9-12-2018

Exploiting Problem Structure in Pathfinding

Qing Cao
University of Windsor

Exploiting Problem Structure in Pathfinding

By

Qing Cao

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2018

Exploiting Problem Structure in Pathfinding

by

Qing Cao

APPROVED BY:

J. Gauld
Department of Chemistry and Biochemistry

M. Kargar
School of Computer Science

S. Goodwin, Advisor
School of Computer Science

Sep 10, 2018

ABSTRACT

With a given map and a start and a goal position on the graph, a pathfinding algorithm typically searches on this graph from the start node and exploring its neighbour nodes until reaching the goal. It is closely related to the shortest path problem. A* is one of the best and most popular heuristic-guided algorithms used in pathfinding for video games. The algorithm always picks the node with smallest f value and process this node. The f value is the sum of two parameters g (the actual cost from the start node to the current node) and h (estimated cost from the current node to the goal). At each step of the algorithm, the node with lowest f will be removed from an *open list* and its neighbour nodes with their f values would be updated in this list. The main cost of this algorithm is the frequent insertion and deleteMin operations of the *open list*. Typically, implementation of A* uses a priority queue or min-heap to implement the *open list*, which takes O(log n) for the operations in the worst case. But this is still expensive when using the algorithm in a large and complicated map with numerous nodes. We came up with a new data structure called multi-stack heap for the open list based on the 2D grid map and Manhattan distance, which only costs $O(1)$ for insertion and deleteMin. It is very efficient especially when we have a considerable number of nodes to explore. Additionally, traditional A* requires checking whether the *open list* contains a duplicated of the being inserted node before every insertion, which takes $O(n)$. We proposed a new implementation method based on admissible and consistent heuristic called “Check From Closed List”, it can reduce the time of this process to $O(1)$.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my supervisor Dr. Goodwin for being patient with me and helping me to come up with the idea of the multi-stack heap. Thanks to his plenty of guidance and encouragement, I had a great time to research the field of pathfinding. It is my great pleasure to be his student and work with him.

I would also like to thank my committee members Dr. Kargar and Dr. Gauld for taking the time to review my paper and attending my thesis proposal and defense. Thanks for their valuable guidance and suggestions to improve this thesis.

Finally, I would like to thank my family and friends for their support over those years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Thesis Claim

In this paper, we proposed a new data structure for A* algorithm to implement its *open list*, and it is named as multi-stack heap. Comparing with the unsorted array and the min-heap, which are traditional data structures used to implement the *open list* , multi-stack heap takes less time than the two data structures. Additionally, with the multi-stack heap, the number of operations executed during pathfinding is often significantly reduced. Taking advantage of admissible and consistent heuristic in pathfinding, we also proposed a new method "Check From Closed List" to replace the process of checking the duplicate node in the *open list* before the insertion by checking the *closed list*.

1.2 Pathfinding

Pathfinding, planning a path, is an important research topic in the Artificial Intelligence community. It is widely used in many popular fields, such as GPS, robotics, logistics, and crowded simulation and those fields are implemented in static, dynamic or real-time environments [1]. The video game is also a special field for using pathfinding techniques. For better user experience, modern games often have high demands on CPU and memory. However, graphics and physical simulations also take a lot of time to process [26]. Pathfinding is one of important elements in this part, which may integrate closely with departure and destination selection, the shortest path planning in a complicated environment. In this paper, we mainly worked on optimization of pathfinding efficiency in games.

1.2.1 Pathfinding Problem

Pathfinding problem closely refers to finding the best path between two locations that meet some criteria such as the shortest, lowest cost or fastest in a spacial network. The problem usually can be divided into two domains by the number of agents. When there is only one agent, i.e., find a best path for an agent with a given start point and a goal point on a given graph, we call this situation Single Agent Pathfinding (SAPF). On the contrary, when given a set of agents with a start point and an end point respectively, find the best paths for all agents while avoiding collisions, we call this problem Multi-Agent Pathfinding (MAPF)[19]. In this paper, we only refer to SAPF problem. Therefore, we provide background for the case of a graph G(V, E) with a start point S and a goal point G, the goal is to find the best path for the agent from point S to point G in the least time with an optimal length.

1.2.2 Graph Representation

The agent has to perform pathfinding on a spacial network. The network, we also call a graph representation, is a basic component of pathfinding. There are several ways to present a graph in games. Normally, we classify it in three classes, which are waypoints, navigation meshes, and grids, respectively.

In waypoint graphs (Fig.1), each node is called a waypoint and specifies a special location in the region; the waypoints are connected by different edges. An edge between two waypoints means that the agent can walk along this path without any collisions [30]. The main idea of the waypoint graph is to put some waypoints and edges on the map so that agents in the game can find their way to the destination around static obstacles. When the obstacles and the waypoints are fixed on a certain map, there is no change of the shortest path between any two nodes. In this way, some optimizations [27] [30] have been made based on waypoint since the waypoints

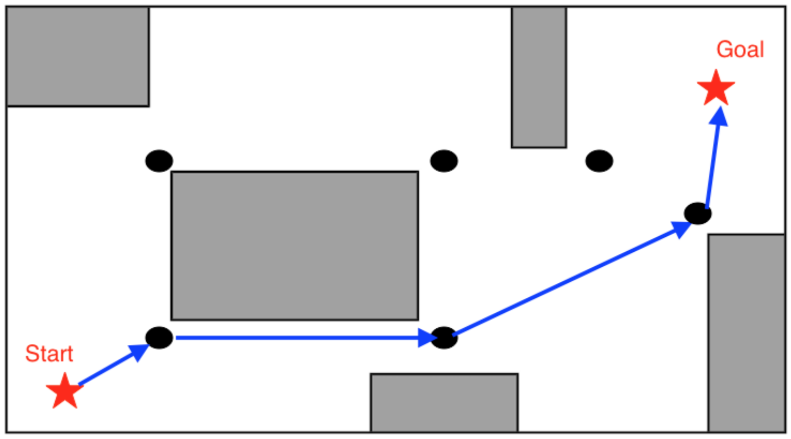

Fig. 1: Waypoints Graph

could be pre-processed on the map before the real execution of pathfinding. However, the disadvantage is that, when there are multiple characters, agents could get stuck on the waypoint. On the other hand, when the map is changed frequently or the obstacles are movable, the system has to pre-process the map again, which could be very expensive.

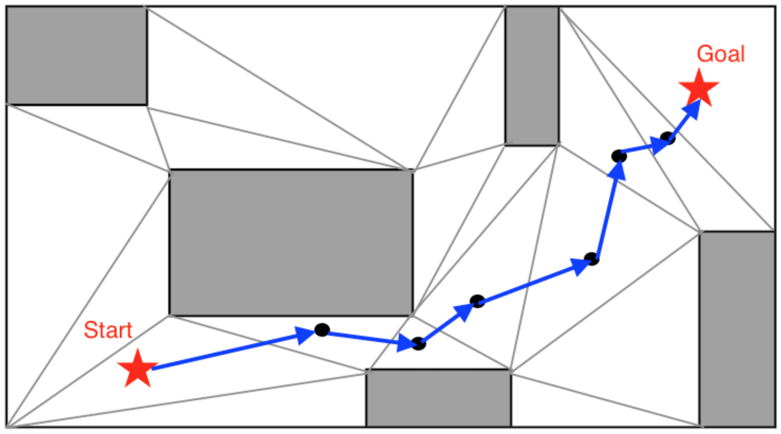

Fig. 2: Navigation Mesh Graph

Navigation mesh (Fig.2), also called navmesh, is formed by a group of connected convex polygons, in which every polygon defines a walkable area of the environment (no obstacles). At each polygon, the agent can access any point in that area, and to any other point in the same area since the polygon is convex [23]. Similar to the waypoint graph, the system has to pre-process the map to draw the navigation mesh to avoid obstacles, and the agent can reach the goal when it walks through a series of polygons. Thus, we can see that navigation mesh has the same disadvantages as the waypoint graph; it costs a lot to deal with meshes when the environment is dynamic, although some new algorithms and optimizations have been made for navigation meshes in [24] [3] [25] for a dynamic environment. The obvious advantage is that, since we represent a large area in a single polygon, the overall density of the graph can be decreased significantly. In this way, the memory footprint can be reduced since the number of stored nodes is decreased. Also, pathfinding times can decrease as the density of the searched graph shrinks [23].

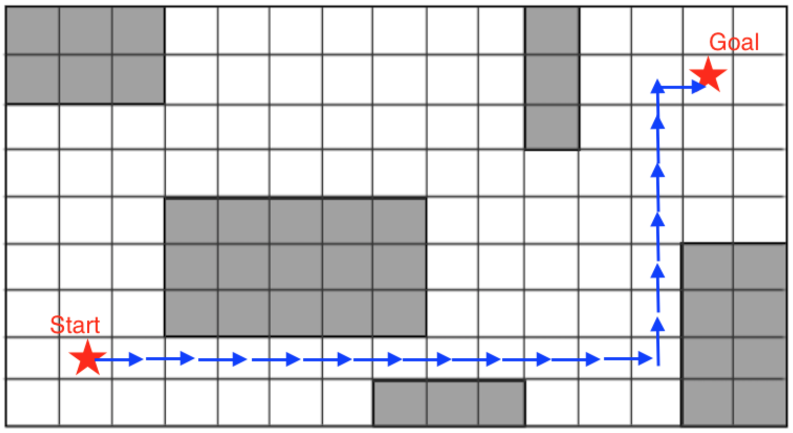

Fig. 3: Grid Graph (Squared-Grid Graph)

The third one is the grid graph (Fig.3), which is the most popular one to be used in research. Grid-based pathfinding is required in many video games and virtual worlds to move agents [14]. Grid graphs are made up by a collection of tiles. Each tile is regarded as a node in the graph, and it can be set as a traversable or non-traversable (obstacles) tile. Within different environments, the tiles could be in various shapes, such as square grid (Fig.4), triangular grid (Fig.5), and hexagonal grid (Fig.6). Compared with waypoint and navigation mesh graphs, we can observe that grid graphs contain both the walkable and obstacles information of the environment, while the other two graphs only refer to traversable areas, which means that grid graphs include the entire game environment and even the environment changes; it will not charge much to process the map since the only operation is to set the cell as traversable or

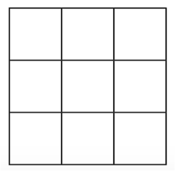
Fig. 4: Square Grid

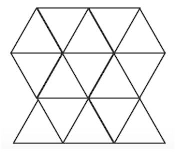
Fig. 5: Triangular Grid

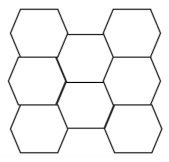
Fig. 6: Hexagonal Grid

non-traversable. Additionally, the gird-based graph can be generated relatively faster and it is widely used in a lot of pathfinding researches due to its simple block-like structure.

We did the research based on the squared-grid graphs as it is used most widely by games. Besides, it is easier to implement in the experiment and more obvious to see the experimental result.

1.2.3 Heuristic

The heuristic function is designed for solving problems with a better performance compared with classic methods: the outcome of the heuristic function is supposed to be more quick, more optimal, or shorter than original results; however, it may or may not end up with a better solution [15]. In pathfinding, the heuristic can decide which direction to explore at each searching step using some given information. In traditional search problems, heuristic means the estimation of the lowest cost between any node n to the destination node, and it is usually represented as h(n); it is a quick way to estimate how close the agent is to the goal.

To calculate the cost of two nodes, generally, three heuristic functions are used: they are Manhattan distance, Octile distance and Euclidean distance, respectively. Suppose we have two points, p1(x1,y1) and p2(x2,y2), the way to calculate the distance between these two nodes with different heuristic functions is shown in the following.

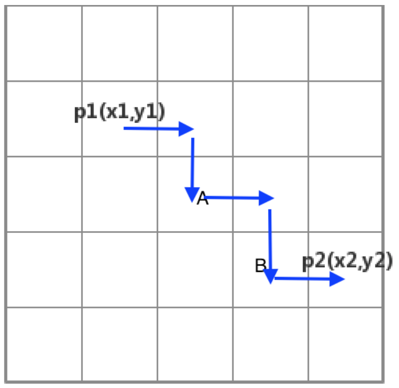

Fig. 7: Manhattan Heuristic

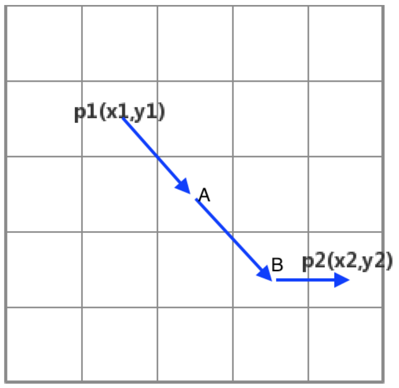

Fig. 8: Octile Heuristic

Manhattan Distance

Manhattan distance is the distance between two points in which path is strictly vertical or horizontal to the axes (see the path from Fig.7) [20]. The distance is formed by the sum of grid lines between the two nodes, and we can get the distance by the following formula:

$$h(p1,p2)_{Manhattan} = |x1 - x2| + |y1 - y2| \tag{1}$$

Octile Distance

Octile distance is the extension of Manhattan distance, which allows diagonal moves on a squared-grid graph [2]. In Fig.8, the agent can walk diagonally from p1 to A, from A to B, and then move horizontally to p2. This path is shorter than the final path in Fig.7 since in Manhattan distance, it takes two units cost from p1 to A (two steps) while in Octile distance only $\sqrt{2}$ units cost are used (one step), and it is the same process from point A to point B. Therefore, it can be concluded that when the agent is allowed to walk diagonally, we can get a shorter solution path. The following formula could calculate the Octile distance:

$$h(p1,p2)_{Octile} = \max((x1-x2),(y1-y2)) + (\sqrt{2}-1) * \min((x1-x2),(y1-y2)) \tag{2}$$

(1.4 is commonly used for $\sqrt{2}$ in the experiment)

Euclidean Distance

Euclidean distance is the straight-line distance between two points, which means when there is no obstacle on the path, the agent can move directly from the start point to the end point without any turns (Fig.9). In this way, the length of the path is shorter than the Manhattan distance and Octile distance (triangle inequality theorem). The following formula can determine Euclidean distance:

$$h(p1, p2)_{Euclidean} = \sqrt{(x1 - x2)^2 + (y1 - y2)^2} \tag{3}$$

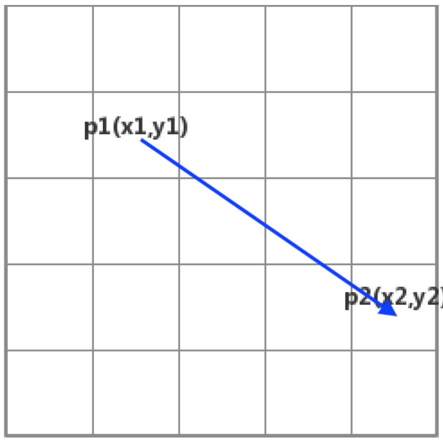

Fig. 9: Euclidean Heuristic

Summary

From the above description of different heuristic functions, it can be observed that when the agent is only allowed to walk vertically or horizontally, i.e., move to east, south, west and north (four directions only), Manhattan heuristic could be the best choice to estimate the distance to the goal; based on walking vertically and horizontally, Octile heuristic could perform best when the agent can also move diagonally, i.e., the agent is permitted to move to east, south, west, north, northwest, northeast, southwest, southeast (eight directions). As for Euclidean distance, from Fig.9 we can observe that there is no limit for moving directions since it calculates the direct distance between two points, the agent could walk in any directions along the Euclidean path. However, there is no fixed heuristic function for the agent based on moving directions. For instance, Euclidean heuristic also could be used when the agent is authorized to walk in four directions only, but as the agent can only move in vertical or horizontal directions, the actual cost (distance) should be longer than the estimated distance by Euclidean heuristic.

1.3 Search Algorithms

There are many search algorithms for solving pathfinding problem, though they can usually be classified under two categories: uninformed search and informed search. Uninformed searches, also known as blind searches, refer to search algorithms that only have knowledge of a start point and local connectively of the graph but have no knowledge of the destination [7]. As a result, they usually traverse all possible locations until reaching the target [7]. Breadth-first search and depth-first search are the typical search algorithms in this type. In contrast, uninformed searches, informed searches utilize local connectivity and some localized knowledge of the graph, such as the location of its goal, the cost to travelling to the destination, to make heuristic decisions during search [7]. In other words, informed searches do pathfinding with some idea of how close the agent is to a given destination. This generates more efficient results than uninformed search because the character does not expand nodes that they know are not on the path to the target. In this classification, the most typical algorithms are greedy search algorithm, Dijkstra's algorithm [6] and A* algorithm [10]. However, A* is the one used most widely in game applications and has a great deal of optimized algorithms, such as Hierarchical A* [12], Windowed Hierarchical Cooperative A* [21], IDA* (depth-first iterative-deeping A*) [13] and EPEA* (Enhanced Partial Expansion A*) [9]. As A* is the most basic and popular algorithm in research and real implementation, it is important to critically explore and analyse how A* algorithm is used, and how to improve and maximize its application. Therefore, we mainly worked on A* algorithm in this thesis , and it will be explained specifically in the later section.

1.4 Properties of A* Algorithm

1.4.1 A* Algorithm

A* is one of the most effective and popular heuristic-guided algorithms used in pathfinding. Given a certain Graph $G(V, E)$ with a start node and a goal node, A* has the ability to efficiently find an optimal path from the start node to the goal node.

Searching from the given start point on the graph, A* builds a tree of all possible paths that beginning from this start node, and every time, it expands one step further on each path until it reaches the node on one of its paths that is the predetermined goal node. To know the specific A* algorithm (Algorithm.1), several variants of A* must first be understood. The process begins on start node, the agent will search the graph and reach a node, which is represented by n. The distance between the start node and n is $g(n)$, while $h(n)$ is a heuristic function that estimates the cost from node n to the goal node. The sum of $g(n)$ and $h(n)$ is represented by $f(n)$:

$$f(n) = g(n) + h(n) \tag{4}$$

A* uses the f value (i.e., the total estimate cost of path through the node n) as the evaluation to determine the node to be expanded in every step. Additionally, A* maintains two sets to store the different nodes: the *open list* and the *closed list*. The *open list* keeps track of nodes that are waiting to be examined in the future, while the *closed list* stores nodes that have already been explored. To ascertain the entire path lately, each node that has been expanded requires a pointer to its predecessors.

In Algorithm.1, A* uses a main loop to repeatedly gather nodes. *Current* is the node with the lowest f value from the *open list*, if *current* is the target node, then the path between it and start node must be found. It requires the agent to track the predecessors from the final node until reach the node whose parent node is the start node, then revises this path to map out the final path. However, if the *current* is not the target node, it will be removed from the *open list* and added to the *closed* *list*. Afterwards, A* generates all possible neighbor nodes to *current*. If a neighbor is already in the *closed list*, it is discarded and the agent will move onto other neighbors. if the neighbor is not in the *closed list*, then the agent will verify whether it is in the *open list*. If not, it will be added to the *open list*, and the agent will calculate the f value of the node and set *current* as its predecessor. However, if the neighbor is already in the *open list*, the agent will calculate the f value of this node and compare it with the duplicated node in the *open list*. If its f value is less than the duplicate, it will replace the duplicate in the *open list*, and the agent will set its predecessor as *current* before moving on. If its f value is equal to or greater than the duplicate, it will be discarded directly and the duplicated one will be retained in the *open list*.

1.4.2 Optimality

The optimality of a solution may differ depending on the situation and could refer to the fastest, shortest, or most efficient solution. In traditional pathfinding problems, search algorithms are always expected to find an optimal path, which means the closest between two nodes. In A*, the agent has to select the best possible optimal nodes to keep on moving when it walks through several intermediate path. This optimal choice allows the agent to achieve a high performance in the environment, specifically the shortest path from the start to the goal node [18]. This solution is called an optimal path.

1.4.3 Admissibility and Consistency

The use of an admissible heuristic function is critical to pathfinding when the goal is to guarantee that the final solution is an optimal path [11]. In A*, the heuristic is used to estimate the cost of reaching the goal node. An admissible heuristic in A* means its estimated cost to the target node, represented as $h(n)$, would never exceed the actual cost it takes from the current location to the goal, represented as $h(n)^*$. According to this requirement, $h(n) <= h(n)^*$ should always be true if we A* is admissible.

