algorithm (RDA) for grid-based maps to speed up pathfinding and reduce search effort by removing dead ends from the search space. RDA clusters and identifies regions based on their local features, which leads to better graph reduction results. RDA uses lookup tables to reduce search space instead of precomputed paths to 1) minimize time and memory consumption in preprocessing phase, and 2) guarantee solution optimality. Moreover, our approach is environment independent, simple, and effective. Our work can be combined with other pathfinding algorithms for further performance boost. Our results show that RDA has little overhead in preprocessing, while having significant performance gain in the pathfinding process to search for optimal solutions. On average RDA has reduced 30% search space in our experiments. In some cases RDA could reduce up to 47% of search space.

To further extend the idea of search space reduction and self-evolving heuristic function, we propose a hierarchical approach named Hierarchical Evolutionary Heuristic A* (HEHA*) that combines the guiding power of EHA* with RDA. HEHA* clusters search space and evolves heuristic functions based on local features of regions. Our experimental results show that HEHA* proves the needs of 1) multiple heuristics in pathfinding and 2) evolutionary heuristic function. Compared to the A* variants that we chose in our experiment, HEHA* showed better overall search performance. HEHA* balances the solution optimality, the memory space occupation, and the optimization time. Compared to previous works, HEHA* reduces 50% additional space compared to RDA and achieves 5.39x speed up compared to EHA*

We also propose a multi-agent version of HEHA* (MA-HEHA*) that searches for solutions with minimal makespan by providing accurate priority decision mechanism and evolving heuristic functions to avoid collisions. MA-HEHA* improves search performance and scalability to solve MAPF problems. We evaluated MA-HEHA* with multiple common MAPF problems on different types of maps to show the effectiveness of the algorithm. The contributions of MA-HEHA* are

threefold: 1) MA-HEHA* decomposes a grid-based map in many small regions and identifies high traffic areas, 2) MA-HEHA* evolves heuristic functions based on local features to increase search performance and avoid potential collisions, and 3) MA-HEHA* minimizes the needs for path restructuring with our hierarchical approach. Our experimental results show that MA-HEHA* reduces the amount of re-computation and improves scalability to handle more agents in MAPF problems which leads to over 90% success rate up to 70 agents. Moreover, MA-HEHA* shows that it is distributed-system-ready since no centralized controller is needed for MA-HEHA* to operate.

8.2 Future Work

With pathfinding problems extending far beyond the two dimensional world, demands for better and more efficient problem complexity reduction techniques will only increase over time. The need for new pathfinding algorithm enhancement techniques with high scalability is essential to manage the constantly growing pathfinding problems. To fulfill these demands and needs, we discuss future work in terms of our two main contributions: Hierarchical pathfinding approach and FPGA accelerator for evolution engine.

8.3 Hierarchical Approach For Multidimensional Pathfinding Problems

Multidimensional pathfinding problems become more popular as robotic systems become more mainstream in the industry. The definition of multidimensional in terms of robotics refers to the degree of freedom of a robot. For instance, a robot arm with 10 joints is considered a 10-dimensional problem. As the number of dimensions increases, applying RDA to cluster the search space becomes more difficult. A new clustering approach is needed to achieve hierarchical pathfinding for multi-dimensional problems. Machine learning clustering algorithms such as Gaussian Mixture Models (GMMs) [89] and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [90] are suitable for discovering the most traveled states in the high dimensional space. By identifying these states and using them as abstracted nodes, a dimension reduced abstract map can be constructed, and the general idea of HEHA* can now be applied to high dimensional pathfinding problems.

8.3.1 Heterogeneous Computing With General Purpose Evolution Engine

In chapter 7 we have presented the idea of a FPGA-based Genetic Algorithm accelerator for EHA*. This is an application-specific accelerator that can be modified to become a truly general purpose Genetic Algorithm accelerator. For the processing element, we have chosen to use a specific type of selection module, crossover module, and mutation module. There are more selection methods including proportional roulette wheel selection and rank-based roulette wheel selection. There are more than one type of crossover methods, for instance, single point crossover and two-point crossover. For mutation, there are swap mutation, inversion mutation, and scramble mutation. With careful logic design, area increment can be trivial while providing more varieties for the user.

