University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

12-12-2018

Performance Evaluation of Competing Data Structures

Mohsen Tavakoli

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Tavakoli, Mohsen, "Performance Evaluation of Competing Data Structures" (2018). *Electronic Theses and Dissertations*. 7626.

https://scholar.uwindsor.ca/etd/7626

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Performance Evaluation of Competing Data Structures in Pathfinding

By

Mohsen Tavakoli

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

©2018 Mohsen Tavakoli

Performance Evaluation of Competing Data Structures in Pathfinding

by

Mohsen Tavakoli

APPROVED BY:

M. Hlynka
Department of Mathematics and Statistics

M. Kargar
School of Computer Science

S. Goodwin, Advisor
School of Computer Science

Dec 12, 2018

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Pathfinding is an essential part of many applications, including video games and robot navigation. A pathfinding algorithm usually finds a path from the given starting point to the endpoint. Many different implementations of pathfinding solutions exist in the industry. One of the most known and used of these algorithms is A*. A* will find the shortest path from the starting point to the endpoint. Classic A* algorithm can guarantee the shortest path to the desired destination which was introduced in 1968 by Hart, Nilsson, and Raphael. A* is widely used in the game industry to solve the shortest path problem. The A* algorithm utilizes two data structures. A* explores the nodes in the graph from the start position one by one and assign them a value of F which is the sum of G cost and H cost. G Cost is the actual cost of exploring the node from the starting position to the current node, and H cost is the estimation of the cost of from the current node to the goal node. The Open List keeps all of the nodes that are not explored at each iteration of the algorithm. In each iteration, the algorithm removes the node with the least value of F cost and run the algorithm. If the node is not the goal, it will be added to the closed list. Interactions with the open list, which are insert (current node) and remove Min, are costliest part of the algorithm. It is well known that using a priority queue will increase the performance of this algorithm. A number of priority queues have been used to implement A* and improve the performance of this algorithm. We propose to use a Lazy binary heap and evaluate its performance compare to other data structures. We expect that due to decreasing the size of current open list, it will outperform other binary heaps.

AKNOWLEDGEMENTS

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 Partial Min-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. Hlynka 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.

The time that I spent on this thesis has been very intense. To anyone reading this paper, I will say that with hard work and commitment, you can achieve your goal in life.

Finally, I would like to thank my parents, my dear brothers Ahmad and Ehsan and finally my friends for their support over the years.

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY III

ABSTRACT IV

AKNOWLEDGEMENTS V

LIST OF TABLES VIII

LIST OF FIGURES IX

1 Introduction 1

1.1 Thesis Claim . . . 1
1.2 Pathfinding . . . 1
1.2.1 Pathfinding Problem . . . 1
1.3 Graph Representations . . . 2
1.3.1 Waypoints . . . 2
1.3.2 Navigation Mesh . . . 2
1.3.3 Grids . . . 3
1.4 Heuristic . . . 4
1.4.1 Manhattan Distance . . . 5
1.4.2 Euclidean Distance . . . 5
1.5 Pathfinding Algorithms . . . 6
1.6 A* Algorithm . . . 7
1.6.1 Heuristic Consistency . . . 10
1.6.2 Problem Statement . . . 11
1.6.3 Min-Heap Example . . . 12
1.7 Thesis Contribution . . . 13
1.8 Thesis Organization . . . 14

2 Literature Review 16

2.1 A* Data Structure . . . 16
2.2 Closed Set . . . 17
2.3 Open Set . . . 17
2.3.1 Array . . . 18
2.3.2 Hash Table . . . 18
2.3.3 Binary Min-Heap . . . 19
2.3.4 Fibonacci Heap . . . 21
2.3.5 Multilevel Buckets . . . 22
2.3.6 MultiQueues . . . 22
2.3.7 Heap On Top Priority Queues . . . 23
2.4 Summary . . . 24

3 Cached Min-Heap and Partial Min-heap **26**
3.1 Motivation . . . 26
3.2 Cached Min-Heap . . . 29
3.2.1 Cached Min-Heap Operations . . . 30
3.3 Partial Min-Heap . . . 36
3.3.1 Partial Min-Heap Operations . . . 36
3.4 Partial Min-Heap Case study . . . 39
3.4.1 Partial Min-Heap Success . . . 39
3.4.2 Partial Min-Heap Failure or Sub-optimallity . . . 41
3.4.3 Summary . . . 41