Algorithm 1 A* Algorithm

```
Input: A Graph $G(V, E)$ with start node $start$ and end node $goal$
Output: Least cost path from $start$ to $goal$
1: Initialize:
2:     $open\_list = \{start\}$
3:     $closed\_list = \{ \}$
4:     $g(start) = 0$
5:     $f(start) = heuristic\_function(start, goal)$
6: while $open\_list$ is not empty do
7:    $current$ = the node in $open\_list$ having the lowest $f$ value
8:    if $current = goal$ then
9:       return “Path found”
10:   end if
11:   $open\_list$.delete($current$)
12:   $closed\_list$.insert($current$)
13:   for each $neighbour$ of current do
14:      if $neighbour$ in $closed\_list$ then
15:         continue
16:      end if
17:      if $neighbour$ not in $open\_list$ then
18:         $open\_list$.insert($neighbour$)
19:      end if
20:      if $g(current) + distance(current, neighbour) < g(neighbour)$ then
21:         $g(neighbour) = g(current) + distance(current, neighbour)$
22:         $f(neighbour) = g(neighbour) + heuristic\_function(neighbour, goal)$
23:         $neighbour$.setParent($current$)
24:      end if
25:   end for
26: end while
27: return “Path not found”
```

Apart from the requirement of admissibility, in A*, the heuristic function also needs to be consistent to guarantee the optimality of a solution path [17]. Consistency means the estimated cost ($h(n)$) to the goal is always less than or equal to the estimate cost ($h(n')$) from any neighbour of the current node to the goal plus the cost ($c(n, n')$) from current location to this neighbour [29], i.e., it satisfies the triangle inequality theorem (Fig.10):

$$h(n) \leq c(n, n') + h(n') \tag{5}$$

while node n is the current location of the agent, n' is a successor of n, and G means

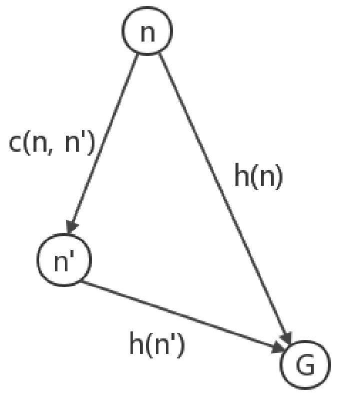

Fig. 10: Consistency Diagram

the goal. From the evaluation function of A* that $f(n) = g(n) + h(n)$, the function for node n' could be described as $f(n') = g(n') + h(n')$, which would provide an induction as follows:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, n') + h(n') \\ &\geq g(n) + h(n) \\ &\geq f(n) \end{aligned} \tag{6}$$

The induction indicates that the f value of node n is always smaller than or equal to the f value of its neighbours, which means that when examining the neighbour node, n in the *closed list* will never be updated again as its f value is always smaller than its neighbour's. Thus, if the heuristic is consistent and if the exploring node is discovered to already be in the *closed list*, A* algorithm moves on without doing anything. A theorem can be gained through the consistency:

Theorem 1 *If the heuristic is consistent, f value along any path is non-decreasing.*

Theorem 1 is an important theorem as it will be used in the implementation. Since there is no decrease of the f value on paths when the heuristic is consistent, it is not necessary to have a decreasing operation of the *closed list*.

Additional, if the heuristic is admissible and consistent, another conclusion could be obtained as:

Theorem 2 *If the heuristic is admissible and consistent, A* could find an optimal path [12].*

The heuristic of Theorem 2 is admissible means that there exists the optimal path in the environment since $h(n) <= h(n)^*$. For example, if the final actual cost from the start to the goal is C^*, which is also the cost of the optimal path since the heuristic is admissible, so we can conclude that A* only expands nodes whose $f$values are smaller than or equal to C^*, and the f value of the last node on the path should be same as the final optimal length. And as the heuristic is consistent, the f value of nodes on paths in the *closed list* would never be changed. When combined with the f value of the goal node $f(goal) = C^*$, it can be proved that the solution path that A* finds is the optimal one.

Our experiment only used admissible and consistent heuristics for the implementation of A*.

1.5 Thesis Contribution

The main cost of A* is the frequent insertion and deletion of the *open list*. Insertion occurs when the algorithm is expanding neighbour nodes, as it needs to add those neighbours with their information in the *open list*. The deletion operation is required in each step when moving the node with the lowest f value from the *open list* to the *closed list*. Therefore, to enhance the efficiency of the A* algorithm during pathfinding, it is critical to implement an open list in an efficient data structure so as to increase the speed of the insertion and deletion operations. Typically, the *open list* of A* is implemented by a priority queue or min-heap to improve performance, which takes $O(log\ n)$ to carry out the insertion and deletion operations. However, this is still very expensive when using A* on a large and complicated map with numerous nodes.

The current study introduced a new data structure called multi-stack heap to store the open list for A* algorithm based on 2D squared-grid map with Manhattan distance, which only takes $O(1)$ to insert and delete an element in the *open list*. It is more efficient, especially when we have a considerable number of nodes to explore comparing with other data structures. To address the frequent checking of duplicated nodes in the *open list* before every insertion, another implementation method is proposed: "Check From Closed List" method, which could save time from $O(n)$ to $O(1)$. Moreover, the implementation of data structures was also optimized by using the LIFO rule to select some nodes among nodes with the same f value to reduce the number of nodes A* must explore.

1.6 Thesis Organization

This paper is divided in six sections. The first chapter offers an introduction and describes the basic information of pathfinding and search algorithms with a particular focus on the A* algorithm, which is principal component of the current study. The second chapter reviews some typical data structures that host open lists for A* and offers some analysis of their respective performances. The third chapter introduces the current studys proposed data structure multi-stack heap in detail and provides the theoretical analysis of the multi-stack heap. The fourth chapter describes a new implementation of A* with respect to checking the duplicate nodes in the open list before the insertion. The fifth chapter outlines the experimental setup, results, and analysis. In this chapter, different data structures, such as the unsorted array and the min-heap are implemented and compared with the multi-stack heap. The sixth chapter offers a summation of the current studys key findings, while the seventh chapter proposes future research.

CHAPTER 2

Literature Review

2.1 Operations of Open List

A* algorithm has two sets, *open list* and *closed list*, to store the nodes that are waiting to be examined and nodes that have already been examined respectively. There is a main loop that is utilized to repeatedly select the node with the lowest f value from the *open list*, add it in the *closed list*, and insert its neighbours in the *open list* until the agent finds its goal. The main cost of A* is the frequent insertion and deletion operations associated with the *open list*, and insertion operations of the *closed list* as the heuristic is consistent. Thus, an efficient data structure for the *open list* is critical for the performance of A* algorithm.

There are four main operations of the *open list*: "deleteMin", "whetherContains", "insertNew" and "decreaseKey". In each iteration, A* algorithm must find and remove the node with the smallest f value in the *open list* (deleteMin). Once the agent has removed the node from the *open list*, it explores all neighbours of the deleted node, and insert them in the *open list*. Before the insertion, A* algorithm will verify whether the inserting node is in the *open list* (whetherContains). If it is a new node for the list, it is inserted into the *open list* (insertNew). However, if the node is already in the list, and its f value is smaller than the one already in the *open list*, then the duplicated node in the *open list* needs to be updated (decreaseKey). With regard to updating the node in the *open list*, there are normally two options. One involves simply changing the information of the node in the *open list*, more specifically, resetting the smaller f value for the node and updating its parent node. The other involves discarding the repeated node in the *open list* and adding this new node as a replacement.

2.2 Data Structures

In the implementation, there are many choices of data structures for the *open list*. The most basic and simplest data structures are unsorted array, unsorted linked list, sorted array, sorted link list. This series of data structures can be classified as the array. The current study discusses the unsorted array and sorted array as representatives as the array and introduces some other frequently-used data structures, such as the hash table and min-heap [28]. In addition, the current study likewise analyses the hot queue [4] for the use of the *open list*. The following discussion about the time complexity is based on, having n nodes in the *open list*.

2.2.1 Array

For the unsorted array, insertNew is the most simple operation, which takes $O(1)$. However, deleteMin requires $O(n)$ in order for the unsorted array to scan the array, find the best node, and remove it. To verify whether the node is in the list, the unsorted array also takes $O(n)$. It needs $O(n)$ to decreaseKey as A* has to scan the *open list* first to find the duplicated node, but there is no other expense associated with the updated node as the *open list* is unsorted.

As for the sorted array, the insertNew operation needs $O(n)$ as the new node should be put in the sorted order. Finding the best node and remove it is fast in this data structure, which only takes $O(1)$ since the array is already sorted and the node with smallest f value is already at the end of the *open list*. As we can use the binary search to check the *open list*, which allows whetherContains to take $O(log\ n)$ to finish the job. For the decreaseKey, it needs $O(log\ n)$ to find the node, and $O(n)$ to the updated node in sorted order.

2.2.2 Hash Table

In the hash table, sometimes it happens that when we apply a hash function to two different keys, it generates the same index for both keys. However, those two items cannot be located in the same address, this situation is called collisions of the hash table [16]. To lower the collision probability, the hash table is usually set twice as big as n. As every node has an individual key in hash table, normally, it only takes $O(1)$ to do the insertNew, whetherContains and decreaseKey. However, when the collision occurs, the time complexity still depends, the worst case could be $O(n)$. To find the minimum f value from the *open list*, the algorithm still needs to scan the whole hash table, so that deleteMin takes $O(n)$.

2.2.3 Min-Heap

The most typical implementation of the *open list* is the min-heap. Min-heap is a complete binary tree, the value of each node on this tree is smaller than or equal to its children's values. In A*, the value indicates the f value. Min-heap is designed for two basic operations, insert a new node and delete the node with the minimum value.

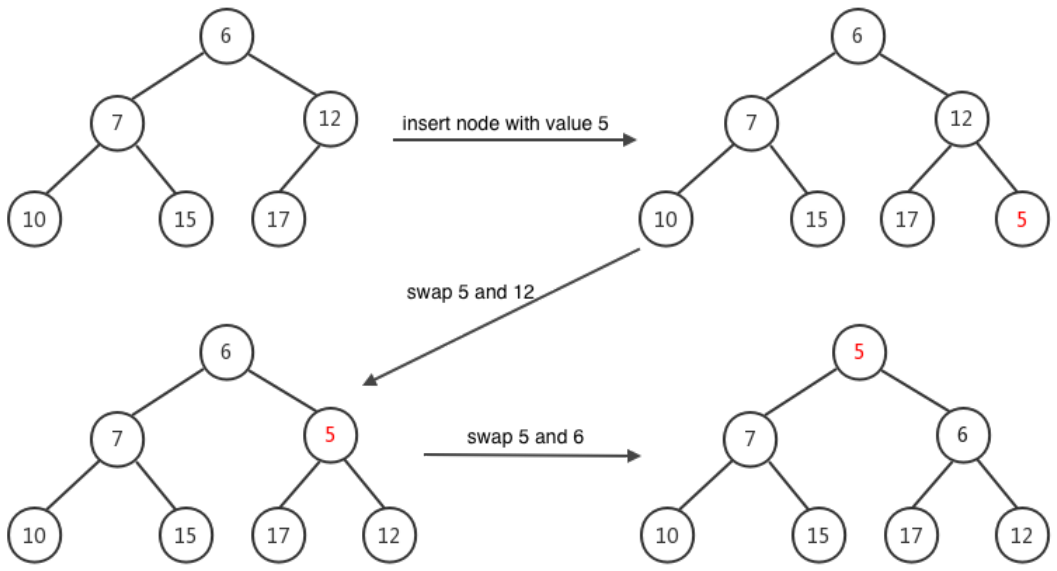

Fig. 11: An Example of Min-Heap Insertion

For the insertion (Fig.11), the new node is initially appended to the end of the tree as the last leaf node. After that, repair the heap by comparing the value of inserted element with its parent's value, swap the position of the two nodes if the added element has a smaller value, and keep the same process until the heap is a min-heap (i.e., the value of each node on this tree is smaller than or equal to its children's values), this process is also named "bubble up".

The smallest element can be found at the root of the heap, therefore, to delete the minimum node, remove the root. After the deletion, move the last node from the deepest level of the tree to the root position, compare the new root value with its children, swap the position with the element with the smallest value and keep comparing and swapping until the each node of this branch has a smaller value than its children, another name of this process is "bubble down".

Combined A* with the min-heap data structure, the insertNew and deleteMin are same as the insert and delete operations of the min-heap respectively, which take $O(log\ n)$. For operating the whetherContains function, it is required to traverse the whole *open list*, thus $O(n)$ is needed. To decrease key on the heap, it takes $O(n)$ to find the element and $O(log\ n)$ to repair the heap (as the updated element is always with a smaller value, only bubble up is needed to repair the heap).

2.2.4 Hot Queue

Hot queue, also called heap-on-top priority queue, is a combination of the multi-level bucket data structure of Denardo and Fox [5] and a heap [4]. It is divided into k-level buckets; the topmost bucket is a min-heap while the other buckets use the unsorted array to store nodes. All buckets have a specific range. As for how to set the k value and range of every bucket, the authors of the hot queue did not give a fixed function, it depends on the specific circumstance.

Put the *open list* in the hot queue, time for insertNew and deleteMin is $O(log\ n/k)$ in the top bucket which is same as the min-heap data structure. However, when the inserted element is not in the range of the top bucket, then it takes $O(1)$ to put in other buckets. After the deletion, if the top bucket is empty, then the next bucket within the unsorted array is converted into a min-heap, this process needs to be completed in $O(n/k)$. For the whetherContains, it still uses $O(n)$ to scan the *open list*. Decreasing key takes $O(n)$ to find the element, if the element is in the topmost bucket, it needs another $O(log\ n/k)$ to decreaseKey on the heap, however, if it is in the other bucket, there is no action needed to decrease the key.

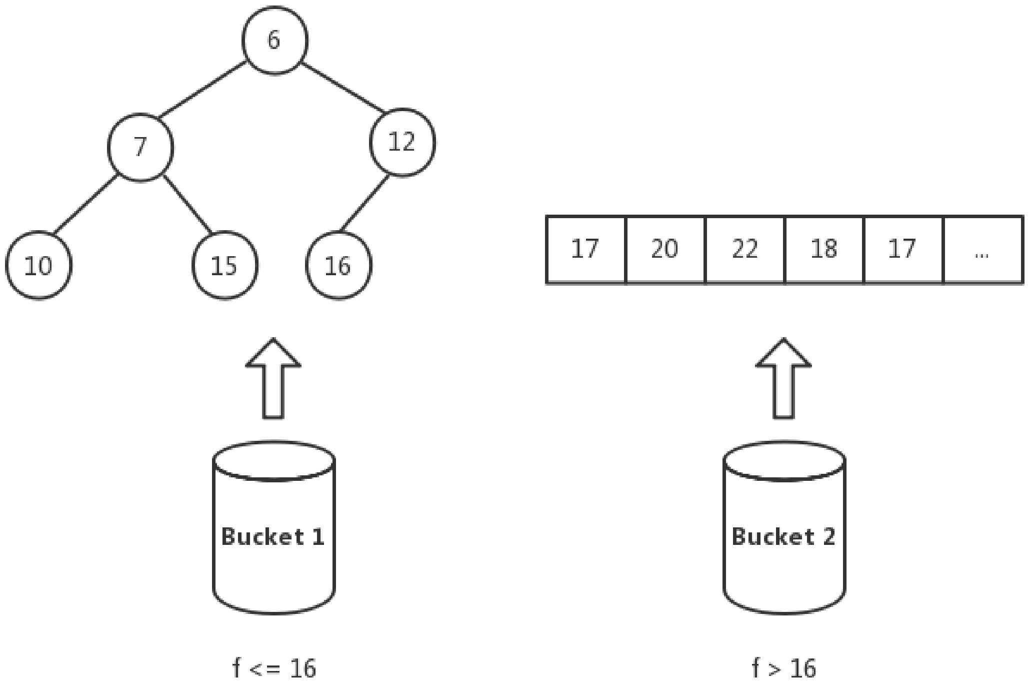

Fig. 12: An Example Diagram of Hot Queue with 2 Buckets

Considering the fact of A* that not every node in the *open list* is necessarily examined, we can take advantage of hot queue data structure as setting the hot queue as a 2-level buckets, put nodes with small f value in the first level as a min-heap, and put nodes with large f value in the second bucket as an unsorted array. However, this is just a theoretical idea, in the real implementation, how to define the "small" and "large" f value still depends on the specific situation (Fig.12).

2.3 Summary

From the above discussions about four operations of the *open list* and different data structures, we can compare the performance of each data structure with various operations as Table.1.

Data Structures	insertNew	deleteMin	whetherContains	decreaseKey
Unsorted Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Array	$O(n)$	$O(1)$	$O(log\ n)$	$O(n) + O(log\ n)$
Hash Table	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Min-Heap	$O(log\ n)$	$O(log\ n)$	$O(n)$	$O(n) + O(log\ n)$
Hot Queue	$O(log\ n/k)$ or $O(1)$	$O(log\ n/k)$ or $O(n/k)$	$O(n)$	$O(n) + O(log\ n/k)$ or $O(n)$

Table 1: Time Complexity Comparison of Different Data Structures

It is hard to decide which data structure is the best since every data structure has its advantages. For example, the unsorted array is good at insertNew while sorted array performs well at deleteMin and whetherContains. Also, all data structures perform similarly with a small number of nodes in the *open list*, i.e., n is relatively small. Considering whehtherContains must scan the whole *open list* (except for hash table and sorted array, although hash table could also take $O(n)$ when collision happens) to act operations; decreaseKey is a special case in pathfinding and it happens rarely, it will not take a large portion of time even when it occurs sometimes. Therefore, for the comparison, we focus on insertion and deletion. As we can see from the Table.1 that the total time complexity of insertNew and deleteMin for the unsorted array, sorted array and hash table is $O(n)$, while the min-heap only needs $O(log\ n)$ to perform the operations. The hot queue may even better than the min-heap when it operates at the topmost bucket, which only takes $O(log\ n/k)$. However, when it precesses nodes in other buckets, the maximum time complexity could be $O(n/k)$, which could be better than the unsorted array and sorted array, but it still depends on the number

k when it compares with the min-heap.

As there are some uncertain factors of hot queue, we can confirm that it should perform better than the Array series, but for the hash table and min-heap, it still depends since hash table could take advantage of whetherContains and min-heap has the possibility to be better than hot queue too, we do not consider the hot queue in our later research. The performance of hash table is also unstable, if there is no collision happens, it would be a more efficient data structures comparing with Array definitely, and it also could be better than min-heap at whetherContains and decreaseKey operations. But when collisions happen, it is hard to say which one is better, therefore, the hash table is not in the final comparison of our research as well.

After reviewed and compared all those data structures, we regard the min-heap as the most competitive data structure for the *open list*, which is the main data structure used and compared in our later research.

CHAPTER 3

Multi-Stack Heap

3.1 Motivation

When we researched pathfinding problem, found that whether Single Agent or Multi-Agent pathfinding, most of those searching algorithms are based on A*. For the implementation of A*, there are several typical data structures to perform the *open list*, while the *open list* is the most consuming part of A* algorithms, which are already discussed in literature review section. Those data structures are pretty efficient to store the *open list*, and can be commonly used in different situations and environment. However, we just found that grid maps have a structure that can be exploited when we were trying to do some simple implementation of A* within different maps. For the squared-grid graph, we have several observations when operating the pathfinding with Manhattan distance.

Suppose we use Manhattan distance as the heuristic, and the agent is allowed to walk only horizontally or vertically. Set the searching environment as Fig.13: white cells are the traversable grids while black blocks indicate obstacles on the graph that the agent cannot walk through; S and G means the start node and goal position of this pathfinding problem; the map is set on a coordinate axis, so every node on the graph can be represented by coordinate (x, y); the cost of every cell is 1, for calculating the h an f value. In each cell, the number at the bottom left corner is the h value of this node while the bottom right corner value is the f value, and the value at the top left corner indicates the f value; additionally, there is an arrow points to its parent node. Nodes in the purple background mean they are removed from the *open list* and have been put in the *closed list*, blue background means the node is in the *open list*.