To completely generalize the evolution engine, the fitness evaluation module must be separated from the engine. Therefore, heterogeneous architecture could be an optimal solution. The fitness evaluation module can be implemented in softwares to reduce the hardware knowledge requirement for users. The communication between the CPU and the FPGA accelerator would be another challenge that needs to be addressed in order to achieve maximum throughput. By adding more selections, crossovers, and mutation methods to process elements, and moving the fitness module away from the hardware, the genetic algorithm accelerator will become a general purpose accelerator that is capable of accelerating different types of applications based on the user's needs.

REFERENCES

[1] 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)*, pp. 25–32, IEEE, 2018.

[2] Y. F. Yiu, J. Du, and R. Mahapatra, "Evolutionary heuristic a* search: Pathfinding algorithm with self-designed and optimized heuristic function," *International Journal of Semantic Computing*, vol. 13, no. 01, pp. 5–23, 2019.

[3] Y. F. Yiu and R. Mahapatra, "Regions discovery algorithm for pathfinding in grid based maps," 2020.

[4] Y. F. Yiu and R. Mahapatra, "Hierarchical evolutionary heuristic a* search," 2020.

[5] Y. F. Yiu and R. Mahapatra, "Multi-agent pathfinding with hierarchical evolutionary hueristic a*," 2020.

[6] Y. F. Yiu and R. Mahapatra, "Heuristic function evolution for pathfinding algorithm in fpga accelerator," 2020.

[7] E. Erdem, D. G. Kisa, U. Oztok, and P. Schüller, "A general formal framework for pathfinding problems with multiple agents," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[8] Z. Abd Algfoor, M. S. Sunar, and H. Kolivand, "A comprehensive study on pathfinding techniques for robotics and video games," *International Journal of Computer Games Technology*, vol. 2015, 2015.

[9] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[10] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of artificial intelligence tools*, pp. 13–13, Springer, 1984.

[11] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[13] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970.

[14] M. Likhachev, G. J. Gordon, and S. Thrun, "Ara*: Anytime a* with provable bounds on sub-optimality," in *Advances in neural information processing systems*, pp. 767–774, 2004.

[15] C. M. Wilt and W. Ruml, "When does weighted a* fail?," in *SOCS*, pp. 137–144, 2012.

[16] K. Grant and D. Mould, "Lpi: Approximating shortest paths using landmarks," in *Workshop on Artificial Intelligence in Games*, p. 45, 2008.

[17] N. Sturtevant and M. Buro, "Partial pathfinding using map abstraction and refinement," in *AAAI*, vol. 5, pp. 1392–1397, 2005.

[18] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of game development*, vol. 1, no. 1, pp. 7–28, 2004.

[19] L. M. Duc, A. S. Sidhu, and N. S. Chaudhari, "Hierarchical pathfinding and ai-based learning approach in strategy game design," *International Journal of Computer Games Technology*, vol. 2008, 2008.

[20] N. Pelechano and C. Fuentes, "Hierarchical path-finding for navigation meshes (hna$*$)," *Computers & Graphics*, vol. 59, pp. 68–78, 2016.

[21] V. Rahmani and N. Pelechano, "Improvements to hierarchical pathfinding for navigation meshes," in *Proceedings of the Tenth International Conference on Motion in Games*, pp. 1–6, 2017.

[22] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a," *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, 2016.

[23] F. Islam, O. Salzman, and M. Likhachev, "Online, interactive user guidance for high-dimensional, constrained motion planning," *arXiv preprint arXiv:1710.03873*, 2017.

[24] V. Narayanan, S. Aine, and M. Likhachev, "Improved multi-heuristic a* for searching with uncalibrated heuristics," in *Eighth Annual Symposium on Combinatorial Search*, 2015.

[25] F. Islam, V. Narayanan, and M. Likhachev, "Dynamic multi-heuristic a," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2376–2382, IEEE, 2015.

[26] H.-C. Chen and J.-D. Wei, "Using neural networks for evaluation in heuristic search algorithm," in *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[27] H. Darko, "Combining heuristics with neural networks," 2019.

[28] A. Keselman, S. Ten, A. Ghazali, and M. Jubeh, "Reinforcement learning with a* and a deep heuristic," *arXiv preprint arXiv:1811.07745*, 2018.

[29] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[30] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[31] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012.

[32] A. Shkolnik and R. Tedrake, "Path planning in 1000+ dimensions using a task-space voronoi bias," in *2009 IEEE International Conference on Robotics and Automation*, pp. 2061–2067, IEEE, 2009.

