International Journal of Semantic Computing
Vol. 13, No. 1 (2019) 5–23
©World Scientific Publishing Company
DOI: 10.1142/S1793351X19400014

Evolutionary Heuristic A* Search: Pathfinding Algorithm with Self-Designed and Optimized Heuristic Function

Ying Fung Yiu
Computer Science & Engineering Department
Texas A&M University
College Station, TX 77840, USA
yfyiu@tamu.edu

Jing Du
Department of Civil Engineering
University of Florida
Gainesville, FL 32603, USA
du.j@ufl.edu

Rabi Mahapatra
Computer Science & Engineering Department
Texas A&M University
College Station, TX 77840, USA
rabi@tamu.edu

The performance and efficiency of A* search algorithm heavily depends on the quality of the heuristic function. Therefore, designing an optimal heuristic function becomes the primary goal of developing a search algorithm for specific domains in artificial intelligence. However, it is difficult to design a well-constructed heuristic function without careful consideration and trial-and-error, especially for complex pathfinding problems. The complexity of a heuristic function increases and becomes unmanageable to design when an increasing number of parameters are involved. Existing approaches often avoid complex heuristic function design: they either trade-off the accuracy for faster computation or taking advantage of the parallelism for better performance. The objective of this paper is to reduce the difficulty of complex heuristic function design for A* search algorithm. We aim to design an algorithm that can be automatically optimized to achieve rapid search with high accuracy and low computational cost. In this paper, we present a novel design and optimization method for a Multi-Weighted-Heuristics function (MWH) named Evolutionary Heuristic A* search (EHA*) to: (1) minimize the effort on heuristic function design via Genetic Algorithm (GA), (2) optimize the performance of A* search and its variants including but not limited to WA* and MHA*, and (3) guarantee the completeness and optimality. EHA* algorithm enables high performance searches and significantly simplifies the processing of heuristic design. We apply EHA* to multiple grid-based pathfinding benchmarks to evaluate the performance. Our experiment result shows that EHA* (1) is capable of choosing an accurate heuristic function that provides an optimal solution, (2) can identify and eliminate inefficient heuristics, (3) is able to automatically design multi-heuristics function, and (4) minimizes both the time and space complexity.

Keywords: A* search; heuristic function; grid-based pathfinding; optimization; genetic algorithm.

1. Introduction

A* search algorithm [1] 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 is resource-hungry on high-dimensional search problems, 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 relies 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.

A well-designed heuristic function is essential for the A* search to obtain solutions 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 resolves the heuristic function design problem. Aine *et al.* [2] 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 [3] 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.

We propose 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 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 prove the optimality of MWH, the completeness 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 Sturtevant [4]. 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 include 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 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 of designing complex Multi-Weighted-Heuristic (MWH) functions with great performance.
(3) EHA* guarantees completeness and optimality.
(4) Optimal solution paths can be found by EHA* with minimal computational cost.

The body of this paper includes related works and background on heuristic search algorithm optimization in Sec. 2. Section 3 discusses the methodology, with a detailed explanation of the algorithm, the optimization models, and the benchmarks that we used for the experiment. Section 4 explains our evaluation method, performance metrics, and the results of the experiment. The results include the accuracy of the heuristic function, the execution time of the heuristic optimization, and the memory usage of the search algorithm. Section 5 concludes the paper and discusses potential future works.

2. Background

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.

The accuracy–speed trade-off is the primary focus for researchers, namely, to provide an optimal solution with respect to computation time and space complexity. For a complex planning problem, it may not be possible to compute optimal solutions within a bounded time. Pohl [3] proposed WA* that inflates the heuristic function by the weight w, which transforms the equation to $f(n) = g(n) + w * h(n)$. The higher inflation of $h(n)$ leads to less priority queue operations and faster search. However, it does not guarantee completeness and optimality. Figure 1 shows the impact of weight to the search behaviors and solutions quality of WA*. While the weight

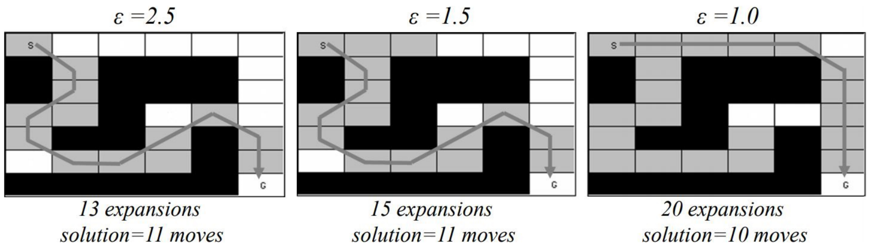


Fig. 1. The impact of weight w to WA* search performance. Higher weight value leads to faster computation but a less accurate result.

decreases 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.