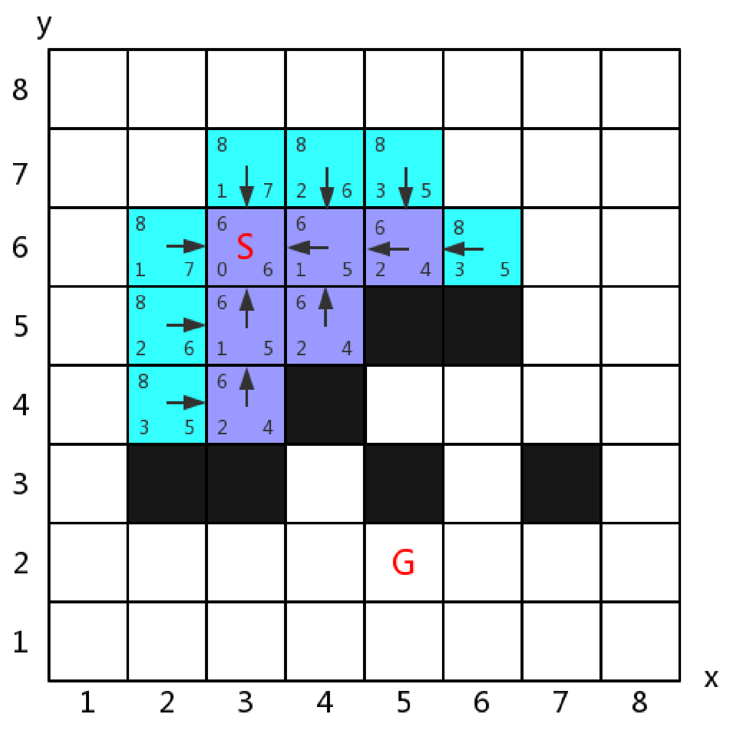

Fig. 13: An Example Environments for A* Pathfinding (all nodes with f value “6” have been explored)

In the Fig.13, nodes with f value “6” have been explored already and they are removed from the *open list* to the *closed list*. Currently, the *open list* still has other nodes, whose f values are “8”. The next step is still finding the node with the smallest f value from the *open list* and examine it until we find the determined goal. As all node in the *open list* with the same f value, the algorithm could pick one randomly and keep going. For instance, the node $(6, 6)$ is the next one to be explored, firstly, the algorithm checks whether it is contained in the *closed list*, which is not, then removes it from the *open list* and inserts the node in the *closed list*. At the same time, expanding its neighbours nodes $(6, 7)$, $(7, 6)$, $(6, 5)$ and $(5, 6)$ and calculating their f value respectively (Fig.14). As the node $(6, 5)$ is an obstacle, it is discarded without any operation. The node $(5, 6)$, whetherContains could find out that it is already in the *open list*, comparing their f values, there is no operation of this node as well as their f values are the same, it is not necessary to update it. Node $(6, 7)$and $(7, 6)$ are inserted in the *open list* as new nodes.

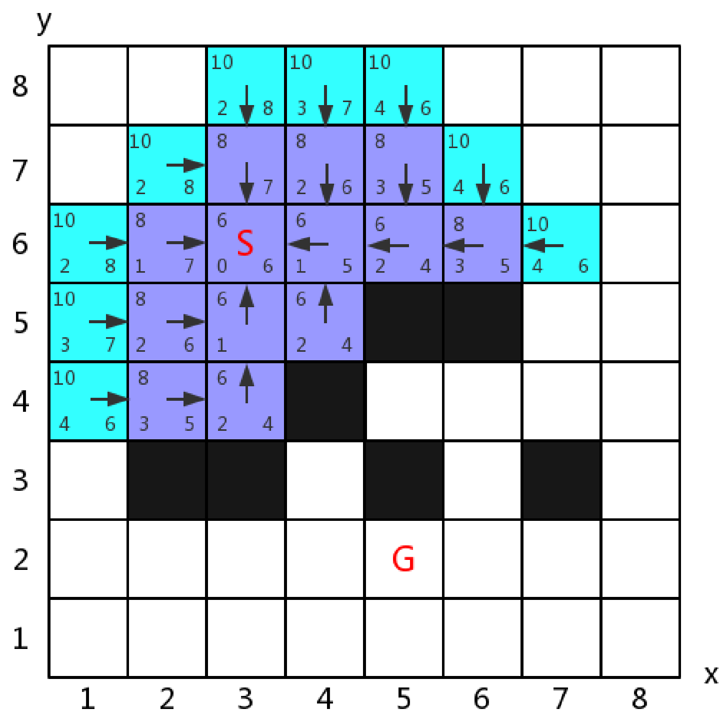

Fig. 14: An Example Environments for A* Pathfinding (all nodes with f value "8" have been explored)

In the Fig.14, it can be seen that all nodes with f value "8" have been explored and the goal has not been found yet. Meanwhile, some nodes with new f value "10" appear in the *open list*. Again, A* has to pick the node with minimum f value again to do the examination until the agent reaches the target. Let's say, we pick the node $(7, 6)$, and expand its neighbours, we could found that a new f value "12" is generated, $(7, 7)$ and $(8, 6)$ are inserted in the *open list* with f value "12" and $(7, 5)$ is added with f value "10". Repeat the exploring process until the goal is found like the status in the Fig.15. And after reaching the goal, follow the pointer to the parent of every node on the path until finding the start node, get the final path by revising the pointer on this path from the start node to the goal node. The pathfinding task is finished in this environment and the optimal length of the final solution is 10.

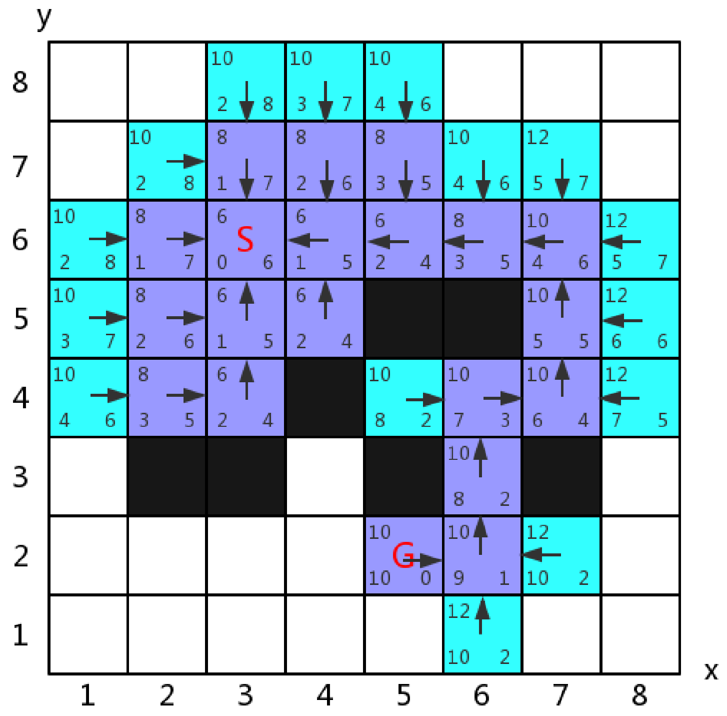

Fig. 15: An Example Environments for A* Pathfinding (find Goal and stop searching)

From the process of solving this pathfinding problem using A*, we have two observations: On a squared-grid graph, using Manhattan distance as the heuristic function (only move in four directions), the f value of a node is always equal to or 2 less than its children's f value; there are at most 2 different kinds of f values in the *open list* at the same time.

When A* is expanding children of the node n and inserting them in the *open list*, the g value of those neighbour nodes should be 1 more than node n as the agent walks 1 step further. However, g value could be different. If the agent is walking in a correct direction, i.e., moving toward the goal, then the g value of this child should be 1 less than node n. Therefore, compared to n, children nodes' g values are 1 bigger, h values are 1 smaller, so that

$$f(n) = f(n.children) \tag{1}$$

this equation is proved. Take the Fig.15 as an example, whether the node $(3, 5)$ or $(5, 6)$, comparing to their parent node, they are always 1 step closer to the G than their predecessors, this is the reason that they have the same f value as their parent. Nonetheless, when the expanded neighbour node is located backward (relative to its parent node n) to the destination, the h value of this child should be 1 larger, because it makes the agent 1 step further to the destination. This happens a lot when the agent wants to avoid obstacles. Check the node $(5, 6)$ in the Fig.15, to reach the destination in a most efficient way, it should move to $(6, 5)$, but as this node is an obstacle, and there is no other choice to get closer to the target node, the agent has to avoid this obstacle to reach the goal by moving to $(6, 6)$. Node $(6, 6)$ is at the opposite direction to the destination node by walking through $(5, 6)$. As both g and h values decrease 1 separately, f value of this child node could be 2 more than the node n. We can get

$$f(n.children) - f(n) = 2 \tag{2}$$

which is correct as well in the specific case. Therefore, the first observation is true.

As for the second observation, primarily, we know that A* always explores the node with the lowest f value from the *open list*. Starting from the start node, it only inserts neighbours of the examining node in the *open list*, combined with the first observation that f value of parent node is whether equal to or 2 smaller than its neighbours, there are at most two different kinds of f values in the *open list*, and if there are two various f values, they should differ in 2. When $f(n.children) - f(n) = 2$, nodes with f value "f(n.children)" would never be picked until all nodes with "f(n)" f values are removed from the *open list*. After that, there is no node with f value "f(n)" in the *open list*, nodes with "f(n.children)" as f value are examined. At this time, those nodes' neighbours could appear in the *open list*, but that is fine as there are only nodes with f value "f(n.children)" in the list, and their neighbours' f values are at most 2 bigger than "f(n.children)", there are still only two different types of f in the *open list*. When f(n) = f(n.children), A* could pick any node from the *open list* as all of those nodes have same f value. Therefore, whichever node is examined, its neighbours f value must equal to or 2 larger than "f(n)", which still meets the second observation. We also can observe it from our example implementation on Fig.13, Fig.14 and Fig.15, when the *open list* still has some nodes with "6" f value, the other f values it inserted are only "8". Only when all nodes with f value "6" have been removed from the *open list*, it begins to expand nodes with f value "8", and some "10" as f value exist in the list. Therefore, we can conclude that in the *open list*, it always has one or two different f values at the same time, which means the second observation is a truth too.

Getting some inspirations from the idea of multi-level bucket data structure, for the *open list*, we decide to put nodes with same f value in one bucket and based on our observations, at most two buckets are required in this new data structure. Even though A* has to find the smallest f value between buckets, there are at most 2 different values, finding the smaller one between two values is not expensive in any way. However, we prefer to build some connections between buckets, especially when we want to extend our limit from Manhattan distance to some other heuristic functions (more than 2 buckets). Considering when there are 2 more buckets, the most expensive operation should be finding the bucket with the smallest f value, we thought of the idea from the binary search. Therefore, we build our data structure based on min-heap, every bucket is regarded as a node on the heap.

3.2 Multi-Stack Heap

3.2.1 Data Structure of Multi-Stack Heap

Multi-stack heap is a data structure that built on a binary tree. However, every "node" on the tree is a stack which contains nodes with the same value, and we called this node container as stack node. Similar with min-heap, the value of each stack node is larger than or equal to its parent stack node, Check the Fig.16, the nodes $n1$, $n2$ and $n3$ in every stack have the same value and $v1$, $v2$, $v3$ ... indicate the value of nodes that stored in each stack node. At the same time, $v1 \leq v2$ and $v1$

$\leq v3$, similarly, $v2 \leq v4$ and $v2 \leq v5$, $v3 \leq v6$ and $v3 \leq v7$ and so on.

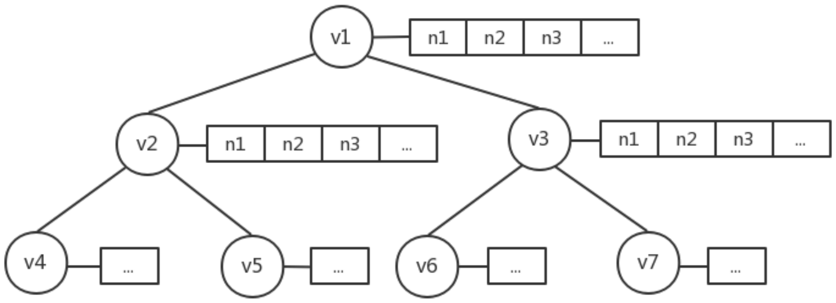

Fig. 16: Data Structure of Multi-Stack Heap

3.2.2 Operations of Multi-Stack Heap

The four main operations of this data structure were analysed as other data structures discussed in chapter two, which are insertNew, deleteMin, whetherContains and decreaseKey. The following explanation of the multi-stack heap is based on that we have k stack nodes on the heap, and n is the total number of nodes that stored on the heap, i.e., the summation of nodes from each stack node ($n \geq k$).

insertNew

For the insertion, check the value of this new node on the heap whether the heap contains this value, if it is, insert this node at the end of the found stack and done, this process takes $O(k)$. For example, in Fig.17, if we want to insert a node with value 8, then we need to find the stack whose value is 8, and then add the new node at the end of the stack (i.e., append after $n4$). Otherwise, create a new stack, push the new node in this stack, append the new stack at the end of the heap as the last leaf and set value of this stack as the value of the inserted node, comparing with value of its parent stack and swap positions of those two stacks if its values is less than its parent stack, do the same comparison until we do not move the new stack. This process is same as the "bubble up" of the min-heap, we call it "bubble up" in the multi-stack heap too. The bubble up process needs $O(log\ k)$, and checking the value on the heap takes $O(k)$. Therefore, it takes $O(k) + O(log\ k)$ to insert a new node if its value is not stored previously on the heap. Take structure in Fig.17 as an example again, when we want to insert a new node $n12$ with value 5, check whether its value is on the heap, and the result is not, then build a new stack and push in $n12$ like the top heap on Fig.18; after that, compare 5 with 10, 5 is smaller, swap their positions; compare 5 with 6 and swap positions of 5 and 6 again since 5 is still the smaller one; there is no parent stack of 5 as stack with value 5 is the root stack now, the task is completed and the final status of this multi-stack heap should be same as the bottom heap on the Fig18.

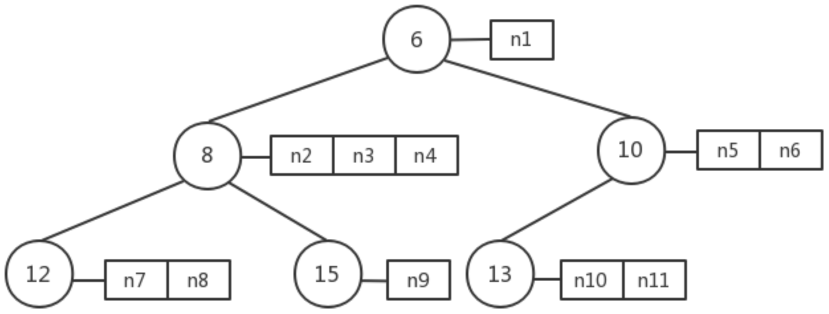

Fig. 17: A Specific Example of Multi-Stack Heap Structure

deleteMin

To delete the node with the smallest value, it could take $O(1)$ or $O(log\ k)$. As the topmost stack on the heap always contains nodes with the minimum value, deleting the minimum one is deleting one from this stack randomly. If the stack has more than one node, just leave the stack and the job is finished, this is the condition it needs $O(1)$. However, after the deletion, if the stack is empty, i.e., the deleted node is the last node in this stack, a bubble down is needed as min-heap. Firstly, delete the empty stack and put the last leaf stack from the heap at the root position, compare new root's value with its child stacks, swap position with the smaller child stack if its value its greater than one of its child stack or both of left and right stack. This is why it could take $O(log\ k)$ to delete the smallest element. For instance, the requirement

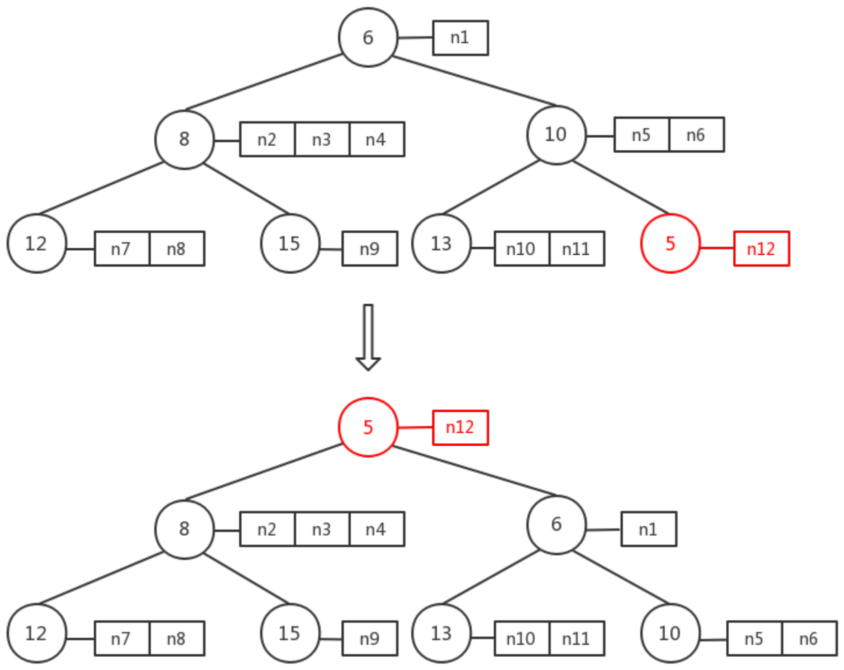

Fig. 18: An Example of Multi-Stack Heap Insertion based on Fig.17 (a new node whose value is not contained on the multi-stack heap)

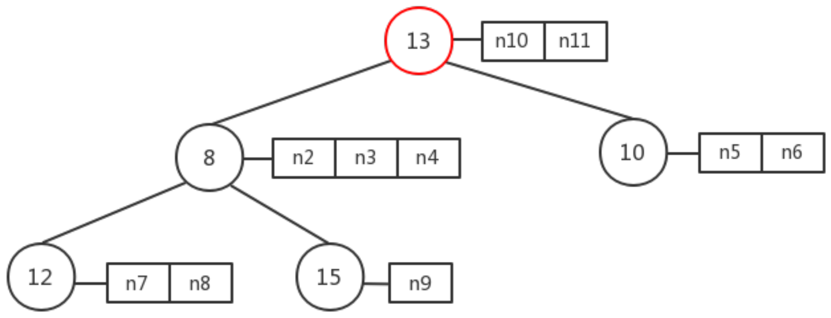

Fig. 19: An Example of Multi-Stack Heap Deletion based on Fig.17 (move the last leaf stack as the root of the heap)

is to delete the node with minimum value on Fig.17 now. As $n1$ is the only node in the top stack, after the deletion, the stack is empty. The first step is to remove the empty stack, move the last stack on the heap, the stack with value 13, to the root position as Fig.19; compare 13 with its children 8 and 10, 8 is the smallest one, change the positions of stacks with value 8 and 13; again, compare 13 with values

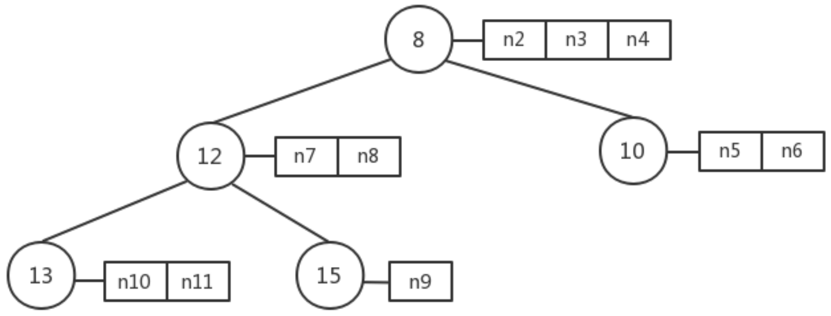

Fig. 20: An Example of Multi-Stack Heap Deletion based on Fig.17 (the final status after the deletion)

of its new child stacks, which are 12 and 15 respectively, 12 is the minimum one so swap 13 and 12, the mission is completed as there is no other child stack of 13, the ultimate heap should be as the multi-stack heap in Fig.20.

whetherContains

As we know, the operation whetherContains is to check whether the data structure contains the checking node. As nodes stored in the multi-stack heap are not only with "value" attribute, it probably has some other attributes such as a pointer to its predecessor, or the specific location of this node on a graph. The property "value" is the metric that we used to store nodes in the structure. Therefore, whetherContains in multi-stack means to check whether the checking node is already inserted on the heap, the value of inserting node contained on the heap is not used as the value could be changed. We have to identify whether two nodes are the same node by comparing their constant attributes such as a specific coordinate or an exclusive name that stored on the heap. The specific whetherContains process is shown in the following example: check whether the node $n3$ is on the multi-stack heap based on Fig.17. Suppose the name of each node is exclusive, i.e., the name "n1" is the specialised name for this node, if another node also with name "n1", they must be the same node. As we already know all information of this node before doing the check, to check whether the heap contains node $n3$, we need to scan from the bottom or top of the heap, and compare names of nodes in each stack to find one with name “n3”. In this example, a node with the name “n3” is detected in the left child of the root stack, so we can return the result that this multi-stack heap is already contained node $n3$. However, if the question is checking whether the heap from the Fig.17 contains node $n15$, then the answer should be negative after scanning the whole tree. whetherContains needs $O(n)$ to get the result.

decreaseKey

decreaseKey means updating information of a node which is already contained in the data structure, and the value of this node is changing to smaller than before. On the multi-stack heap, find the node that is already on the tree needs $O(n)$ (same with the operation of whetherContains). Next step is to update the information of the found node. As the node’s value is changed, it should be relocated at some other stack. Relocating process is the same as inserting a new node on the heap. Whether the changed value is already on the heap is being checked first, if the stack with the same value is found, move this node from the old location to this stack, put it at the end; otherwise, create a new stack, put the node in the stack, append the new stack at the end of the multi-stack heap and bubble up until the heap meets the requirement of a multi-stack heap. Compared to other operations, decreaseKey is expensive as it takes the time to do the whetherContains first and then do the operation of insertNew.