4 Experiments and Results **42**
4.1 Implementation Methods . . . 42
4.2 Experimental Environment . . . 43
4.3 Experiment Setups . . . 44
4.4 Performance Evaluation . . . 45
4.4.1 Success . . . 45
4.4.2 Time . . . 45
4.4.3 Path Length and Cost . . . 46
4.4.4 Nodes Expanded . . . 46
4.4.5 Operations . . . 46
4.5 Experiment Results and Analysis . . . 49
4.6 Number of operations . . . 51
4.7 Runtime . . . 62
4.8 Summary . . . 64

5 Conclusion **68**

6 Future Work **70**

APPENDICES **71**

REFERENCES **87**

VITA AUCTORIS **90**

LIST OF TABLES

1 Time Complexity Comparison of Different Data Structures 24

2 Summary of experiments 50

3 Map Size 120*120 using K=6 Full Data 72

4 Map Size 120*120 using K=7 Full Data 73

5 Map Size 120*120 using K=8 Full Data 74

6 Map Size 200*200 using K=6 Full Data 75

7 Map Size 200*200 using K=7 Full Data 76

8 Map Size 200*200 using K=8 Full Data 77

9 Map Size 200*200 using K=9 Full Data 78

10 Map Size 300*300 using K=8 Full Data 79

11 Map Size 300*300 using K=9 Full Data 80

12 Map Size 400*400 using K=8 Full Data 81

13 Map Size 400*400 using K=9 Full Data 82

14 Map Size 400*400 using K=10 Full Data 83

15 Map Size 512*512 using K=9 Full Data 84

16 Map Size 512*512 using K=10 Full Data 85

17 Map Size 512*512 using K=11 Full Data 86

LIST OF FIGURES

1 Waypoint Map . . . 3

2 Navigation-Mesh Map . . . 3

3 Square-Grid Map . . . 4

4 Manhattan & Euclidean Distance . . . 6

5 A* algorithm on the left & Best-First search in the middle & Dijkstra's algorithm on the right finding the same path . . . 8

6 Current Min-heap . . . 12

7 Problem Itteration 1 . . . 12

8 Current Min-heap . . . 13

9 Problem Itteration 2 . . . 13

10 Current Min-heap . . . 13

11 Problem Itteration 3 . . . 13

12 Implementation of Min-Heap using an Array . . . 19

13 Inserting to Min-Heap . . . 20

14 Remove Min - Min-Heap . . . 21

15 Fibonacci-Heap Insert Example . . . 22

16 Example of search environment . . . 27

17 Example of search environment (Exploring a node) . . . 27

18 Example of search environment (Exploring a node) . . . 28

19 Cached Min-Heap Data Structure d=3 . . . 30

20 Cached Min-Heap Data Structure Example d=3 . . . 31

21 Cached Min-Heap Data Structure Example d=3 . . . 32

22 Cached Min-Heap Data Structure Example d=3 . . . 32

23 Cached Min-Heap Data Structure Contains . . . 34

24 Cached Min-Heap Data Structure Replenish Cache Failure Using Heap Sort . . . 35

25 Partial Min-Heap Data Structure Insert Example d=3 38
26 Partial Min-Heap Data Structure Remove Min Example d=3 38
27 Pathfinding Example 40
28 Pathfinding Example 40
29 Pathfinding Example 41
30 Map Size 45*45 - Obstacle chance 10% 43
31 Map Size 60*60 - Obstacle chance 10% 49
32 Map Size 120*120 using K=6 Full Data in Table 3 52
33 Map Size 120*120 using K=7 Full Data in Table 4 53
34 Map Size 120*120 using K=8 Full Data in Table 5 54
35 Map Size 200*200 using K=6 Full Data in Table 6 55
36 Map Size 200*200 using K=7 Full Data in Table 7 56
37 Map Size 200*200 using K=8 Full Data in Table 8 57
38 Map Size 200*200 using K=9 Full Data in Table 9 58
39 Map Size 300*300 using K = 8 Full Data in Table 10 59
40 Map Size 300*300 using K = 9 Full Data in Table 11 60
41 Map Size 400*400 using K=8 Full Data in Table 12 61
42 Map Size 400*400 using K=9 Full Data in Table 13 62
43 Map Size 400*400 using K=10 Full Data in Table 14 63
44 Map Size 512*512 using K=9 Full Data in Table 15 64
45 Map Size 512*512 using K = 10 Full Data in Table 15 65
46 Map Size 512*512 using K = 11 Full Data in Table 16 66
47 Run Time Map Size 400*400 using K=9 & 10 67
48 Run Time Map Size 512*512 using K=10 & 11 67

CHAPTER 1

Introduction

1.1 Thesis Claim

In this thesis, we present two new data structures for the A* search algorithm which implement the *open set* Partial MinHeap, and Cached MinHeap. Test results in comparison to traditional data structures used for this algorithm, such as unsorted list, min-heap and heap-on-top priority queue (Hot Queue), show improvements in runtime. Additionally, because of the distributed architecture of this data structure, the number of operations is often reduced.