To address the shortcomings of WA*, Likhachev [5] 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.

While WA* shows its practical use on motion planning and robotics, Wilt *et al.* [6] showed that there are several occasions where WA* and its variants failed to accomplish their goal. WA* 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 [6]. Therefore, researchers choose different approaches when WA* fails to deliver.

There are other complex problems that require A* search to be complete and optimal, therefore researchers usually exploit the multi-agent system to maximize search performance. Aine *et al.* [2] developed multi-heuristic A* (MHA*) that simultaneously uses multiple heuristic functions to search, which guarantees completeness and bounds on suboptimality. MHA* has two versions: Independent Multi-Heuristic A* (IMHA*) and Shared Multi-Heuristic A* (SMHA*). IMHA* independently explores the path by running n searches simultaneously, where n is the number of heuristic functions, and each search uses its own priority queue. Although the search algorithm is efficient, memory usage can be enormous. In contrast to

IMHA*, SMHA* shares the current path to a state. When a better path to a state is discovered by any of the searches, the information is updated in all the priority queues [2]. The path sharing feature allows SMHA* to expand each state to a maximum of two times, therefore both computation time and memory space are reduced compared to IMHA*.

Table 1 shows the properties of the search algorithms from above, along with their strengths and weaknesses. Each algorithm has its advantages in various domains, which 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 applications 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. To reduce the complexity to design an accurate search algorithm, our approach, EHA*, focuses on automating the design and optimization process of heuristic functions using GA.

Table 1. Comparisons between previous heuristic search algorithms including A*, WA*, ARA*, and MHA*.

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. Highly depended on parallel computing. Long searching time.

One of the most important features of EHA* is that it optimizes heuristic functions through GA. GA was first introduced by Holland [7], 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 [7]. 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 [7]. 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.

3. Methodology

We start by presenting the optimization models for the heuristic function in EHA*. An agent contains a 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. *Multi-Weighted-Heuristic (MWH) function*

Figure 2 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 value in MWH serves the purpose to prioritize each heuristic. However, it is time consuming to

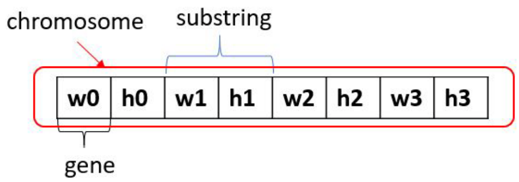


Fig. 2. Chromosome of an agent. The chromosome is a numerical representation of a MWH function. Each heuristic function contains multiple weight and heuristic. As the figure shows, a gene is a single parameter, whether a weight value or a heuristic index. A substring is a bundle of a heuristic index and a corresponding weight value.

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 (PIM), and Re-Initialization Model (RIM). The implementation detail of the characteristics from each model to optimize the heuristic function is presented in the following section.

3.2. *EHA* implementation models*

We choose to use 20 agents in our experiment for the convenience of experiment setup. The number of agents that 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 2. 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

Algorithm 1. Pseudocode of EHA*.

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

Table 2. 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

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 pseudocode 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