Summary

From the analysis of the four operations, we can see that the time complexity really depends on k, i.e., the number of stacks. When k is far smaller than the total number of nodes on the multi-stack heap n, which means that there are a large number of nodes with the same value stored in one stack, this data structure could be very efficient compared with the data structures discussed in the second chapter, such as the array, min-heap. However, when there are few nodes in each stack, k is very close to n, in this case, comparing with other data structures, the advantage of the multi-stack heap is not obvious. And when k equals to n, i.e., there is no node with duplicated value on the heap, every stack contains one node, then the multi-stack heap is same as the min-heap structure and time complexity of operations should be the same as the min-heap as well.

3.3 Multi-Stack Heap for A*

In A*, implementing multi-stack heap as the *open list*, for every node, the f value is stored as the value on the heap. Therefore, each stack contains nodes with same f value from the *open list* and the f value of nodes stored in the stack node is equal to or greater than the f value of nodes in its parent stack node as the feature of the multi-stack heap.

Combined the observations we obtained based on a square-grid graph using Manhattan heuristic with the data structure of multi-stack heap, there are at most 2 stacks on the heap as there are at most 2 different f values in the *open list* at the same time ($k = 2$). And when there are two stacks on the multi-stack heap, their f values must differ in 2.

3.3.1 A Pathfinding Case

To illustrate the detailed functions of multi-stack in A*, we set a pathfinding environment as Fig.21 (the initial status of the environment from Fig.13), and the multi-stack heap is performed as the *open list* in the whole solving process. On the graph, S is the start node while G is the goal node, black cells mean obstacles, every node could be represented as the coordinate (x, y) and the cost to pass a traversable cell is 1.

A* begins to search from the start location S and initially there is only S in the *open list*, therefore, the multi-stack heap only has one stack with one node $S(3, 6)$ stored in this stack. The next step is to explore the node S and move it to the *closed list*. Meanwhile, expand its neighbours $(3, 5)$, $(4, 6)$, $(3, 7)$ and $(2, 6)$ and calculate their f values respectively. Get those nodes' information as Table.2 and put the nodes in the *open list* separately. Before the insertNew operation, we need to check whether this neighbour is already stored in the *closed list* and *open list* (whetherContains), the answer is no, then begin to insert. Check the insertion process from the Fig.22, (a) creates new stack with f value 6 and node $(3, 5)$ is inserted in this stack; the f value of $(4, 6)$ is still 6, append it after $(3, 5)$ as (b); in (c), the algorithm builds a child stack as $(2, 6)$ has f value 8, which is not contained on the heap; (d) shows the node $(4, 6)$ is inserted after $(3, 7)$ as its value is also 8.

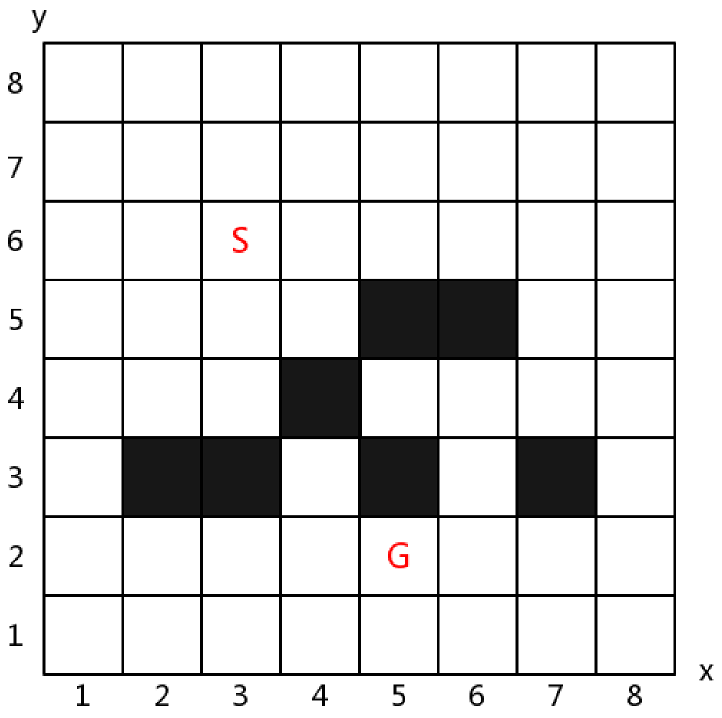

Fig. 21: An Example Pathfinding Environment

Neighbour Nodes	f Value	Parent Node
$(3, 5)$	6	$(3, 6)$
$(4, 6)$	6	$(3, 6)$
$(3, 7)$	8	$(3, 6)$
$(2, 6)$	8	$(3, 6)$

Table 2: Information of S Neighbour Nodes

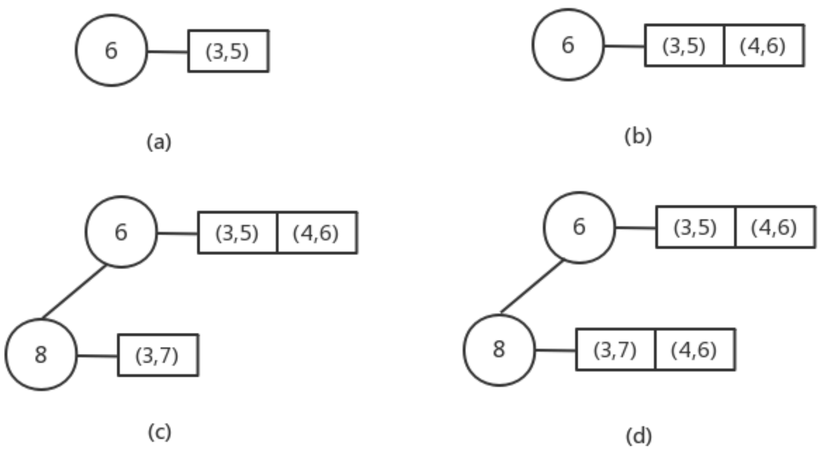

Fig. 22: Insert Neighbour Nodes of S on Multi-Stack Heap

In the next loop, A* has to find the node with minimum f value from the *open list* and move it to the *closed list* (deleteMin). Thus, we pick a random node from the root stack of the heap, for instance, we select node$(4,6)$ on Fig.22 (d), delete it and get the multi-stack heap as Fig.23 (a). After the deletion, the node$(3,5)$ is still in the stack, which is not empty so that we can just leave it and continue the next step. Expand neighbours of $(4,6)$ and get the information as Table.3.

Neighbour Nodes	f Value	Parent Node
$(4,5)$	6	$(4,6)$
$(5,6)$	6	$(4,6)$
$(4,7)$	8	$(4,6)$
$(3,6)$	8	*none*

Table 3: Information of node$(4,6)$ Neighbour Nodes

Same as the insertNew operation in the last loop, the result of whetherContains for node $(4,5)$ is negative so it can be inserted on the heap directly. As the f value of $(4,5)$ is 6, it is added at the end of the root stack (Fig.23 (b)). Node$(5,6)$ and $(4,7)$ are also new members on the heap, insert them as Fig.23 (c) and Fig.23 (d) as their

f values are 6 and 8. For the node$(3,6)$, when we checked the *closed list* and found that it is already stored in the *closed list*, so we skip this neighbour node and keep going... Do the same process until we find the exploring node is the G$(5,2)$.

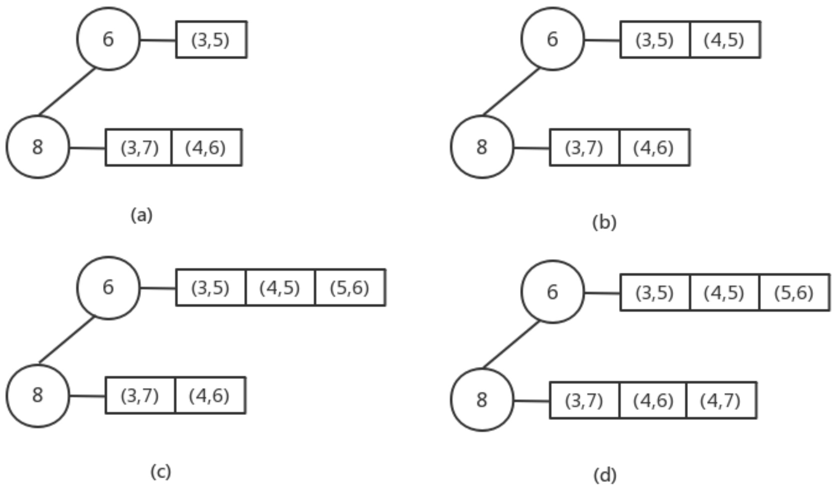

Fig. 23: Insert Neighbour Nodes of $(4,6)$ on Multi-Stack Heap

3.3.2 Operations Analysis

The agent is only allowed to walk in four directions on a squared-grid graph with Manhattan heuristic, take advantages of this, implementing the *open list* with the multi-stack heap could be much more efficient.

insertNew

When inserting the new member on the multi-stack heap, there are two conditions. One is that the f value of the inserting node is already on the heap, then we need to find the stack whose value is same as this node and add it at the end of the stack. As there are at most two stacks on the heap, finding the specific stack between one or two takes $O(1)$. Another condition is that a new stack has to be created for the inserting node when there is no same f value stored on the heap, then we need to append the new stack at the end at the heap with the inserting member, and bubble up this multi-stack heap. Since there are at most two stacks on the heap at the same time, if we need to create a new stack, there must be just one stack on the heap before we append the new stack. Therefore, bubble up the heap between two nodes also takes $O(1)$. As a result, insertNew in multi-stack heap takes $O(1)$.

deleteMin

Nodes with the smallest f value are always stored on the root stack of the multi-stack heap, therefore, to delete the minimum one, go to the top stack and pick one randomly to remove, which needs $O(1)$ to do it. After the deletion, if the stack is empty, and there are two stacks on the multi-stack heap at this time, then we delete this stack and put another stack as the root; if the empty stack is the only stack on the heap, and there is no neighbour node to insert in the *open list*, then the pathfinding is terminated as there is no path found. Therefore, it can be concluded that the deleteMin operation takes $O(1)$ to delete the node with minimum value on the multi-stack heap.

whetherContains

Before inserting a node on the heap, we have to check whether this node is contained in the *closed list*. As there are not many operations and cost of the *closed list*, the same data structure (normally, an array) is used for the *closed list* in different implementations, so that the operation of the *closed list* is not calculated and compared. The whetherContains means the operation to check whether the multi-stack heap already contains the inserting member on the *open list*. The f value of the checking node could be different with the same node that is already stored on the heap (if the f value of checking node is smaller, then we need to decreaseKey). Hence, in our indicated background, as there is other information of nodes stored on the heap, to check whether the two nodes are identical, we compare coordinates of nodes by scanning the whole multi-stack heap. Return to a positive result if found the two n-odes with the same coordinate. Therefore, similar with most of other data structures, decreaseKey needs $O(n)$.

decreaseKey

Locate the same node with larger f value that is stored on the multi-stack heap by operation whetherContains takes $O(n)$. After that, update the information of this duplicated node. As in the operation decreseKey, the f value of this node must be changed to smaller, and there are at most two stacks simultaneously, the updated node must be in the last stack of the heap (i.e., if there are two stacks on the heap, the node is stored at the child stack; if there is just one stack, the node is in this stack). Therefore, when do the whetherContains to find this node, we could start from the bottom of the multi-stack heap, which could same time to some extent in the implementation. When there are two stacks, as the f values of two stacks differ in 2, the decreased value must be 2 smaller than the f value the node has initially. Therefore, move this node from the child stack to the root stack to complete decreaseKey operation; when there is only one stack, then a new stack with 2 reduced f value is created, remove the decreased node from the old stack and add it in this new stack. As the f value of the new stack is smaller than the old stack, set the new stack as the root and the old stack as the child. From this description of decreaseKey, it can be observed that the main cost of this operation is still doing the whetherContains, update the information and heapify the tree takes approximately $O(1)$. Therefore, the operation decreaseKey needs time $O(n)$.

3.3.3 Optimization

The specific data structure of each "stack" on the multi-stack heap has not been explored yet. The "stack" here does not mean the data type stack with pop and push operations currently, it just indicates the specific "node" on the heap. The "node" is not simply a traditional node, but it is a container which can store numbers of real nodes. As nodes with same f values are stored in the same stack, the operations of each stack are picking a node randomly and deleting it from the stack, or inserting a new member at the end of the stack; those operations take the same time ($O(1)$) in different data structures. The most general and simple data structure to implement this stack is the array.

In the following part, we will do the pathfinding based on the Fig.24 while the array is implemented as each stack. On this graph, S and G are start node and goal node, and the alphabet in every cell is the name of this node, the cost to walk from one cell to its vertical or horizontal adjacent cell is 1.

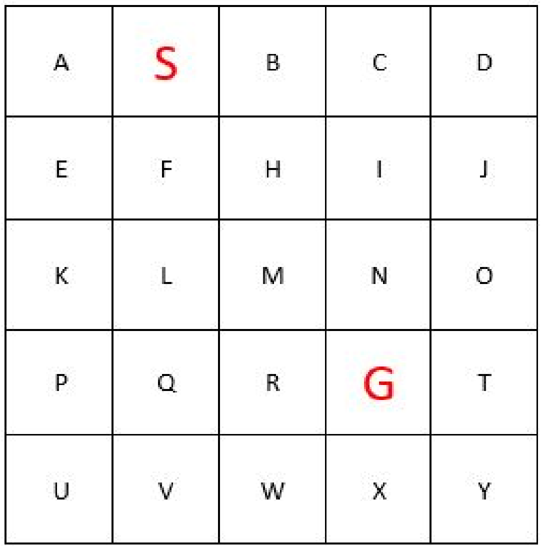

Fig. 24: An Example Graph with Start Node S and End Node G

Beginning from the start node, remove S and insert its neighbour nodes, we get the *open list* in multi-stack as Fig.25 (a). Next, we have to pick a node from the top stack of the heap to explore. All the nodes in this stack are with the smallest f value, ideally, we can select one randomly. For example, we always choose the first one from the stack (implemented in the array). Therefore, we remove node B from the *open list* and put it in the *closed list*, at the same time, expand its neighbour nodes C, H and S. S is already in the *closed list*, so we do nothing with this node. Then we get the multi-stack heap as Fig.24 (b). Similarly, we pick F from the *open list* as it is the first node in the root stack, remove it and add neighbours L and H as Fig.24 (c). After that, the node to delete from the *open list* is C and insert new members D and I in Fig.24 (d). The next exploring node is H, and after H, L will be selected, I is the next one after L and so on.

In this example, the new node is always inserted at the end of the stack and the

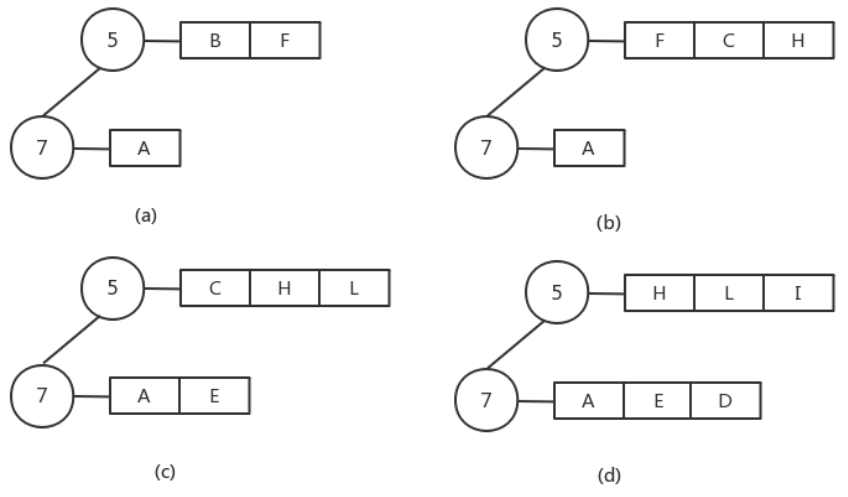

Fig. 25: Operations of the *open list* Based on Fig.24 with FIFO Rule

first node in the stack is always selected primarily, we can conclude that we use the First In First Out (FIFO) Rule to select the node from the *open list*. In this way, we will explore nodes in different directions, and all nodes on paths to G would be explored at the last when we reach the goal, which means, by FIFO rule, we will get all optimal paths from S to G. However, one optimal final solution is enough for our searching. There is no necessary to explore nodes on other final paths, which could result in a lot of redundant costs.

Considering the final path should be in one consistent direction, it would be more efficient if we can keep traversing direction successively. To achieve this, we found that Last In First Out rule could help (LIFO), i.e., pick the last node that inserted in the top stack on the multi-stack heap to explore first. For instance, in Fig.24, after explored S, we get its children B, F and A as Fig.26 (a). At the beginning node, it does not matter to choose which child to explore as walking from S to B and from S to F are same, as walk to either of them is walking in a consistent direction. Say we choose to walk to B, then B is removed on the heap and its neighbours C and H are inserted as Fig.26 (b). The next choice is important, as we want the agent to walk in a continuous direction, the neighbour of last explored node should be picked to keep the exploring path adjacent, i.e., C or H should be the next node to be explored, in this way, the agent can walk from S to B, to C or from S to B, to H, both of the paths satisfy the requirement of walking in a successive direction. As an explored node could have maximum four child nodes, to make the operation easier, we always pick the last child that put in the stack. So we select H as the next exploring node. Then H is deleted from the stack and its neighbours I and M are inserted at the end as Fig.26 (c). With the LIFO rule, we select M and move it to the *closed list*, add its children N, R and L on the Fig.26 (d). Equally, R is deleted and its child nodes G, W and Q are added in its relative stacks, then we get the multi-stack heap as Fig.26 (e). On this heap, G is the selected one to explore in the next step, but when check G, it is found as the end node, therefore, the searching is stopped as the path is found. And from the nodes in the *closed list* and their pointers to predecessors, we get the final path as S - B - H - M - R - G. And in this process, we explore very few redundant nodes.