[33] W. Lee and R. Lawrence, "Fast grid-based path finding for video games," in *Canadian Conference on Artificial Intelligence*, pp. 100–111, Springer, 2013.

[34] Z. A. Algfoor, M. S. Sunar, and H. Kolivand, "A comprehensive study on pathfinding techniques for robotics and video games," *International Journal of Computer Games Technology*, vol. 2015, 2015.

[35] E. D. de Jong, D. Thierens, and R. A. Watson, "Hierarchical genetic algorithms," in *International Conference on Parallel Problem Solving from Nature*, pp. 232–241, Springer, 2004.

[36] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald, "Hierarchical a*: Searching abstraction hierarchies efficiently," in *AAAI/IAAI, Vol. 1*, pp. 530–535, Citeseer, 1996.

[37] C. Astengo-Noguez and J. R. C. Gómez, "Collective pathfinding in dynamic environments," in *Mexican International Conference on Artificial Intelligence*, pp. 859–868, Springer, 2008.

[38] A. Bleiweiss, "Gpu accelerated pathfinding," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 65–74, Eurographics Association, 2008.

[39] G. R. Jagadeesh, T. Srikanthan, and C. Lim, "Field programmable gate array-based acceleration of shortest-path computation," *IET computers & digital techniques*, vol. 5, no. 4, pp. 231–237, 2011.

[40] K. Anderson, "Additive heuristic for four-connected gridworlds," in *Third Annual Symposium on Combinatorial Search*, 2010.

[41] R. C. Holte, M. Perez, R. Zimmer, and A. MacDonald, "Hierarchical a*," in *Symposium on Abstraction, Reformulation, and Approximation*, 1995.

[42] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pp. 569–577, 2002.

[43] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. S. Kumar, *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[44] O. Salzman and R. Stern, "Research challenges and opportunities in multi-agent path finding and multi-agent pickup and delivery problems," in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 1711–1715, 2020.

[45] F. Grenouilleau, W.-J. van Hoeve, and J. N. Hooker, "A multi-label a* algorithm for multi-agent pathfinding," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 29, pp. 181–185, 2019.

[46] S. S. Chouhan and R. Niyogi, "Dmapp: A distributed multi-agent path planning algorithm," in *Australasian Joint Conference on Artificial Intelligence*, pp. 123–135, Springer, 2015.

[47] R. Nissim and R. I. Brafman, "Multi-agent a* for parallel and distributed systems," in *ICAPS Workshop on Heuristics and Search for Domain-Independent Planning*, pp. 43–51, 2012.

[48] C. M. Wilt and A. Botea, "Spatially distributed multiagent path planning," in *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

[49] V. Rahmani and N. Pelechano, "Multi-agent parallel hierarchical path finding in navigation meshes (ma-hna*)," *Computers & Graphics*, vol. 86, pp. 1–14, 2020.

[50] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. S. Kumar, and S. Koenig, "Adding heuristics to conflict-based search for multi-agent path finding," in *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[51] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved heuristics for multi-agent path finding with conflict-based search.," in *IJCAI*, pp. 442–449, 2019.

[52] 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 *Eighth annual symposium on combinatorial search*, Citeseer, 2015.

[53] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[54] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[55] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–130, 2011.

[56] F. Liu and G. Zeng, "Study of genetic algorithm with reinforcement learning to solve the tsp," *Expert Systems with Applications*, vol. 36, no. 3, pp. 6995–7001, 2009.

[57] G.-F. Nan, M.-Q. Li, and J.-S. Kou, "Two novel encoding strategies based genetic algorithms for circuit partitioning," in *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826)*, vol. 4, pp. 2182–2188, IEEE, 2004.

[58] J. Shapiro, "Genetic algorithms in machine learning," in *Advanced Course on Artificial Intelligence*, pp. 146–168, Springer, 1999.

[59] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[60] M. M. Drugan, "Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms," *Swarm and evolutionary computation*, vol. 44, pp. 228–246, 2019.

[61] H. Jun and Z. Qingbao, "Multi-objective mobile robot path planning based on improved genetic algorithm," in *2010 International Conference on Intelligent Computation Technology and Automation*, vol. 2, pp. 752–756, IEEE, 2010.

[62] J. K. Parker, A. R. Khoogar, and D. E. Goldberg, "Inverse kinematics of redundant robots using genetic algorithms," in *1989 IEEE International Conference on Robotics and Automation*, pp. 271–272, IEEE Computer Society, 1989.