1.2 Pathfinding

Pathfinding is the task of finding a traversable path from the starting position to the ending position. A universal problem which exists in multiple areas such as games, robotics and computer networks. These problems require a reliable solution. In this thesis, we focused on improving the A* algorithm performance in fully connected grids.

1.2.1 Pathfinding Problem

A specific problem of RTS (Real Time Strategic) for games is pathfinding. By the nature of RTS games, players are not concerned with agents in the game and their movements. Developers determine each unit's actions and their behaviors[23]. Many solutions exist for the problem of single source shortest problem such as Dijkstra's

algorithm, A*, Hierarchical and Cooperative pathfinding algorithms. This research aimed to find a better solution for A* *Open set*, which has a significant impact on runtime. In this research, we treat operations and time as a performance variable. We came up with a solution that delivers suboptimal path, using less memory and time which could be a better solution in the right circumstances.

1.3 Graph Representations

In pathfinding, we need a representation of the search space for our algorithm to find a correct path from our starting point to our ending point. A pathfinding algorithm tries to find a path using a simplified representation of the search space. Common ways of representing the maps are Waypoints, Navigation Meshes, and Grids.

1.3.1 Waypoints

Using a collection of linked and fully connected nodes for navigation is the waypoint(Fig. 1) system. Each waypoint refers to a physical space or coordinates in the map. AI agents are capable of traveling from one waypoint to another waypoint. Game engines such as Unity3D, Unreal Engine support waypoints. The advantage of using waypoints is that it will decrease the amount of memory usage since it uses fewer nodes [13]. The disadvantage of waypoints is that the path found is unrealistic and sub-optimal. Waypoints are usually created manually by the developer to have the highest performance.

1.3.2 Navigation Mesh

Navigation meshes(Fig. 2) or nav-mesh are a way of representing the map with a group of polygons connected within the map. Each polygon can have attributes such as the cost of traversing, type of terrain, require a tool to pass, etc. We do not need to store the obstacles in navigation mesh. Agents are free to roam from polygons to another. The pathfinding algorithm finds a series of polygons for the agent to cross to

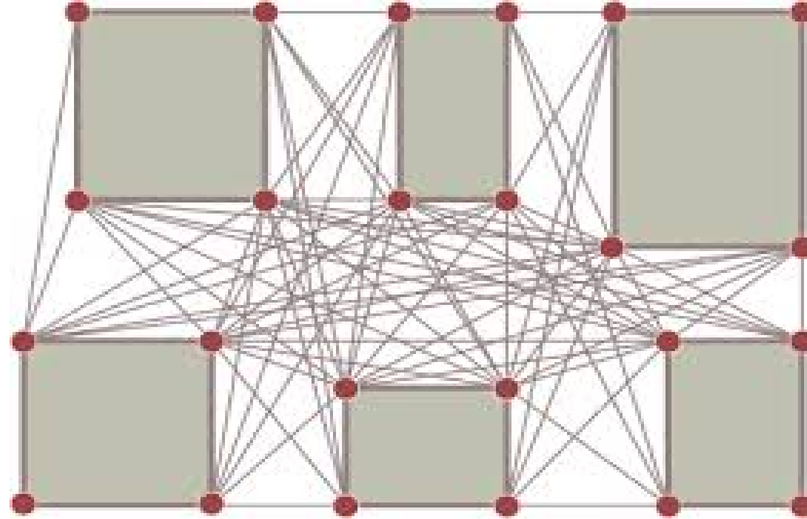


Fig. 1: Waypoint Map

reach the desired destination. Using polygons to represent nodes in the map results in using less memory, but the path quality will suffer.