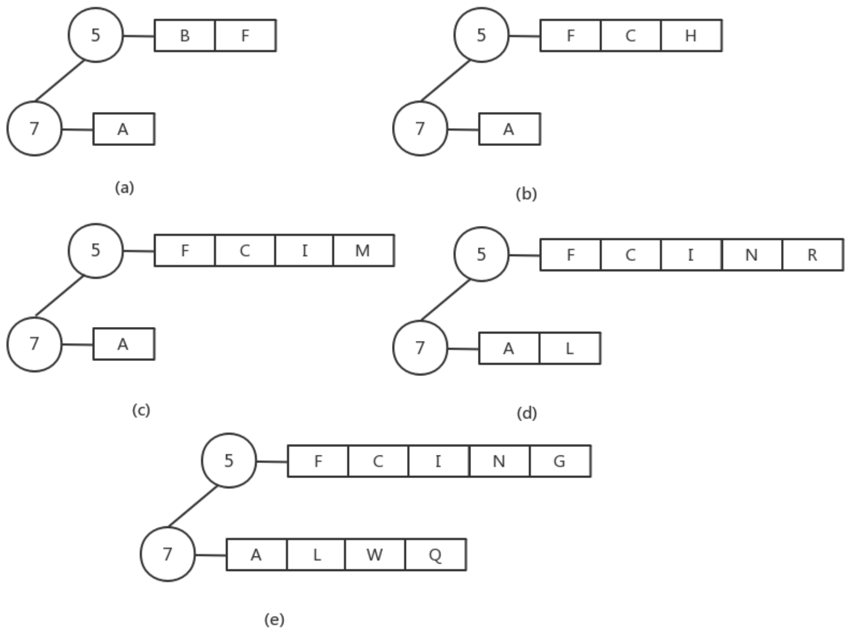

Fig. 26: Operations of the *open list* Based on Fig.24 with LIFO Rule

To make the exploring process more efficient, we found that using LIFO rule to pick a node among nodes with same minimum f value could save a lot, especially when there are at least two optimal paths. That is also the reason why we call our proposed data structure as the multi-stack heap, as data type stack with the property of Last In First Out rule, we name container of each level on the heap as “stack”. Even though we call it stack, it does not mean those containers need to be implemented in the real stack structure. It could be implemented in any data structures but when pick the node, we always pick from the end of this structure and when insert the node, also insert it at the tail of the structure.

3.3.4 Summary

To conclude, the discussion of the multi-stack heap with A*, we know that based on a square-grid graph with Manhattan heuristic, and the agent only moves in four directions (east, south, west, north), implementing the *open list* with multi-stack structure could be more efficient than the other data structures. Their time complexity is compared in Table.4.

Data Structures	insertNew	deleteMin	whetherContains	decreaseKey
Unsorted Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Min-Heap	$O(log\ n)$	$O(log\ n)$	$O(n)$	$O(n) + O(log\ n)$
Multi-Stack Heap	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Table 4: Time Complexity Comparison of Main Data Structures

The performance of the hash table and hot queue is not stable, the time complexity of the hash table depends on the collision and the hot queue relies heavily on the distribution of nodes in different buckets. Additionally, unsorted array and array performs very close. Therefore, we mainly compare and discuss the performance of the unsorted array, min-heap and multi-stack heap.

Table.4 shows clearly that whether the insert operation or deletion, multi-stack heap takes the least time between the three data structures. All three need $O(n)$ to check whether the node is contained in the data structure. And decreaseKey is based on whetherContains ($O(n)$), but min-heap takes another $O(log\ n)$ to complete it while unsorted array and multi-stack heap takes $O(n)$.

As a result, the performance of multi-stack heap is proved theoretically that, in the certain case, it could use $O(1)$ to insert and delete the minimum node from the *open list*, and it is more efficient than array and min-heap.

CHAPTER 4

*Two Ways to Implement A**

Before implementing A* in those different data structures, we propose two plan-s to perform A* algorithm regarding the operations “whetherContains” and “de-creaseKey”. From the description of A*, we know that when inserting a member in the *open list*, A* needs to check whether this member is already stored in the *open list*, which means it is required $O(n)$ to scan the *open list* every time before each insertion, this is expensive whichever data structures we will use. Therefore, we think of a different way to implement A* to save some time from the “whetherContains”. To make it easy to understand, we name the traditional method as Check From Open List, and the new method as Check From Closed List. The first is based on checking from the *open list*, and update the node if find the duplicated one; the second way is to ignore whether the inserting node is duplicated in the *open list* already, but check the duplication from the *closed list* when exploring nodes from the *open list*.

4.1 Check From Open List

The first one is the most typical method, checking from the *open list* to see whether the inserting node is already expanded in the list. In this way, A* needs to scan the *open list* before every insertion and this process takes $O(n)$. The implementation of A* in this plan is same as the description of Algorithm.1.

4.2 Check From Closed List

To decrease the time at checking the *open list* before every insertion, we come up with another method, check the *closed list* instead of checking the *open list*. When inserting a member in the *open list*, A* does nothing but inserts this node in the list, disregarding whether this node is already duplicated. However, when exploring the node with the smallest f value from the *open list*, we need to check the *closed list* to see whether it is already explored. If it is, then we delete this node from the *open list* and discard it, exploring the next node with the minimum f value in the *open list*; otherwise, move it from the *open list* to the *closed list*. If we find the exploring node is already stored in the *closed list*, and since we use admissible and consistent heuristic (no decrease-key operation in the *closed list*), that means this node with smaller or same f value has been explored already and put in the *closed list*, the exploring one is the duplicated node when insert it in the *open list*, so we just throw away this exploring node.

In this way, A* is not checking the *open list* before inserting, but check the *closed list* when exploring nodes from the *open list*. To lower the cost to check the *closed list*, we implement the *closed list* in a hash table. Therefore, we can save time from $O(n)$ to scan the *open list*, to $O(1)$ to check the *closed list*. More specifically, we skip the process from line 17 to line 24 in Algorithm.1, instead, only insert the node in the *open list* and set its predecessor from line 20 to 21 in Algorithm.2, at the same time, add the process to check the *closed list* when exploring every node from the *open list* from line 12 to line 14 in Algorithm.2.

However, there are some extra costs for this method when duplicated nodes appeared in the *open list*. Because we do not check the *open list* before the insertion, when a duplicated node is inserted in the *open list*, it takes extra time to operate the "insertNew" for this node. The specific cost depends on which data structure we use to implement the *open list*, for example, if we use min-heap to perform the *open list*, it could take extra $O(log\ n)$ to insert this duplicated node; if we use multi-stack heap or unsorted array, extra $O(1)$ could be used for this insertion. However, this extra

Algorithm 2 A* Algorithm: Check From Closed List

```
Input: A Graph $G(V, E)$ with start node $start$ and end node $goal$
Output: Least cost path from $start$ to $goal$
 1: Initialize:
 2:     $open\_list = \{start\}$
 3:     $closed\_list = \{\ \}$
 4:     $g(start) = 0$
 5:     $f(start) = heuristic\_function(start, goal)$
 6: while $open\_list$ is not empty do
 7:    $current$ = the node in $open\_list$ having the lowest $f$ value
 8:    if $current = goal$ then
 9:       return “Path found”
10:    end if
11:    $open\_list$.delete($current$)
12:    if $current$ already in $closed\_list$ then
13:       continue
14:    end if
15:    $closed\_list$.insert($current$)
16:    for each $neighbour$ of current do
17:       if $neighbour$ in $closed\_list$ then
18:          continue
19:       end if
20:       $open\_list$.insert($neighbour$)
21:       $neighbour$.setParent($current$)
22:    end for
23: end while
24: return “Path not found”
```

cost only occurs when duplicated nodes appear in the *open list*, which means we have to operate the “decreaseKey” in the *open list*. As we discussed in chapter two that “decreaseKey” rarely happens (which is also proved when we do the experiment), this cost could be very small. Therefore, the total cost for whetherContains could be reduced largely compared with the “Check From Open List” method.

4.3 Summary

The two implementation plans of A* have been described in Algorithm.1 and Algorithm.2, respectively. The most difference between the two methods is based on how to implement the “whetherContains” before the “insertNew” operation to the *open list*.

"Check From Open List" is the most typical implementation of A*, which scans the whole *open list* before every insertion. "Check From Closed List" takes account of admissible and consistent heuristic, it ignores the "whetherContains" before "insertNew", but check "whetherContains" in the *closed list* when the node is being explored from the *open list*. Since the *closed list* is implemented in a hash table, "whetherContains" in the *closed list* could be very efficient which only takes $O(1)$ for every check.

Theoretically, we can conclude from the above discussions that "Check From Open List" needs more time as it takes $O(n)$ to scan the *open list* before every insertion; "Check From Closed List" omits time to check the *open list*, but when the duplication happens in the *open list*, it requires extra cost for the insertion of this duplicated node.

CHAPTER 5

Experiments and Results

5.1 Implementation Specifics

Our testing environment is developed with Java in Eclipse IDE. The system simulates single agent pathfinding problem and is mainly designed for comparing the performance of different data structures of the *open set* in A* algorithm. The basic scenario is to give a start and a goal location randomly on a map, the agent's mission is to find an optimal path from the start node to the goal.

A* would be the search algorithm that implemented in the system. But different implementation plans of A* could be realized, they are "Check From Open List" and "Check From Closed List" that we described in Chapter four. To make the comparison as fair as possible, we always implement the *closed list* in a hash table no matter which data structures we use as the *open list* and which plans we adopt to implement A*.

The primary data structures in our comparisons are the unsorted array, min-heap and multi-stack heap. To lower the bias between comparisons, all of three data structures employ "Last In First Out Rule" to pick nodes from the *open list*. Additionally, for the implementation of the unsorted array, we use the data structure ArrayList; for the min-heap, we also adopt ArrayList to store elements; the mulit-stack heap needs a two-dimensional array to store nodes according to its data structure speciality, so an ArrayList nests another ArrayList are implemented while the inner ArrayList is used to store nodes as a stack and outer ArrayList is the container of those stacks.

5.2 Experimental Setup

5.2.1 Search Environment

As our new data structure is proposed based on the squared-grid map, we set the search environment on grid graphs as Fig.27. Each square cell on the graph weights 1. The agent is only allowed to walk to its adjacent cells while those cells are not obstacles. In our environment, adjacent cells of a node indicate its left cell, above cell, right cell and below cell, i.e., the agent can move vertically or horizontally in one cell at every step. Moreover, Manhattan heuristic is used to calculate the distance of any two cells.

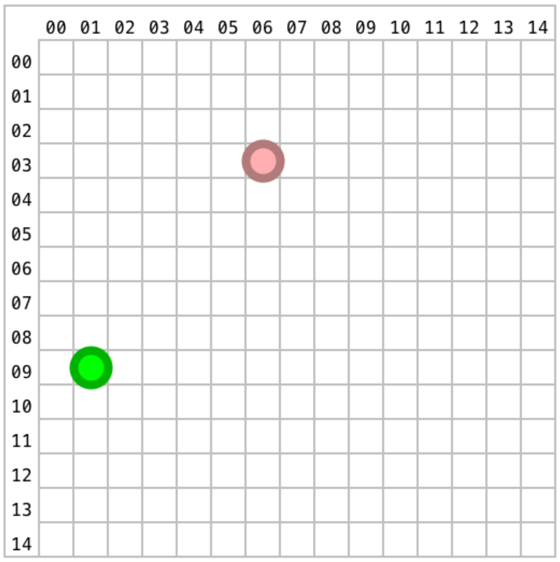

Fig. 27: Search Environment

In the experiment, different sizes of the search environment will be set such as 40x40, 80x80, 120x120, etc... We also did some research about the map size of some popular games, such as Baldur's Gate II and Warcraft III, their map size is usually scaled to 512x512. To make our experiment more close to the real game environment [22], we will test our map in the similar size too.

Obstacles on the map also take an important role in pathfinding. Different percentage of obstacles will be set on the graph to test the performance.

5.2.2 Search Parameters

Before doing the search on the map, there are several parameters to set. Firstly, we have to set the size of the map and obstacles distribution. The size could be set from 0x0 to 560x560. The percentage of obstacles could be generated on the graph from 0% to 40% as we found in the experiment that when the obstacles go up to over 40%, there is usually no path from the start node to the goal. Another argument we have to set is the implementation plans of A* we mentioned above, select one method among "Check From Open List" and "Check From Closed List" before do the pathfinding. After settle down of those parameters, we choose which data structures we are going to use as the *open list* for the search, unsorted array, min-heap or multi-stack heap.

The parameters could be set within different combinations, which depends on what performance result is expected. For example, when comparing the performance of different data structures, we set up the map size, obstacles, implementation plans of A* at first, then do the pathfinding with different data structures separately based on the previous setup. In contrast, to test which implementation of A* is the best choice, map size and obstacles, the certain data structure of the *open list* should be decided primarily, and then combine the setup with different implementation methods to do the pathfinding respectively.

5.2.3 Measurements

To compare the performance of different pathfinding search, searching time, final path length, the maximum size of the *open list* and *closed list*, and total number of different operations are collected in the experiment.

Time

Time refers to the duration from the beginning of the searching to the end, i.e., find the goal on the graph and return with a final path. With this measurement, the efficiency of the search could be shown intuitively.

However, as our system is built with Java, the execution time of a Java application is different from run to run. There are some factors such as JIT (Just-In-Time) compilation and optimization in the virtual machine driven by timer-based method sampling, thread scheduling, garbage collection, or some system effects, that affect the execution time significantly [8]. To lower the time error, before doing the test, we shut down all other applications that are running on the computer. Additionally, from our huge number of experiments, we obtain the rule of time changing according to the number of execution as Fig.28 (The parameters setup is kept the same in each execution). We found that in the same scenario, the first three execution times are usually very high, but in the later executions, time of each execution is very close. Therefore, to make our result of execution time more precise, we get time by executing the pathfinding 15 times, ignoring the first 5 execution time and getting the average of remaining 10 as our final execution time. After the first 5 runs, the variance was insignificant.

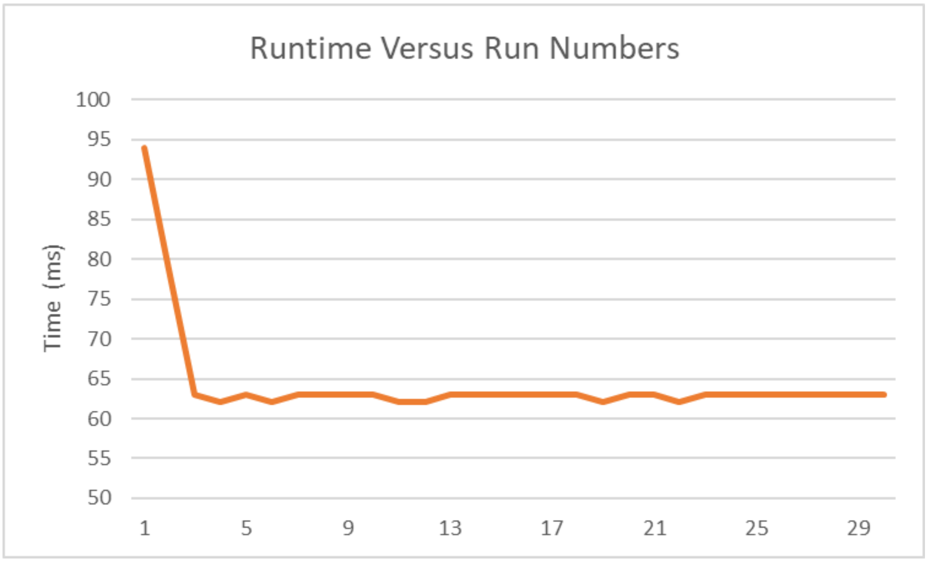

Fig. 28: Execution Time Versus to Search Times

Path Length

As we use admissible and consistent heuristic, the final path must be optimal (we have proved this in chapter 1.4.3). As long as we do the pathfinding on the same graph, no matter which implementation plans of A* is selected and data structures we use for the *open list*, the length of the final path should be the same. There is no necessity to set the length of the final path as a measurement for comparing performance as they are same on the same map. However, it can be used as the verification to make sure the path we attain is a correct final solution with different implementation plans and data structures.

Maximum Size of Lists

During the searching, the size of the *open list* and the *closed list* is always changing as we keep inserting and deleting nodes from the *open list*, and keep adding nodes in the *closed list*. Therefore, we record the maximum size of them and regard it as the memory requirement in every pathfinding.

Number of Operations

Even though we track the time to show the efficiency of each search, considering the time is not high accurate, the number of different operations is also being counted as a measurement. During the process, we count the times of operating the "insertNew", "deleteMin", "whetherContains" and "decreaseKey". In this way, we can ignore the extra cost of implementations that may affect the final performance. However, the time complexity of those operations is different in various data structures. For instance, the "insertNew" takes $O(1)$ in the unsorted array as it only needs to insert the new node at the end of the data structure while in the min-heap, $O(log\ n)$ is required to heapify the tree after insertion. As a result, we track the number of steps of every operation within different data structures as follows:

For the unsorted array, the "insertNew" is inserting the new node at the end of the array, so we count 1 step for every "insertNew". When deleting the node with the minimum f value, it has to find the smallest value from an unsorted array. Therefore, every comparison to find a smaller one happens in each "deleteMin" is counted as 1 step. As for the whetherContains, we count the number of nodes it compares with the target node until it finds the duplicated member. The "decreaseKey" occurs after the "whetherContains", only when we find the duplicated node in the *open list*, the "decreaseKey" is triggered. However, as the data structure is the unsorted array, even the key has been decreased, no operations are needed to maintain the data structure.

It takes $O(log\ n)$ to "insertNew" and "deleteMin" when we use the min-heap as the *open list* as the tree has to be heapified after the insertion and deletion. To heapify the tree, the basic step is to compare the node with its children and parent nodes to locate their positions properly. Therefore, we count the number of comparisons as the costs of "InsertNew" and "deleteMin" in the min-heap. For the "whetherContains", it also has to scan the whole *open list*, so the steps of comparing with nodes are counted as the "whetherContains". If the "decreaseKey" is needed, that means the heapify operation is required again, therefore, the comparisons are counted again.

Though the multi-stack heap only takes $O(1)$ in the "insertNew" and "deleteMin", there is more than 1 step in both two operations. Before the insertion, which stack the new node is going to insert in, needs to make a decision. This refers to finding which stack contains the same f value as the inserting node. Therefore, before inserting a node, comparing the f value between stacks and the new member is counted as well though there are at most two different stacks at the same time. For the "deleteMin", if the top stack is not empty after the deletion, then it just takes 1 step; however, when the deleted node is the only node in the top stack, then we need to heapify the heap, that cost is counted as well. Similar with unsorted array and min-heap, the "whetherContains" in the multi-stack also needs to scan the whole *open list*, therefore, the number of scanned nodes is counted. When the "decreaseKey" is triggered, as there are at most two different stacks on the heap at the same time, if the decreasing node is on the top stack, that means there is only one stack on the heap and as the new key must be smaller, we have to create a new stack and make it as the top key; if the node is on on the second stack, we just need to move this node to the end of the top stack. The steps of those process are counted as the cost of the "decreaseKey".

In addition to the "insertNew", "deleteMin", "whetherContains" and "decreaseKey", there are other costs in each search: operations of the *closed list*. The two main operations of the *closed list* are checking whether the inserting node is contained in the *closed list* and inserting the node in the *closed list*. Therefore, every check and insertion are counted in each search as the cost.

As a result, the total number of steps is calculated as the total cost of each search. There is no variance in the number of operations as for the same graph and data structures, each run requires same number of operations. We use it as a measurement to compare the performance of each pathfinding search.

5.3 Result and Analysis

Even with a certain size and obstacle percentage, we can generate a lot of different maps, and final paths on those map could be dissimilar significantly. Therefore, the length of final paths from different maps are different, and the length from a 520x520 sized map could be shorter than the path on the map scaled in 360x360. Though, with the combination of each size and obstacle density, we generated the map randomly within a situation that as complicated as possible to test our data structures and implementation plans to the greatest extent. For example, when there are no obstacles, we set the start node and end node as far as possible as the map size

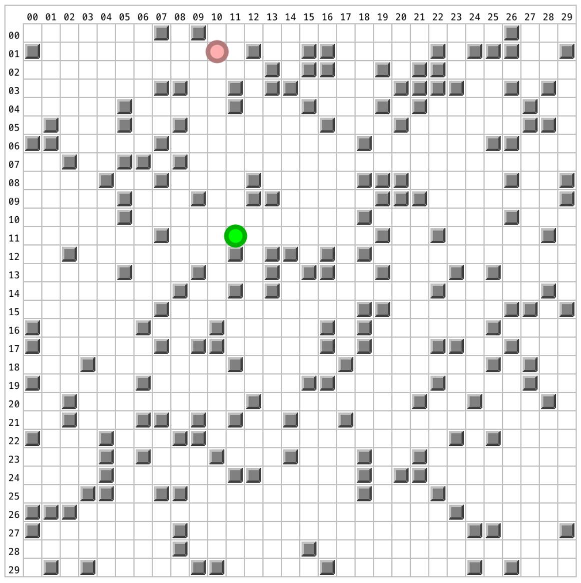
Fig. 29: Map 1 with size 30x30 and 20% Obstacles

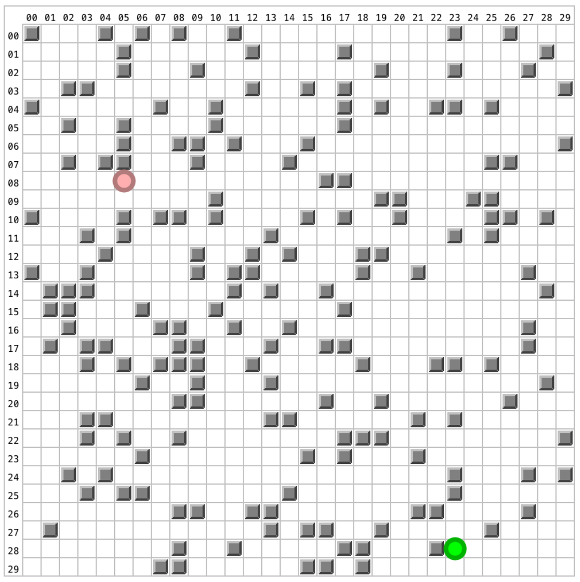
Fig. 30: Map 2 with size 30x30 and 20% Obstacles

goes up; with a certain percentage of obstacles, we preferred to the map with more obstacles on the path from the start to the end node. Take Fig.29 and Fig.30 for instance, both of the two maps are generated with size 30x30 and 20% obstacles, we would rather choose Fig.30 as our experiment graph since the obstacles distribution on this map from the start node to the end node is more complicated than the map in Fig.29.