[63] T. Technologies, "De1-soc development kit."

[64] M. Mitchell, "Genetic algorithms: An overview.," *Complexity*, vol. 1, no. 1, pp. 31–39, 1995.

[65] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *computer*, vol. 27, no. 6, pp. 17–26, 1994.

[66] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[67] J. Luo, S. Fujimura, D. El Baz, and B. Plazolles, "Gpu based parallel genetic algorithm for solving an energy efficient dynamic flexible flow shop scheduling problem," *Journal of Parallel and Distributed Computing*, vol. 133, pp. 244–257, 2019.

[68] J. R. Cheng and M. Gen, "Parallel genetic algorithms with gpu computing," in *Industry 4.0-Impact on Intelligent Logistics and Manufacturing*, IntechOpen, 2020.

[69] P. Pospichal and J. Jaros, "Gpu-based acceleration of the genetic algorithm," *GECCO competition*, 2009.

[70] J.-M. Li, X.-J. Wang, R.-S. He, and Z.-X. Chi, "An efficient fine-grained parallel genetic algorithm based on gpu-accelerated," in *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, pp. 855–862, IEEE, 2007.

[71] P. Pospichal, J. Jaros, and J. Schwarz, "Parallel genetic algorithm on the cuda architecture," in *European conference on the applications of evolutionary computation*, pp. 442–451, Springer, 2010.

[72] J. Jaros, "Multi-gpu island-based genetic algorithm for solving the knapsack problem," in *2012 IEEE Congress on Evolutionary Computation*, pp. 1–8, IEEE, 2012.

[73] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "High-speed fpga-based implementations of a genetic algorithm," in *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, pp. 9–16, IEEE, 2009.

[74] M. F. Torquato and M. A. Fernandes, "High-performance parallel implementation of genetic algorithm on fpga," *Circuits, Systems, and Signal Processing*, vol. 38, no. 9, pp. 4014–4039, 2019.

[75] N. Attarmoghaddam, K. F. Li, and A. Kanan, "Fpga implementation of crossover module of genetic algorithm," *Information*, vol. 10, no. 6, p. 184, 2019.

[76] S. Koizumi, S. Wakabayashi, T. Koide, K. Fujiwara, and N. Imura, "A risc processor for high-speed execution of genetic algorithms," in *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pp. 1338–1345, 2001.

[77] W. Gilreath and P. Laplante, "A one instruction set architecture for genetic algorithms," in *Biocomputing*, pp. 91–113, 2004.

[78] P. R. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica, "Customizable fpga ip core implementation of a general-purpose genetic algorithm engine," *IEEE Transactions on Evolutionary Computation*, vol. 14, no. 1, pp. 133–149, 2009.

[79] S. P. H. Alinodehi, S. Moshfe, M. S. Zaeimian, A. Khoei, and K. Hadidi, "High-speed general purpose genetic algorithm processor," *IEEE transactions on cybernetics*, vol. 46, no. 7, pp. 1551–1565, 2015.

[80] N. Kavvadias, V. Giannakopoulou, and S. Nikolaidis, "Development of a customized processor architecture for accelerating genetic algorithms," *Microprocessors and Microsystems*, vol. 31, no. 5, pp. 347–359, 2007.

[81] S. Areibi, M. Moussa, and G. Koonar, "A genetic algorithm hardware accelerator for vlsi circuit partitioning," *International Journal of Computers and Their Applications*, vol. 12, no. 3, p. 163, 2005.

[82] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *2010 International Conference on Field-Programmable Technology*, pp. 94–101, IEEE, 2010.

[83] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pp. 836–838, IEEE, 2008.

[84] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, IEEE, 2014.

[85] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, "A comparative study on asic, fpgas, gpus and general purpose processors in the o (nˆ 2) gravitational n-body simulation," in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 447–452, IEEE, 2009.

[86] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," *Internet: www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final*, 2013.

[87] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "Real-time optical flow calculations on fpga and gpu architectures: a comparison study," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pp. 173–182, IEEE, 2008.

[88] M. Goresky and A. M. Klapper, "Fibonacci and galois representations of feedback-with-carry shift registers," *IEEE Transactions on Information Theory*, vol. 48, no. 11, pp. 2826–2836, 2002.

[89] D. A. Reynolds, "Gaussian mixture models.," *Encyclopedia of biometrics*, vol. 741, 2009.

[90] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Kdd*, vol. 96, pp. 226–231, 1996.