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
```

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.

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 demonstrate there are improvement for the Base Model, we present two additional models, the PIM and the RIM, 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 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 illustrated in Fig. 3. Individuals within each group crossover reproduce with each other during normal operations, while the inter-islands crossover reproduction happens only every Nth generation. 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 4 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

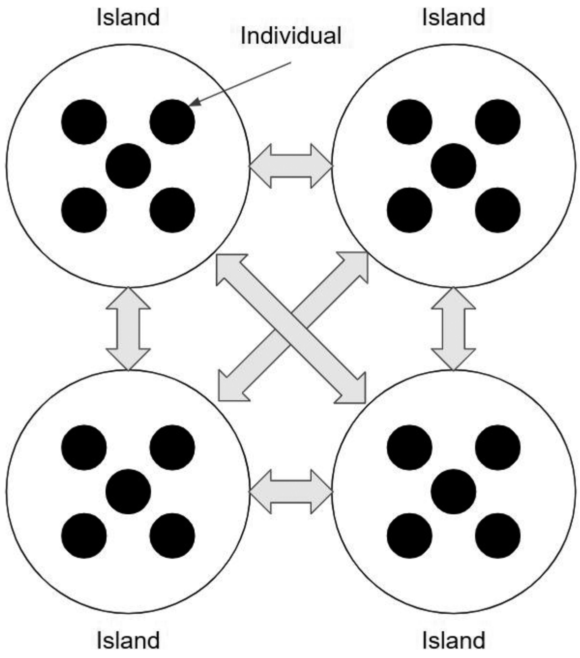


Fig. 3. Graphical representation of PIM.

the efficiency of the models. To reduce the impact of the initial population, new agents should join into the competition pool as every Nth generation.

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 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 Nth generation, and the rest of the settings remains the same as PIM.

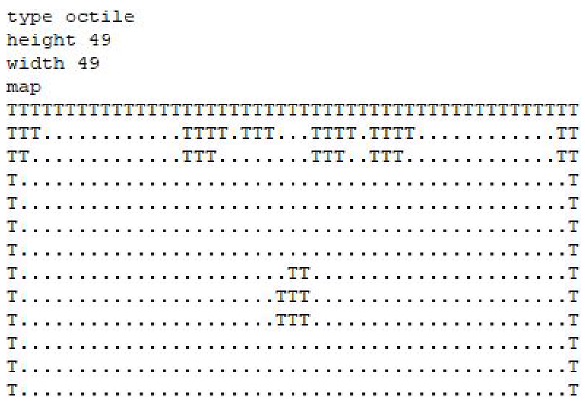
Table 3 summarizes each model implementation and configuration. The distribution of agents is based on the total population of twenty. In the experiments, we focus on the performance of EHA* to optimize the weighted heuristic functions, the accuracy of the heuristic functions, and the search performance 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.

3.3. *Benchmarks*

We are using grid-based pathfinding benchmark [4] which includes multiple maps to evaluate EHA*. The benchmark sets contain maps with high diversity including

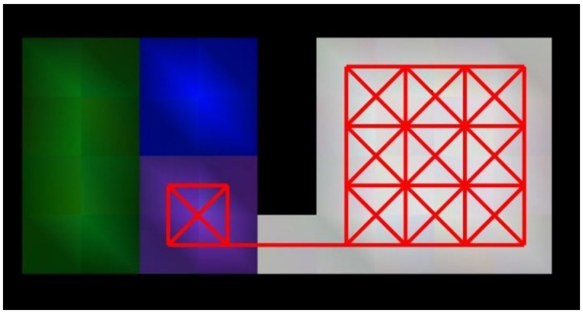
Algorithm 3. Pseudocode of Children Generation of PIM.

```
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
```



```
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)

Fig. 4. Sample map textual (a) and graphical (b) representation. The lines from (b) show the connectivity between each grid cell.

Table 3. 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.

		PIM		EIM		RIM	
	Base	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

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 [4]. As Fig. 1(a) 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 more than just binary. We modified the setting that the cost of walking on shallow water is doubled compared to normal ground, meaning it takes a cost of 2 units to cross 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 5 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