We generated 126 maps with different sizes and obstacle densities. On each map, we did the pathfinding six runs by A* algorithm with the three data structures and two implementation plans, respectively. The combinations of the data structures and implementation plans are selected as Table.5. The results are shown in the following five aspects: runtime, number of operations, memory usage, obstacle density and final solution path.

Implementation Plans	Data Structures of *Open List*
Check From Open List	Unsorted Array
	Min-Heap
	Multi-Stack Heap
Check From Closed List	Unsorted Array
	Min-Heap
	Multi-Stack Heap

Table 5: Combination of Experiments

5.3.1 Runtime

When there was no obstacle generated on the test maps, the paths and time are shown on Fig.31 and Fig.32. On each chart, the running time when each data structure that used as the *open list* are shown by “Check From Open List” and “Check From Closed List”, respectively. Additionally, the length of the path has been recorded as as a reference.

The charts obviously show the result that in each test environment, when the multi-stack heap was implemented as the *open list*, it always used less time than the data structures unsorted array and min-heap whether A* was implemented by “Check From Open List” plan or “Check From Closed List” plan. Moreover, comparing “Check From Open List” and “Check From Closed List” implementation plans, with the same data structure, “Check From Open List” took more time than “Check From Closed List”.

In most cases,when the obstacles density went bigger and longer path were selected, the multi-stack heap often needed the least time, and the min-heap was after the multi-stack while the unsorted array usually took the longest time to do the search; “Chech From Closed List” often more efficient than “Check From Open List”. We list the results of map 40x40 with different density of obstacles as Fig.33 for an example. In each chart on Fig.33, whether in “Check From Open List” or in “Check From Closed List”, the runtime of unsorted array often stood higher than the min-heap and multi-stack heap and the runtime of the multi-stack heap was always the smallest one. And when it came to the same data structure, runtime in “Check From Closed List” often lower than the runtime in “Check From Open List”.

However, there were some special case shown that the min-heap required more time than the unsorted array; unsorted array and min-heap performed better with “Check From Open List” than with “Check From Closed List”. Take map 160x160 as an example, we collected the results as Fig.34.

On Fig.34, when the obstacles density is 10%, the min-heap required the longest time with “Check From Open List”; when the percentage of obstacles were 15%, 20% and 25%, the unsorted array in “Check From Closed List” took longer time than in “Check From Open List”. We found that the performance of different data structures is also relevant with the operation “decreaseKey”. The trigger times of “decreaseKey” in those cases are concluded in the following tables. From Table.6, we can see that when the min-heap triggered more “decreaseKey” operations, it would take more time than the unsorted array. We observed that when the map with obstacles density 15%, 20% and 25%, unsorted array performed better with “Check From Open List”. Combining with Table.7, Table.8 and Table.9, the number of “decreaseKey” operations is pretty big for the unsorted array.

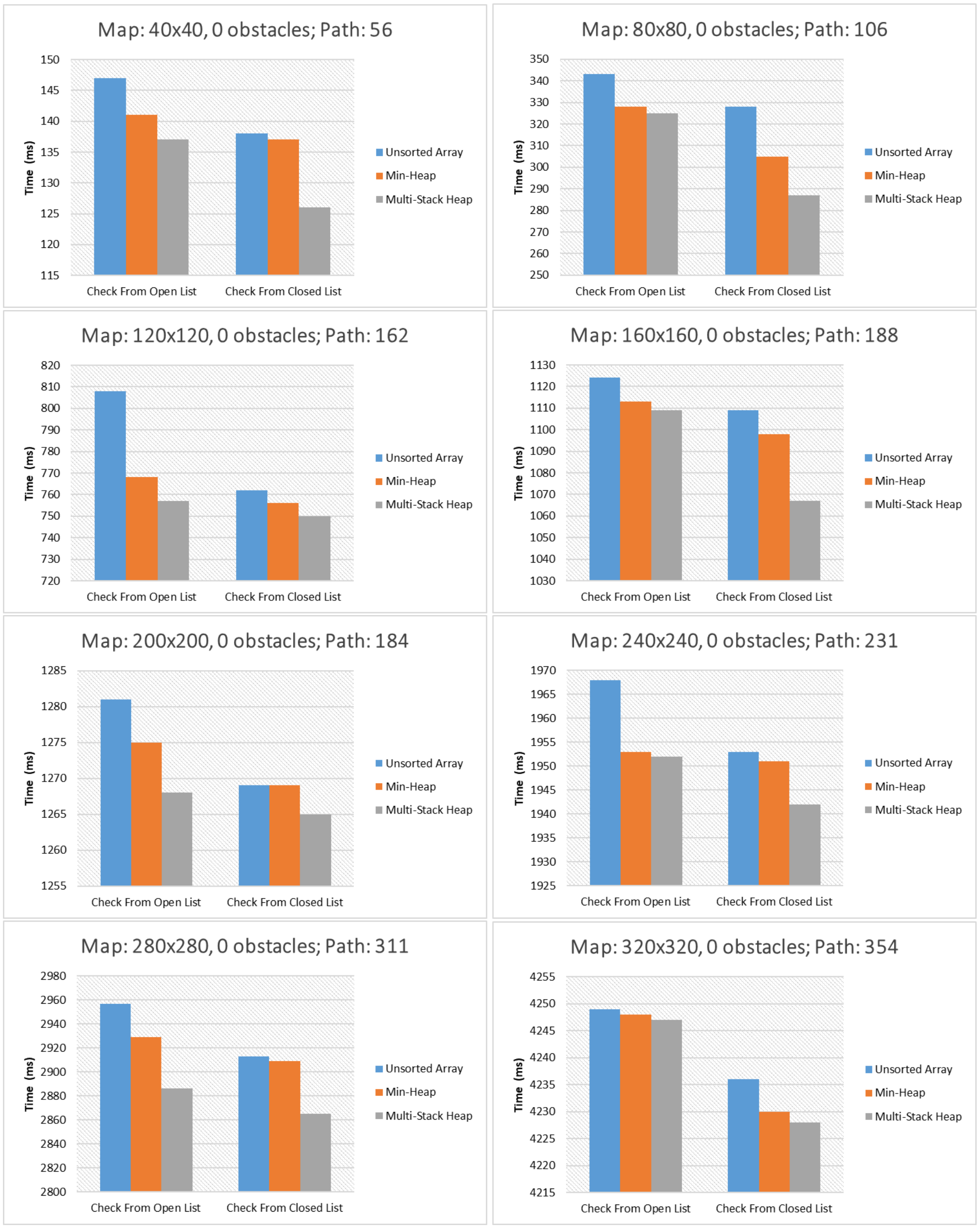

Fig. 31: Runtime of Environments From Size 40x40 to 320x320 Without Obstacles

Therefore, we can conclude from the runtime measurement that in the most cases, the multi-stack heap cost the shortest time, while the runtime of the min-heap was sorted at the second position, the unsorted array often took the longest time.

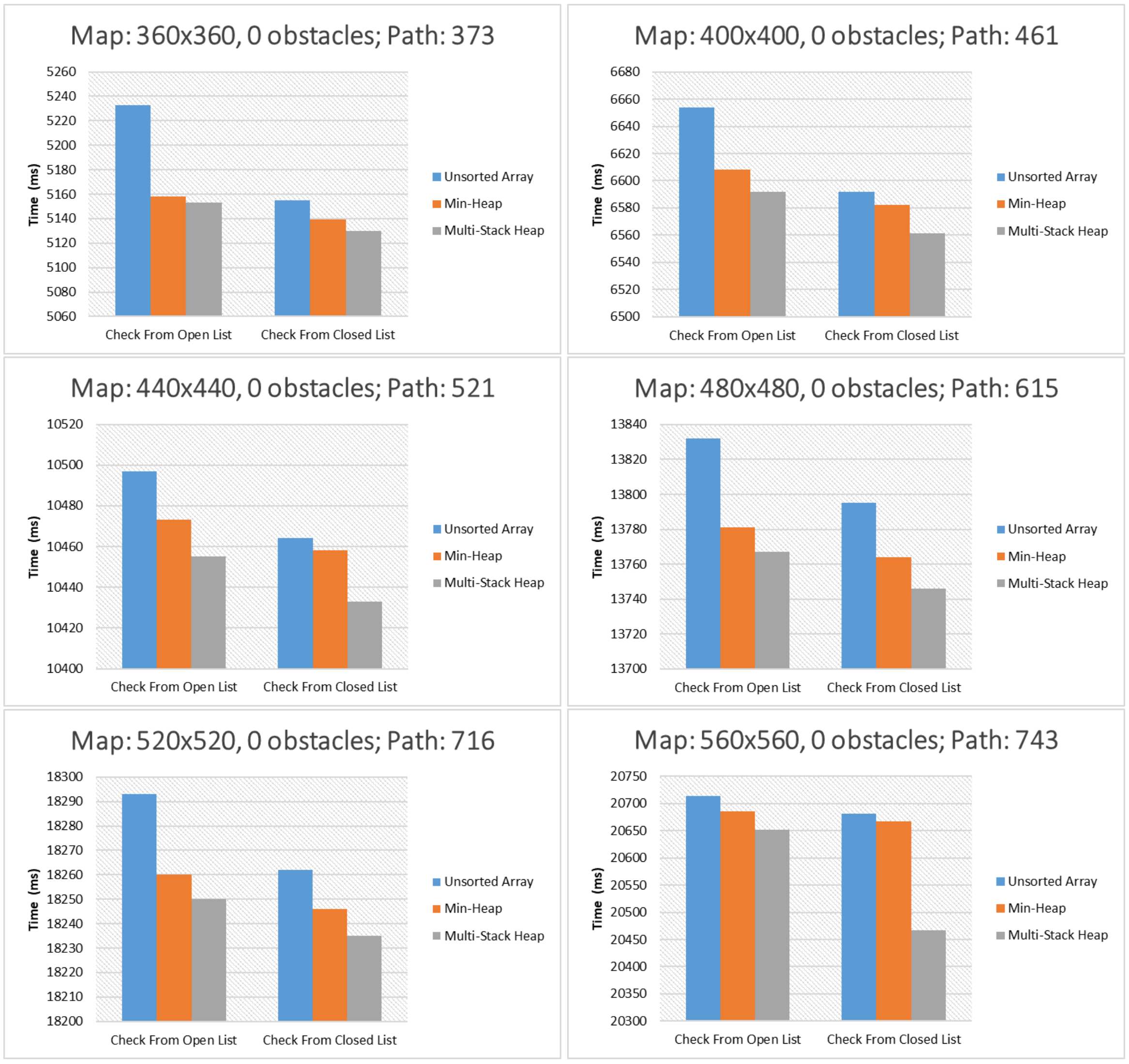

Fig. 32: Runtime of Environments From Size 340x340 to 560x560 Without Obstacles

Data Structures	Obstacles Density	Path Length	decreaseKey	Time
Unsorted Array	10%	170	37	1875
Min-Heap	10%	170	43	2390
Multi-Stack Heap	10%	170	44	1781

Table 6: Number of decreaseKey on 160x160 Map With 10% Obstacles

However, the performance of the unsorted array and min-heap was also relevant with the frequency of "decreaseKey" operations. In another word, it also depended on

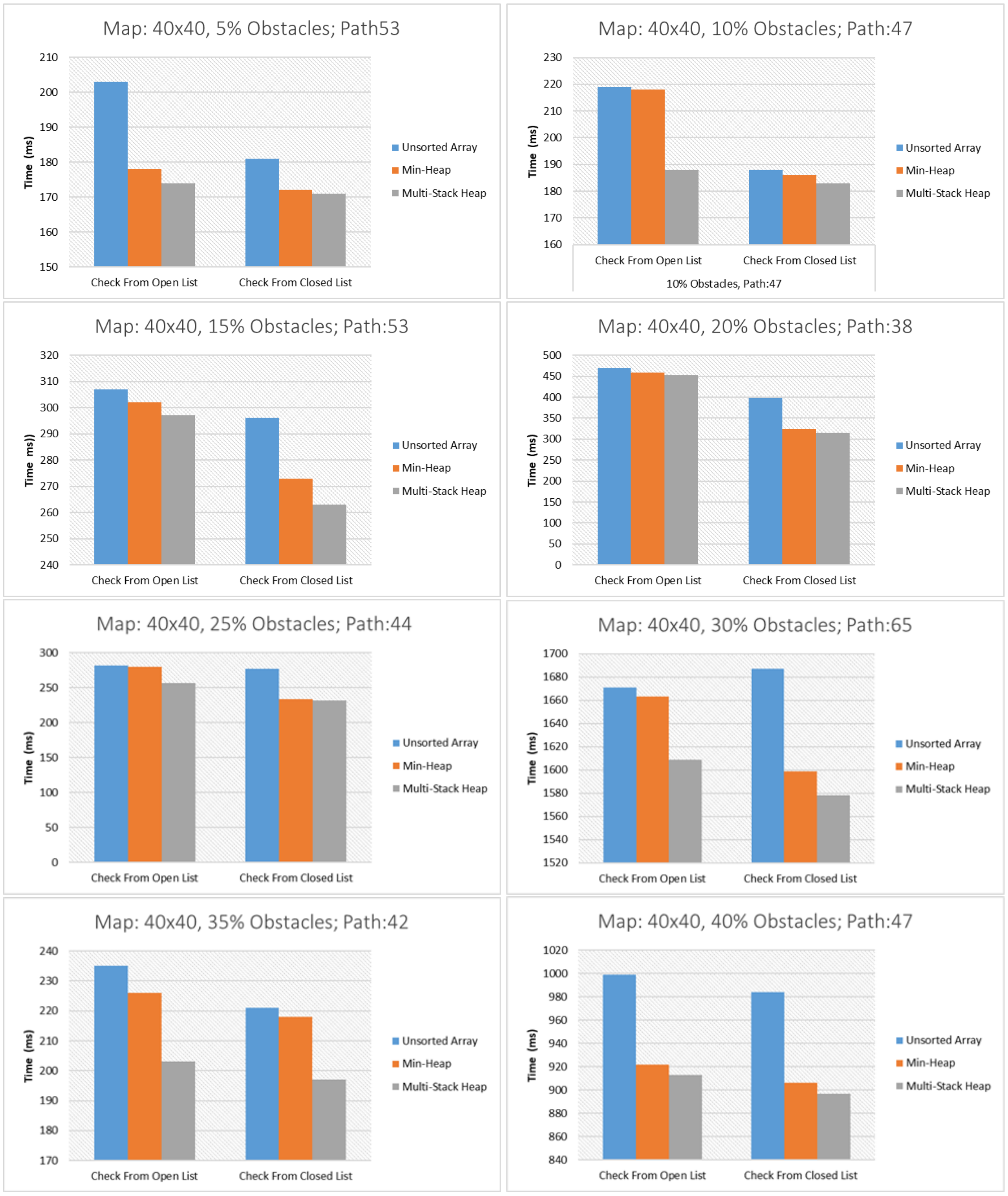

Fig. 33: Runtime of 40x40 Map With Obstacles (5% - 40%)

the distribution of obstacles. However, the multi-stack was not affected, it always performed best among the three data structures, and the runtime was shorter when it changed from “Check From Open list” to “Check From Closed List”.

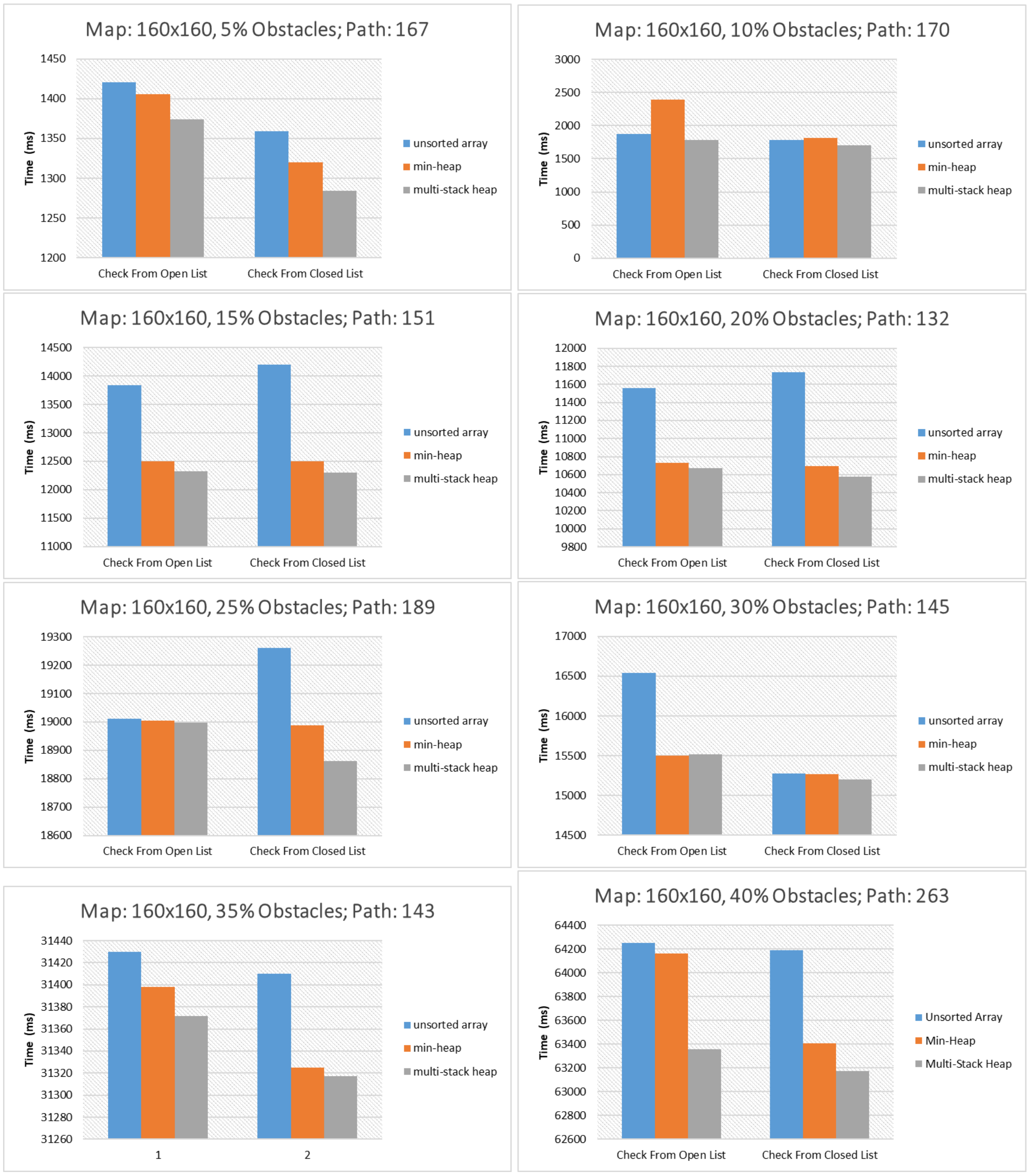

Fig. 34: Runtime of 160x160 Map With Obstacles (5% - 40%)

5.3.2 Number of Operations

We can see that most of the runtime were still quite close to each other when different data structures were adopted in the same implementation plan of A*. The reason is that there were a bunch of side effects could affect the runtime of each execution, which we have discussed in the measurement part. Also, even a slight difference in

Data Structures	Obstacles Density	Path Length	decreaseKey	Time
Unsorted Array	15%	151	580	13840
Min-Heap	15%	151	170	12497
Multi-Stack Heap	15%	151	659	12325

Table 7: Number of decreaseKey on 160x160 Map With 15% Obstacles

Data Structures	Obstacles Density	Path Length	decreaseKey	Time
Unsorted Array	20%	132	449	11560
Min-Heap	20%	132	245	10732
Multi-Stack Heap	20%	132	556	10671

Table 8: Number of decreaseKey on 160x160 Map With 20% Obstacles

Data Structures	Obstacles Density	Path Length	decreaseKey	Time
Unsorted Array	25%	189	550	19011
Min-Heap	25%	189	304	19005
Multi-Stack Heap	25%	189	623	18997

Table 9: Number of decreaseKey on 160x160 Map With 25% Obstacles

implementing the three data structures could result in a big distinction of the runtime. The counting of total operations triggered in the searching could be more obvious. As there are too many experiment sets, we list the same text environments as we did in the runtime section for a better comparison.