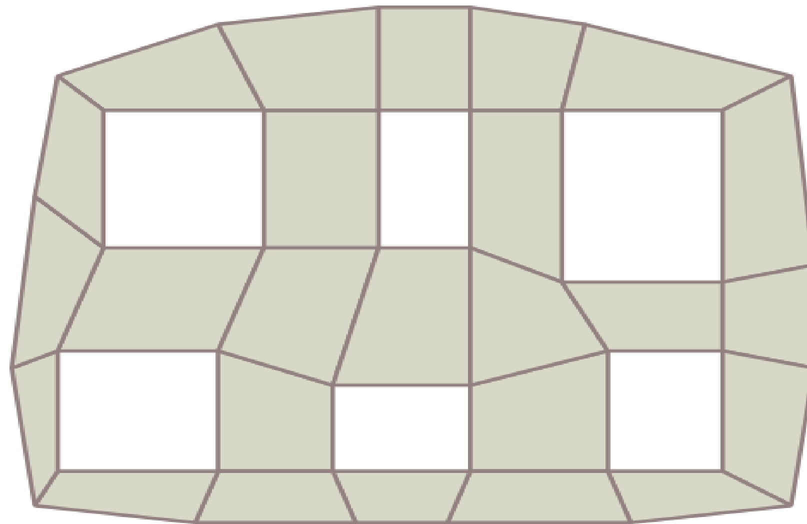


Fig. 2: Navigation-Mesh Map

1.3.3 Grids

A* algorithm is designed to work with arbitrary graphs. Grids are one of the more commonly used ways of representing the map. Grids divide the search space into uniform, regular shapes tiles. Each tile can inherit multiple characteristics such as

walkable or cost. Grids will cover all of the game maps such as obstacles. Nav-mesh and waypoints only addressed areas that are traversable. Commonly used forms are squares, triangles and hexagonal.

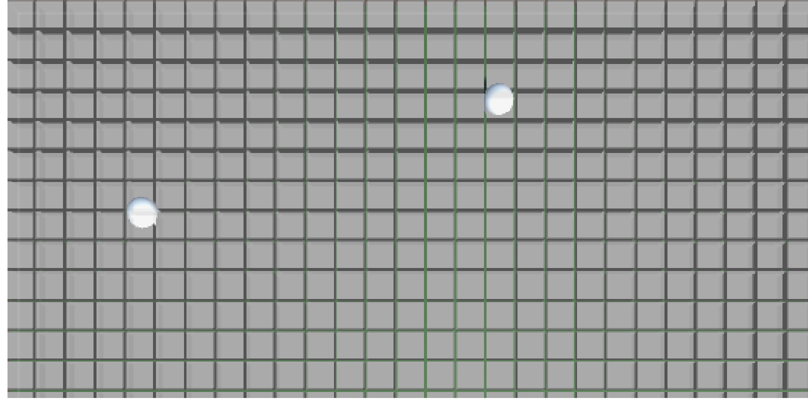


Fig. 3: Square-Grid Map

Square tiles are the most common grid(Fig. 3). Each tile in the map uses the familiar X, Y coordinates. Most of the commercial games such as Warcraft III, Dragon Age: Origins are using grids in their games. Each grid has three different parts: tiles, edges, and vertices. Faces are a 2D surface surrounded by the edges. Lines that are enclosed by two vertices are edges. Each vertex is the point where each tile's edges meet to form the desired shape.

In this research, we ran experiments using squared grids since they are easy to visualize and implement. We ran our tests on randomly generated maps with a chance of blocking grid cells. Based on the suggestion [23] we ran our experiments using an implementation of A* on fully connected squared grids so that other scientists can compare their results to our results.

1.4 Heuristic

Classic search algorithms such as Dijkstra's algorithm explore the search space to find the shortest path. The heuristic function guides the algorithm into the direction of the correct location of the goal node, which may result in finding optimal path

faster[19]. The heuristic value is an estimation of the path cost from any given node and represented as h(n). If our heuristic has a value of 0, our A* algorithm will act as Dijkstra's algorithm. If the returned value by our heuristic function is smaller or equal than the actual cost of reaching the goal, A* is guaranteed to find the optimal path. If the value is greater than the actual cost of the path, A* is not guaranteed to find the optimal path. If our heuristic function is not admissible, which means that it will not overestimate or underestimate the cost of the actual path, our pathfinding algorithm is guaranteed to find the optimal path, which means that if our heuristic is accurate, it will result in that our pathfinding algorithm only expanding the nodes along the path. The developer can pre-compute the heuristic value the shortest path between any pair of nodes in the map. This approach is not suitable for large maps since that the precalculated heuristic values occupy more memory than the search space representation [4]. There are three well-known heuristics functions for calculating the shortest distance between two given nodes namely, Manhattan distance and Euclidean distance.