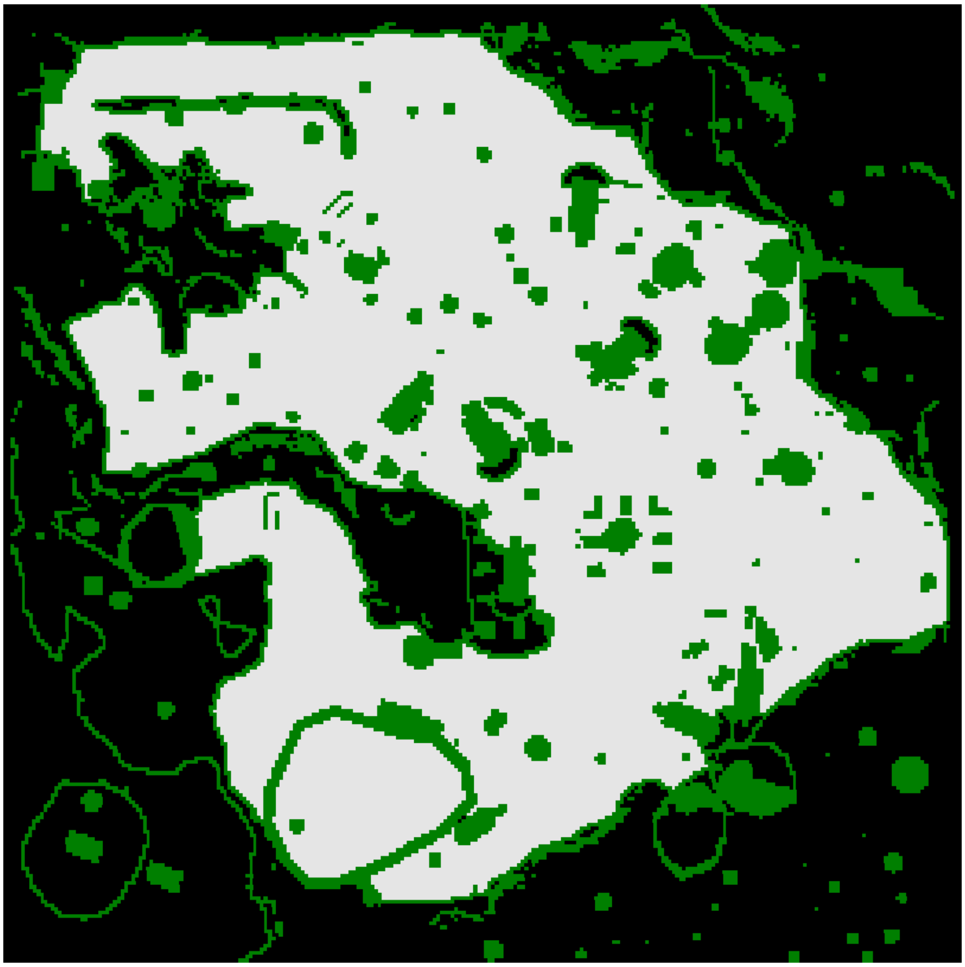


Fig. 5. Graphical representation of EIM.

possible. That is, it is not possible to take a diagonal move between two obstacles, nor take a diagonal move to cut a corner [4]. 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 have large local minima to increase the difficulty; (3) and mazes. Figure 6 illustrates a sample map from DAO. In our experiment, we chose the largest maps that have the size of 512 by 512. The largest map has 137,375 walkable states, while some maps only have a few hundred walkable states [4]. Therefore, the benchmark contains different levels of difficulty to evaluate the efficiency of a pathfinding algorithm.

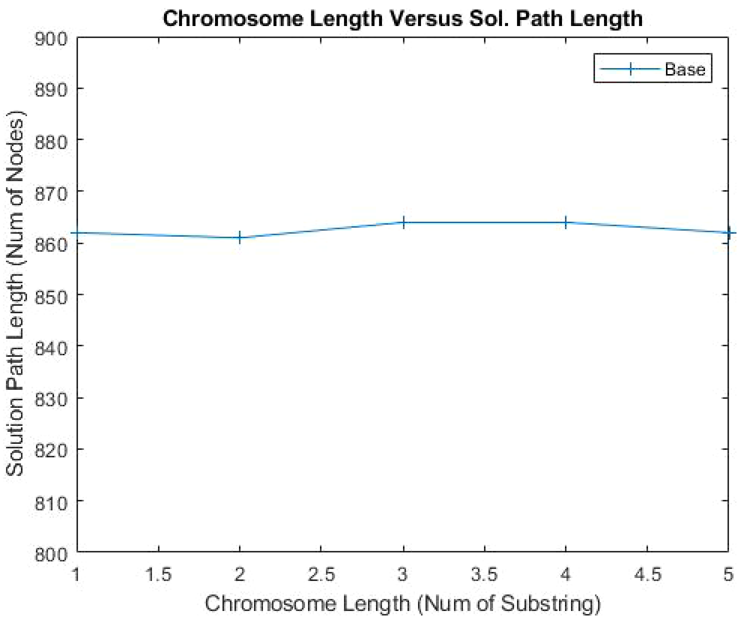


Fig. 6. Chromosome length versus solution path length.