On Fig.35 and Fig.36, the numbers of total operations of every data structures and implementation plans were different significantly in each chart when there were no obstacles in the testing environment. When implemented A* algorithm with "Check From Open List", the unsorted array executed more operations than the min-heap and the multi-stack heap. The operations number of those two data structures were closer than the unsorted array while the multi-stack heap had the smallest number of operations.

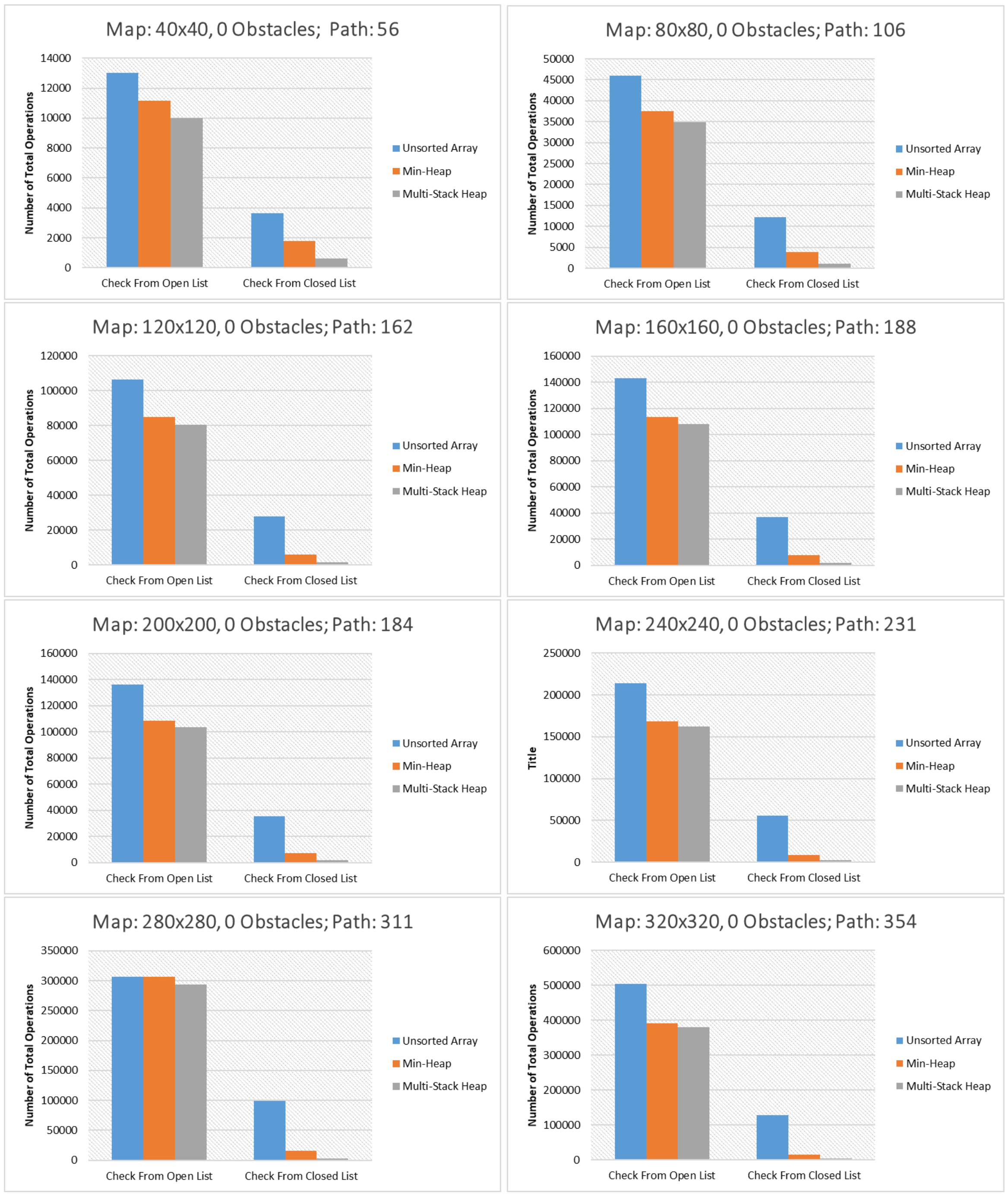

Fig. 35: Operations of Environments From Size 40x40 to 320x320 Without Obstacles

When we used “Check From Closed List” implementation method, the efficiency of each data structure was improved significantly. Especially the multi-stack heap, it executed very few operations compared with the unsorted array and the min-heap with “Check From Closed List”.

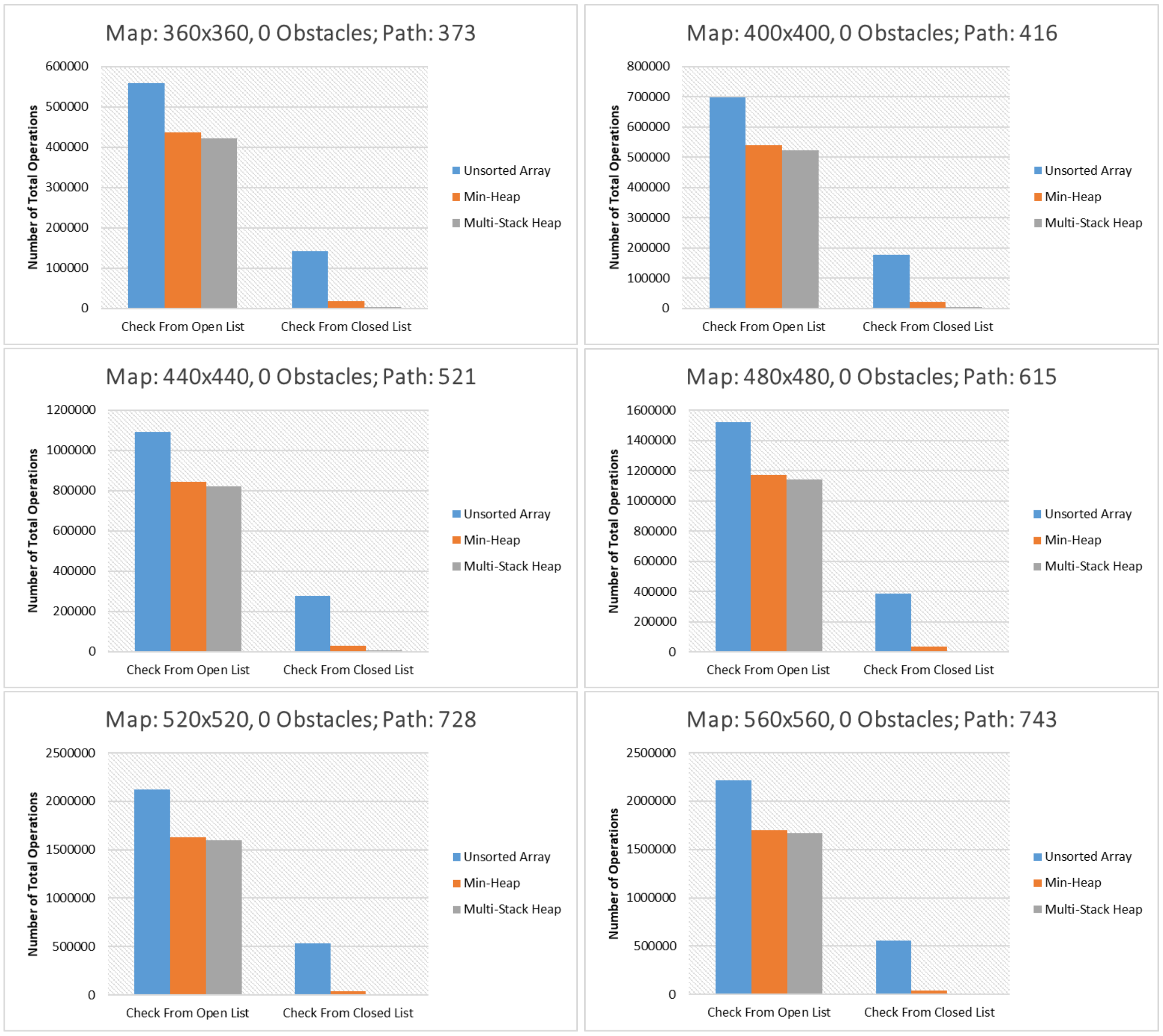

Fig. 36: Operations of Environments From Size 340x340 to 560x560 Without Obstacles

However, when there were obstacles, the performance was affected when some "decreaseKey" operations happened, similar with what we discussed about the time. By analysing the runtime of 160x160 map with obstacles from 5% to 40%, we knew that there were several special cases. When the percentage of obstacles was 10%, the min-heap took more time than the unsorted array. Regarding the number of operations on Fig.37, when the obstacles density was 10%, the min-heap also executed more operations, near 10,000 more operations than the unsorted array.

Additionally, the runtime of the unsorted array was smaller with "Check From Open List" than with "Check From Closed List" when the percentages of obstacles

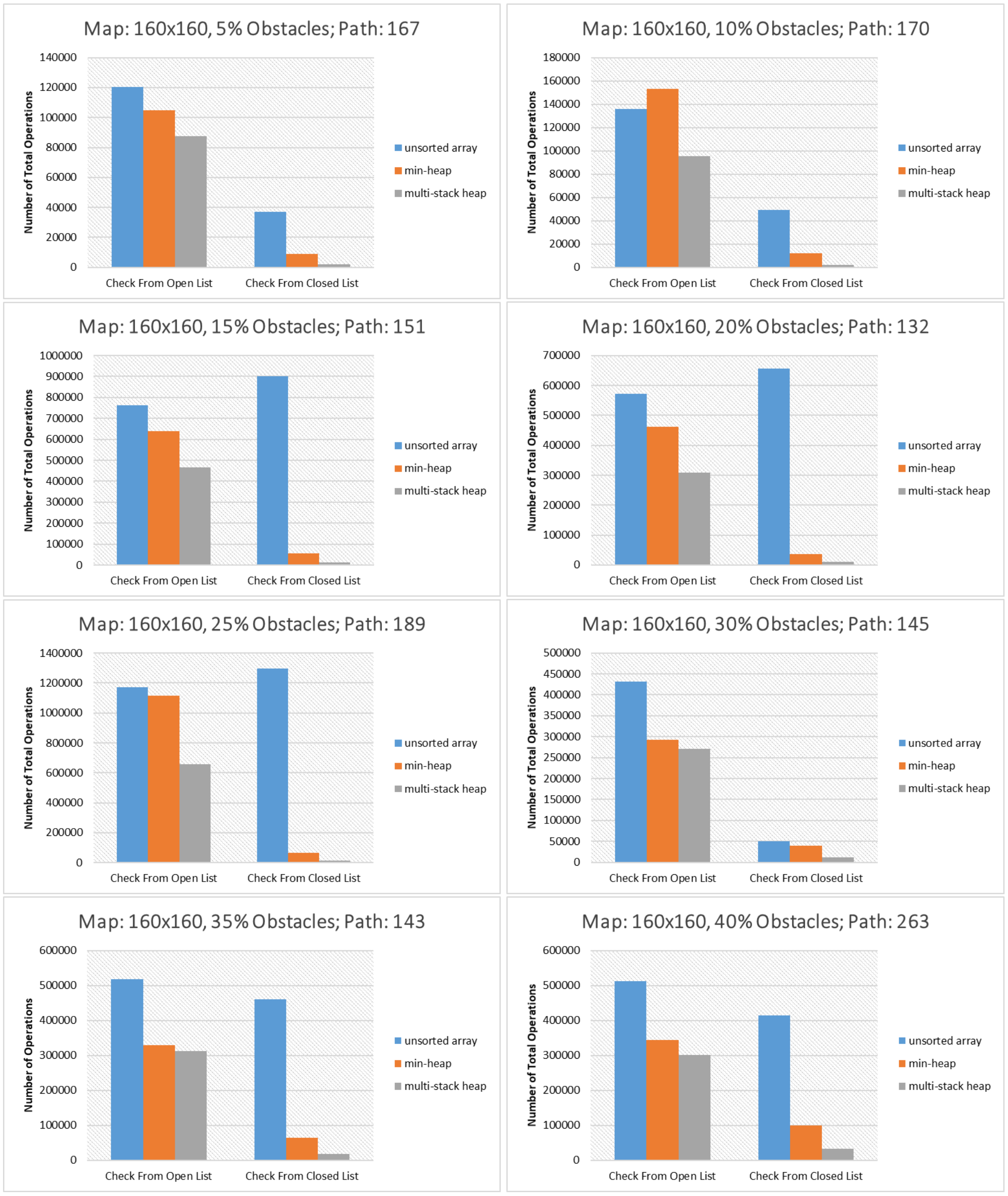

Fig. 37: Operations of 160x160 Map With Obstacles (5% - 40%)

were 10%, 15% and 25%. Corresponding to the number of operations on Fi.g.37, for the unsorted array, the number of its operations than with "Check From Closed List" executed more operations with "Check From Open List" when the obstacles were 15%. Similar with the test environments who had 20% and 25% obstacles, more operations were processed by the unsorted array with “Check From Closed List”. The counts of “decreaseKey” operations were shown in Table.6, Table.7, Table.8 and Table.9. The decreaseKey operations also affected the number of operations.

In conclusion, for the number of total operations, we can summarize as: in most cases, the multi-stack heap executed the least number of operations, the min-heap was the next one, and the unsorted array had the largest number of operations. But in some special cases, especially when “decreaseKey” happened very frequently, it had the possibility that the min-heap triggered more operations than the unsorted array. And when there was a large number of “decreaseKey”, it could make the unsorted array have more operations with “Check From Closed List” than with “Check From Open List”. However, the multi-stack always had the smallest number of operations compared with the unsorted array and the min-heap. When implemented A* with “Check From Closed List”, its number of operations had been lowered significantly.

5.3.3 Memory

We summed the maximum sizes of the *open list* and the *closed list* as the memory requirement in the pathfinding. When there was no obstacle on the testing environment, the three data structures used the same memory within the same implementation plan. Even implemented A* from “Check From Open List” to “Check From Closed List”, there was only 1 difference on the size of the *open list* and the *closed list*. The results has been shown clearly on Fig.38 and Fig.39. The reason is that, when there were no obstacles and all three data structures used “Last In First Out” Rule, all of them found the final path directly without traversing any redundant nodes. As the final paths of each data structures were in the same length, their memory requirements were same or super close. They might have 1 difference on the memory sometimes, that depended on which path the data structure selected as a great number of optimal paths existing at the same time when there was no obstacle. For example, the left and right graphs on Fig.40 show two different optimal paths on the same map. The two paths are in the same length, and no redundant node is not on the path, so that the sizes of their *closed list* are same; but one more node is expanded in the *open list* on the right graph, thus their sizes of the *open list* differ in 1.

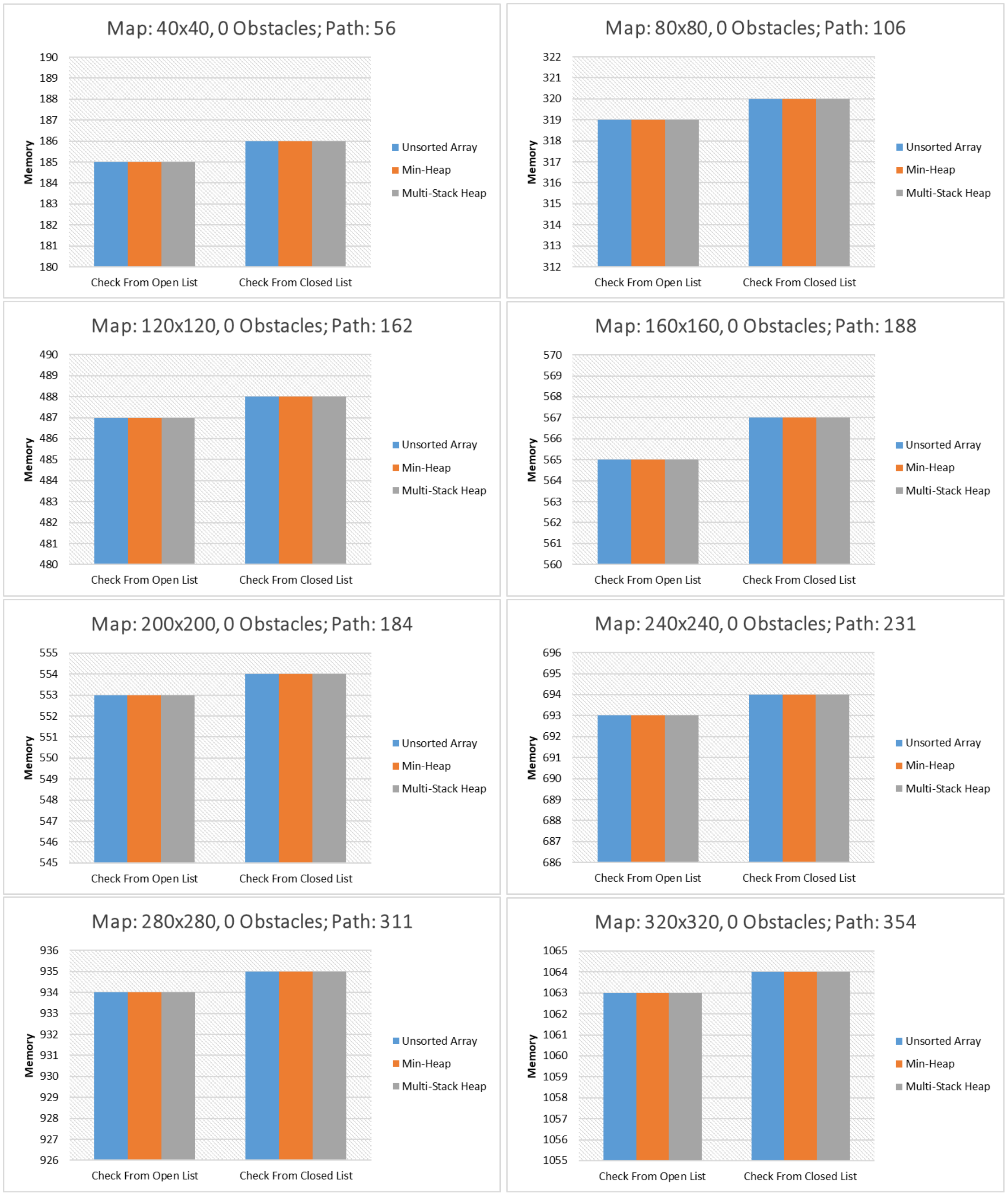

Fig. 38: Memory of Environments From Size 40x40 to 320x320 Without Obstacles

When there were obstacles, the memory requirements were different. The difference was slightly when the map size was 40x40, because the path selected on the map