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 from being too close and decrease the difficulty. Detail of the experiment setup will be explained in the beginning of the following section.

4. 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 rules to ensure that 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 and 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 WA*. With a special case of the weight equal 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 inclusiveness. 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 resulting 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.

4.1. *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 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 show the result of the Base Model. Moreover, the solutions have 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 less than or equal to 2, does not always find the solution because the priority queue is overflowed.

4.2. *Priority queue size*

Compared to solution path length, the priority queue size comparison is obviously more interesting. Figure 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 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.

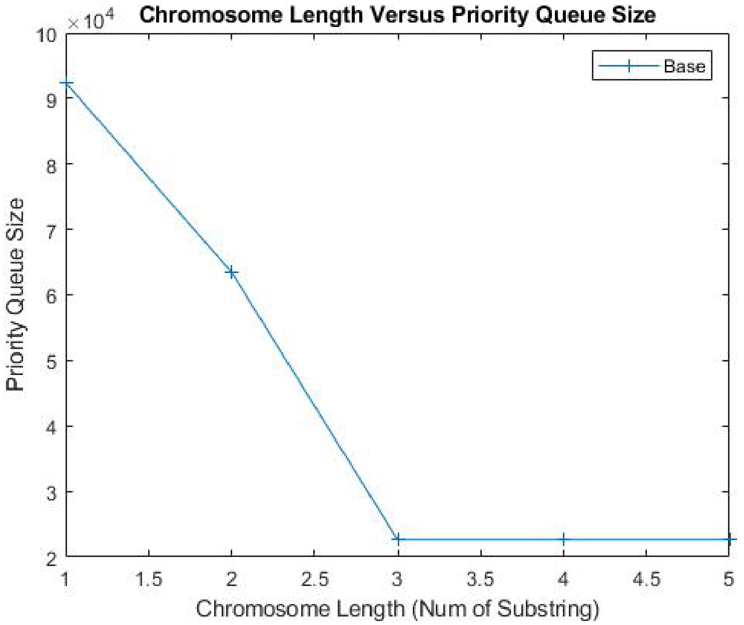


Fig. 7. Chromosome length versus priority queue size.

4.3. *Optimization time*

Figure 8 shows the effects of each model acting on the heuristic function optimization. The base model gradually improves the solution quality until it reaches the optimal; the PIM has a massive improvement in a short period of time; the EIM behaves similar 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.

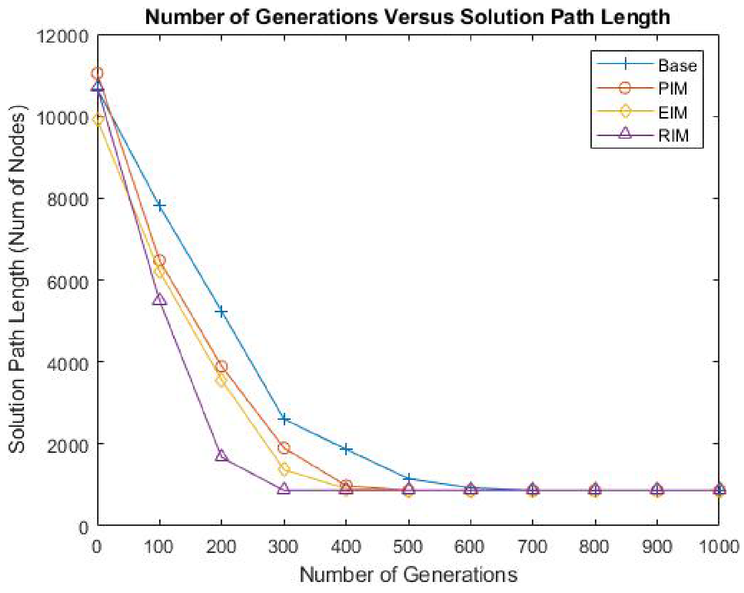


Fig. 8. Number of generations versus solution path length.