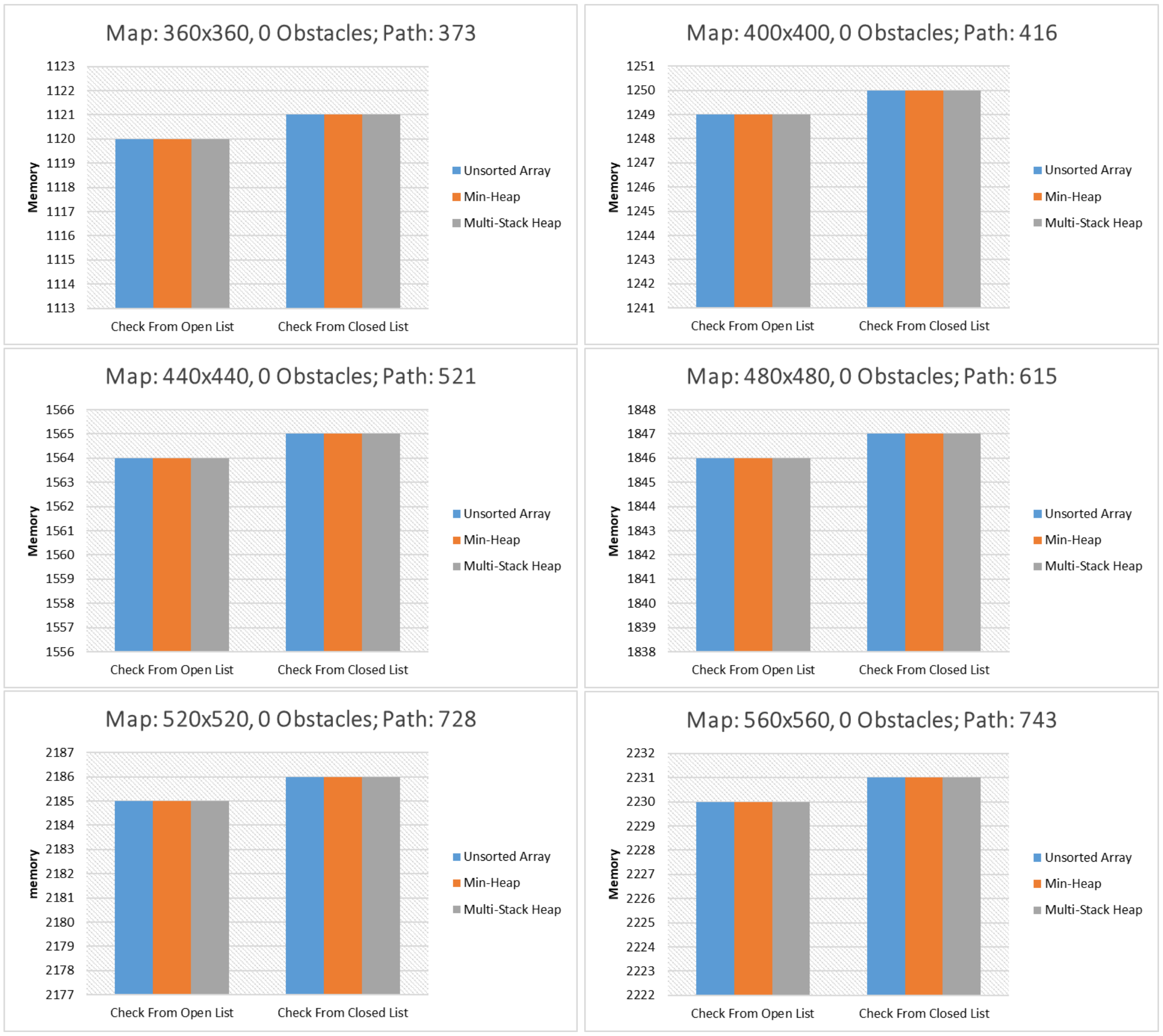

Fig. 39: Memory of Environments From Size 340x340 to 560x560 Without Obstacles

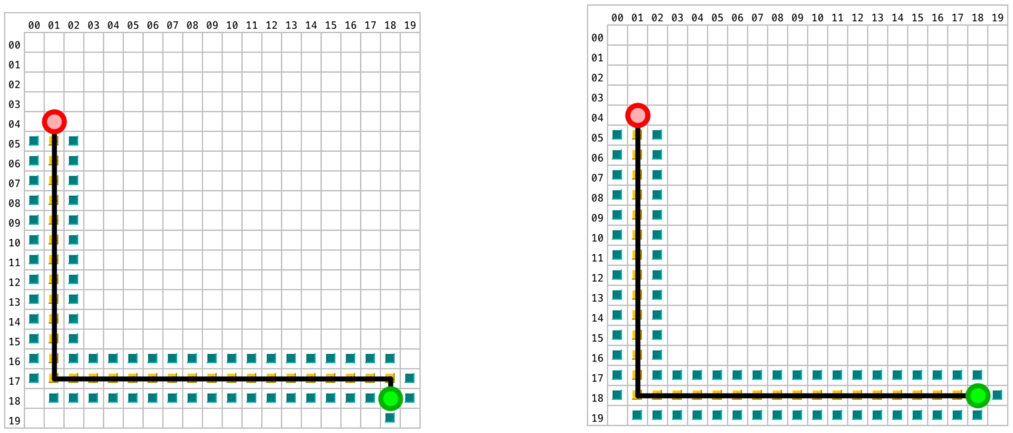

Fig. 40: Pathfinding In Test Environment with No Obstacles

was short, the difference between memory requirements was not obvious. Therefore, we selected a bigger environment sets 160x160 and 520x520 for an example.

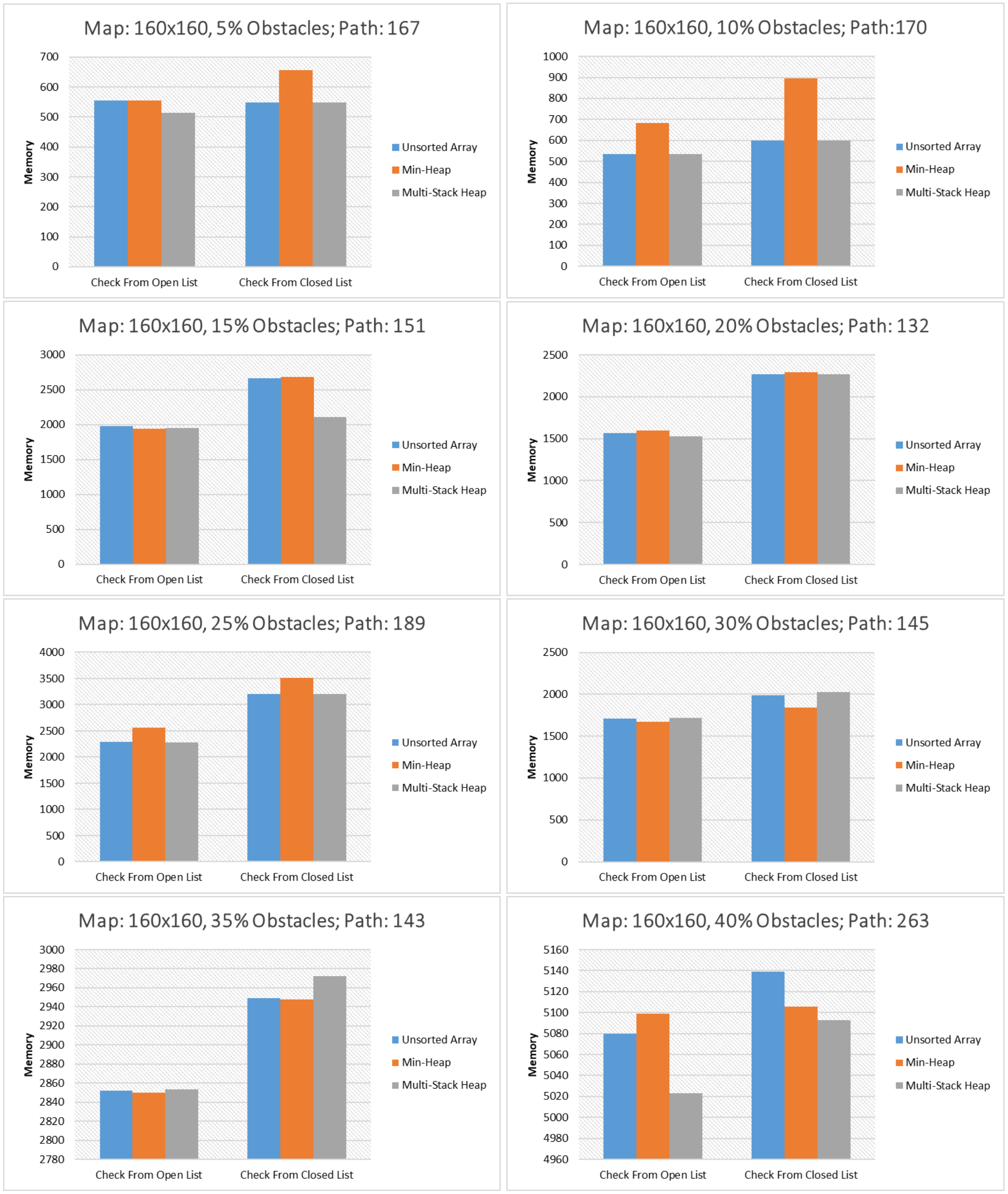

Fig. 41: Memory Requirement of 160x160 Map With Obstacles (5% - 40%)

It is hard to determine which data structure used less memory from Fig.41 and Fig.42, as any of them could be the one used the least memory but in some case, it also

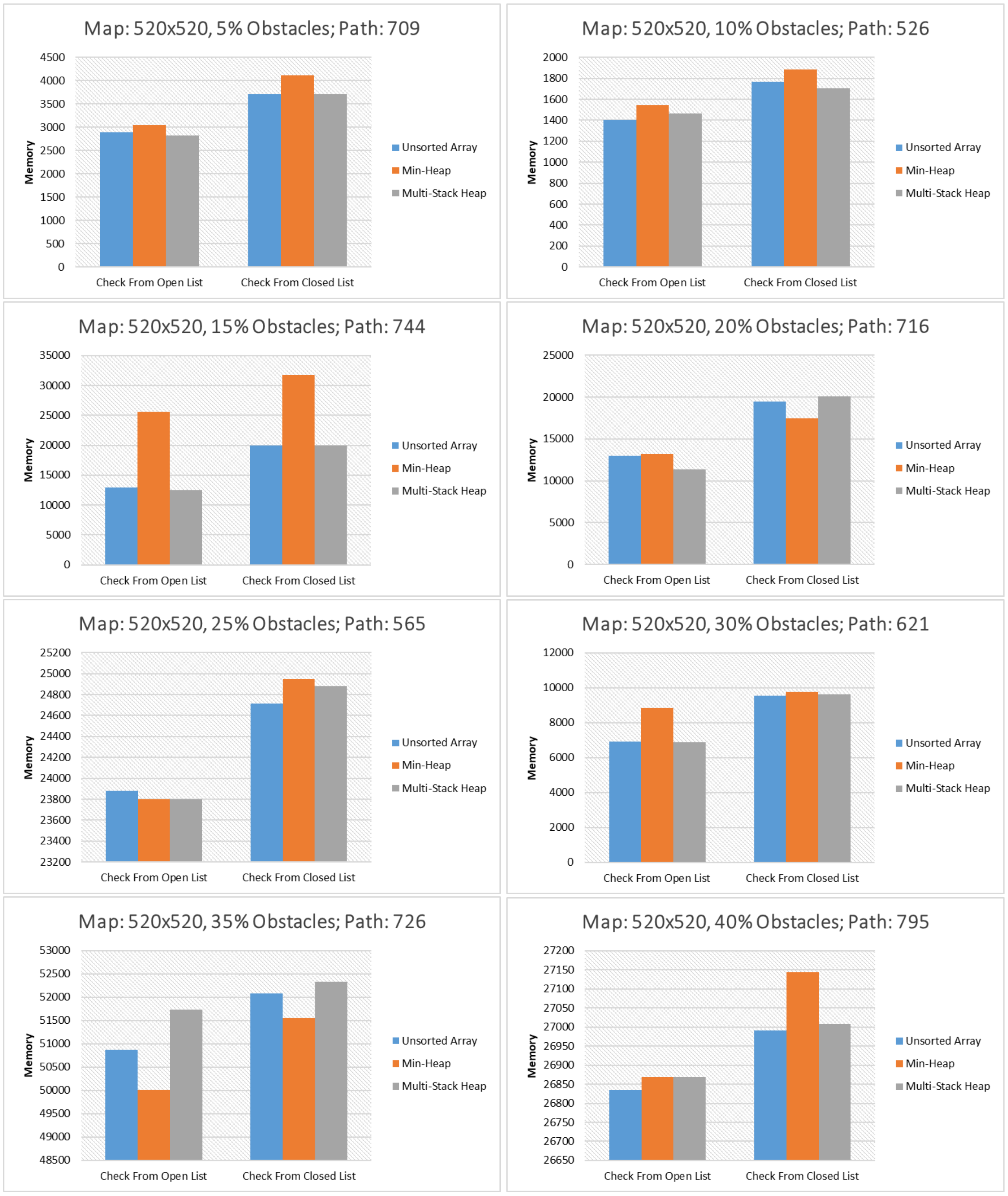

Fig. 42: Memory Requirement of 520x520 Map With Obstacles (5% - 40%)

could be the one used the most memory. However, comparing the two implementation plans, it can be found obviously that “Check From Closed List” used more memory than “Check From Open List”, which means, “Check From Closed List” explored more nodes in the *open list* and the *closed list*.

CHAPTER 6

Conclusion

In this paper, we explored the process of A* in pathfinding, and found that when the agent is only allowed to walk horizontally or vertically with Manhattan heuristic, there are at most 2 different f values in the *open list* at the same time, and when there are two f values, they must differ in 2. Based on this observations, we proposed a new data structure for the *open list* called multi-stack heap, and we analyzed it and compared it with the unsorted array and min-heap, which are the typical data structures to implement the *open list*.

Theoretically, multi-stack heap takes $O(1)$ to do the insertion and deletion, while the unsorted array needs $O(1)$ and $O(n)$ for the insertion and deletion, and the min-heap requires $O(log\ n)$ to do the same operations. All of them cost $O(n)$ to check whether the inserting node is in the *open list* before the insertion. If found a duplicated one, then the min-heap needs extra $O(log\ n)$ to decrease the key, but the unsorted-array and multi-stack heap just takes $O(1)$ to do it. Additionally, we found that the traditional "whetherContains" operation of A* is very expensive as it needs $O(n)$ to process it and it is triggered before every insertion. Considering we are using admissible and consistent heuristic, we came up with another implementation plan called "Check From Closed List". It is supposed to use $O(1)$ while the *closed list* is implemented in the hash table, though the performance could be affected when the decreaseKey happens frequently.

In the experiment, we found that the multi-stack heap always takes the least time among the three data structures, though the performances of the unsorted array and min-heap are not stable. In most cases, the min-heap needs less time than unsorted array; but when the decreaseKey is triggered frequently, the min-heap might spent more time than the unsorted array. And numbers of operations for the three data structure are in same condition as the runtime, the multi-stack heap executed the smallest number of operations during the search. However, we did not find any clear results about the memory requirements of different data structures, it highly relies on the distribution of obstacles but we generated the map randomly. Additionally, runtime and number of operations are decreased significantly for each data structure when we implemented A* with “Check From Closed List” instead of “Check From Open List”, though decreaseKey could affect the performance of the unsorted array and min-heap. The performance of the multi-stack heap could be improved a lot by “Check From Closed List”, especially on the number of operations. However, the disadvantage of “Check From Closed List” is that it requires more memory than “Check From Open List”.

In conclusion, our new data structure multi-stack heap could be more efficient than the unsorted array and min-heap whether on runtime and number of operations. The implementation plan “Check From Closed List” also could improve the performance of A*, especially on the number of operations, which could perform much better than the traditional “Check From Open List”.

CHAPTER 7

Future Work

We proposed multi-stack heap based on the precondition: four directions only with Manhattan distance. However, when the agent is allowed to walk diagonally, and use the Euclidean or Octile distance, there would be more than just two different f values at the same time, and they will not simply differ in 2. Though we can still use the data structure shown on Fig.16, we cannot take advantage of this data structure as there are not many nodes on the same stack. When there is only one node in each stack, then it is the same with the min-heap. In the future work, we can keep exploring the multi-stack heap to make it suitable for any condition. We can design a range of f value for each stack to decrease the number of stacks on the heap, it will have fewer "nodes" than the min-heap. However, how to set the range of each stack is still a question to be researched. Moreover, it is also a solution to give up the optimal path but get a suboptimal path in less time by setting a proper f value range for each stack.

Additionally, from the experiment we found that the distribution of obstacles on the testing graph could affect the performance of different data structures significantly, especially when "decreaseKey" happens frequently. It would be an interesting research direction in the future work to explore that how does the distribution of obstacles on the map can affect A* algorithm's performance.

REFERENCES

[1] Algfoor, Z. A., Sunar, M. S., and Kolivand, H. (2015). A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015:7.

[2] Björnsson, Y. and Halldórsson, K. (2006). Improved heuristics for optimal pathfinding on game maps. *AIIDE*, 6:9–14.

[3] Chen, H., Chen, S., and Rosenberg, E. S. (2018). Redirected walking strategies in irregularly shaped and dynamic physical environments. In *IEEE VR Workshop on Everyday Virtual Reality*.

[4] Cherkassky, B. V. and Goldberg, A. V. (1996). Heap-on-top priority queues. *TR*, 96(42).

[5] Denardo, E. V. and Fox, B. L. (1979). Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27(1):161–186.

[6] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

[7] Fletcher, G. H., Sheth, H. A., and Börner, K. (2004). Unstructured peer-to-peer networks: Topological properties and search performance. In *International Workshop on Agents and P2P Computing*, pages 14–27. Springer.

[8] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76.

[9] Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N., Holte, R. C., and Schaeffer, J. (2014). Enhanced partial expansion a. *Journal of Artificial Intelligence Research*, 50:141–187.

[10] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

[11] Haslum, P., Bonet, B., Geffner, H., et al. (2005). New admissible heuristics for domain-independent planning. In *AAAI*, volume 5, pages 9–13.

[12] Holte, R. C., Perez, M., Zimmer, R., and MacDonald, A. (1996). Hierarchical A*. In *AAAI*, volume 1, pages 530–535.

[13] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

[14] Lee, W. and Lawrence, R. (2013). Fast grid-based path finding for video games. In *Canadian Conference on Artificial Intelligence*, pages 100–111. Springer.

[15] Mathew, G. E. and Malathy, G. (2015). Direction based heuristic for pathfinding in video games. In *Electronics and Communication Systems (ICECS), 2015 2nd International Conference on*, pages 1651–1657. IEEE.

[16] Maurer, W. D. and Lewis, T. G. (1975). Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19.

[17] Pearl, J. and Kim, J. H. (1982). Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (4):392–399.

[18] Sharma, P. and Khurana, N. (2013). Study of optimal path finding techniques. *International Journal of Advancements in Technology*, 4(2):124–130.

[19] Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66.

[20] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News*, 30(5):45–57.

[21] Silver, D. (2005). Cooperative pathfinding. *AIIDE*, 1:117–122.

[22] Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148.

[23] UDK (2012 (accessed July 3, 2018)). Navigation mesh reference. Available at `https://api.unrealengine.com/udk/Three/NavigationMeshReference.html`.

[24] Van Toll, W., Cook, A. F., and Geraerts, R. (2011). Navigation meshes for realistic multi-layered environments. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3526–3532. IEEE.

[25] Van Toll, W., Cook IV, A. F., Van Kreveld, M. J., and Geraerts, R. (2017). The explicit corridor map: A medial axis-based navigation mesh for multi-layered environments. *arXiv preprint arXiv:1701.05141*.

[26] Vermette, J. (2011). A survey of path-finding algorithms employing automatic hierarchical abstraction. *Journal of the Association for Computing Machinery*, V:377–383.

[27] Wardhana, N. M., Johan, H., and Seah, H. S. (2013). Enhanced waypoint graph for surface and volumetric path planning in virtual worlds. *The Visual Computer*, 29(10):1051–1062.

[28] Williams, J. (1964). Algorithm 232 heapsort. *Communications of the ACM*, 7(6):347–348.

[29] Zhang, Z., Sturtevant, N. R., Holte, R. C., Schaeffer, J., and Felner, A. (2009). A* search with inconsistent heuristics. In *IJCAI*, pages 634–639.

[30] Zhu, W., Jia, D., Wan, H., Yang, T., Hu, C., Qin, K., and Cui, X. (2015). Waypoint graph based fast pathfinding in dynamic environment. *International Journal of Distributed Sensor Networks*, 11(8):238727.

VITA AUCTORIS

NAME:	Qing Cao
PLACE OF BIRTH:	Anhui, China
YEAR OF BIRTH:	1992
EDUCATION:	Tianjin Polytechnic University, B.Eng., Computer Science and Technology, Tianjin, China, 2015
	University of Windsor, M.Sc in Computer Science, Windsor, Ontario, 2018