Figure 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 the 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 to

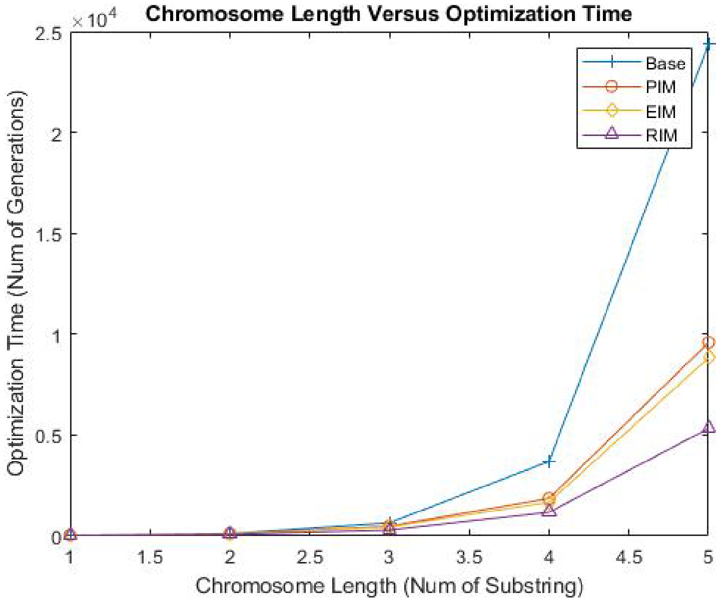


Fig. 9. Chromosome length versus optimization time.

design a very informative MWH that saves computational time and space. That is, EHA* is capable of designing 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 accurate weight and heuristic using GA.

5. Conclusion and Future Works

We presented an auto-optimized heuristic search algorithm named EHA* which has the capability to design and optimize a MWH 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 [8], videogames [9], and robotics [10]. The future researches can be focused on hierarchical GA [11], hierarchical pathfinding [12], pathfinding for dynamic environments [13], and a scalable architecture design on GPU [14] and

FPGA [15]. 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.

Acknowledgments

This material is partially supported by National Science Foundation (NSF) under the grant number 1416730.

References

[1] P. E. Hart, N. J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Systems Sci. Cybern.* **4**(2) (1968) 100–107.
[2] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang and M. Likhachev, Multi-heuristic A*, *Int. J. Robot. Res.* **35**(1–3) (2016) 224–243.
[3] I. Pohl, Heuristic search viewed as path finding in a graph, *Artif. Intell.* **1**(3–4) (1970) 193–204.
[4] N. R. Sturtevant, Benchmarks for grid-based pathfinding, *IEEE Trans. Comput. Intell. AI Games* **4**(2) (2012) 144–148.
[5] M. Likhachev, G. J. Gordon and S. Thrun, ARA*: Anytime A* with provable bounds on suboptimality, in *Advances in Neural Information Processing Systems*, 2004, pp. 767–774.
[6] C. M. Wilt and W. Ruml, When does weighted A* fail?, in *SOCS*, 2012, pp. 137–144.
[7] J. H. Holland, Genetic algorithms, *Sci. Am.* **267**(1) (1992) 66–73.
[8] A. Shkolnik and R. Tedrake, Path planning in 1000+ dimensions using a task-space voronoi bias, in *IEEE Int. Conf. Robotics and Automation*, 2009, pp. 2061–2067.
[9] W. Lee and R. Lawrence, Fast grid-based path finding for video games, in *Canadian Conf. Artificial Intelligence*, 2013, pp. 100–111.
[10] Z. A. Algfoor, M. S. Sunar and H. Kolivand, A comprehensive study on pathfinding techniques for robotics and video games, *Int. J. Comput. Games Technol.* **2015** (2015).
[11] E. D. de Jong, D. Thierens and R. A. Watson, Hierarchical genetic algorithms, in *Int. Conf. Parallel Problem Solving from Nature*, 2004, pp. 232–241.
[12] R. C. Holte, M. Perez, R. Zimmer and A. MacDonald, Hierarchical A*: Searching abstraction hierarchies efficiently, in *AAAI-96 Proceedings*, 1996, pp. 530–535.
[13] C. Astengo-Noguez and J. R. C. Gómez, Collective pathfinding in dynamic environments, in *Mexican Int. Conf. Artificial Intelligence*, 2008, pp. 859–868.
[14] A. Bleiweiss, Gpu accelerated pathfinding, in *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware*, 2008, pp. 65–74.
[15] G. R. Jagadeesh, T. Srikanthan and C. Lim, Field programmable gate array-based acceleration of shortest-path computation, *IET Comput. Digit. Tech.* **5**(4) (2011) 231–237.