1.4.1 Manhattan Distance

This heuristic is standard for squared grids that movement is only allowed non-diagonally. Manhattan distance is not admissible since it overestimates the cost of diagonal movements.

$$ManhattanDistance = |dX| + |dY| \tag{1}$$

1.4.2 Euclidean Distance

Euclidean distance is the mathematically calculated straight-line distance between two points in the search space. Euclidian distance is accurate when there are no obstacles in the search space since it is the cost of direct movement from the starting position to the end position. This heuristic function will result in finding the shortest path exploring the fewer number of nodes. Since that calculating Euclidean distance

requires computational power, in some of the cases not using the heuristic function might result in less number of operation to find the correct path [14].

$$EuclideanDistance = \sqrt{dX^2 + dY^2} \tag{2}$$

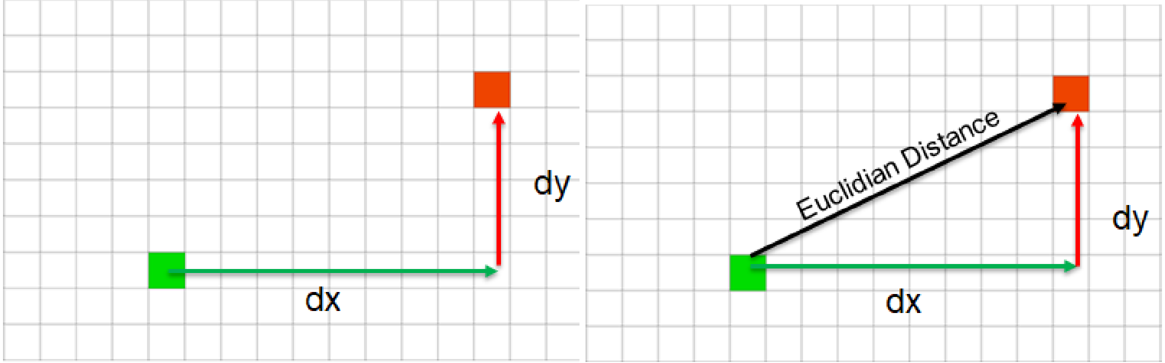


Fig. 4: Manhattan & Euclidean Distance

In our research, we designed our agent to be able to move in eight directions and our map representation was squared grids. We used squared grids since that it was easy to implement and understand. Also, most of the commercial games use this representation, and we wanted to compare our results compare to actual game maps. We selected our heuristic function as Euclidean distance based on our map representation and our agent's movements. We fixed our non-diagonal movement cost to 10 and our diagonal movement to 14.

1.5 Pathfinding Algorithms

The single source shortest path problem is searching for a traversable path with the least path cost from a given source to the desired destination. Existing solutions for this problem divide to two categories: informed and un-informed search. Breadth-first Search and Depth-first search do not use the heuristic information available based on the map, so their performance suffers since they blindly search for the goal node. Not using the existing data will result in exploring multiple unnecessary nodes, hence increasing the time of the algorithm to find a path [2]. Another approach

of finding a solution to the shortest path problem is using the given information to optimize the process of finding a solution. Information such as the location of the goal node in the search space, the relative cost of reaching the goal node might help our pathfinding algorithm to perform better in terms of the number of nodes expanded. A* algorithm [15] and Dijkstra's algorithm [9] are two of the most popular solutions for solving this problem. Game industry mostly used A* to develop and solve their pathfinding problems since it requires less computational power and it has better performance compare to Dijkstra's algorithm. Researchers and developers proposed different versions of A* algorithm to increase the performance of it [4] and mostly tailor it to their need. Researchers suggested that improvements are possible in terms of performance by pre-calculating and processing of the map in Partial Pathfinding[22] for A* algorithm. Algorithms such as Hierarchical Pathfinding A* [3] proposed a clustering solution to divide the search space into local and global clusters. Iterative Deepening A*[17] combined the depth-first search algorithm with A* algorithm which uses the heuristic function to guide the search algorithm in the correct path.

Our primary focus in this research paper is the widely explored and popular A* pathfinding algorithm. We analyzed the performance of the A* algorithm's data structures and believed that we could improve its performance.

1.6 A* Algorithm

A combination of the Dijkstra's algorithm and greedy Best First Search is A* search algorithm. Dijkstra's algorithm is guaranteed to find the shortest path, but it explores all the directions in the search space and will allow us to find the path to multiple locations. Best First Search explores in the goal direction to find the shortest path, but it is not guaranteed to find the optimal path. A* algorithm combines the idea of using the actual cost of reaching the goal and the estimated cost of reaching the goal to find the optimal path to the goal.

Dijkstra's algorithm uses $f(n) = g(n)$ which is the actual cost from the starting

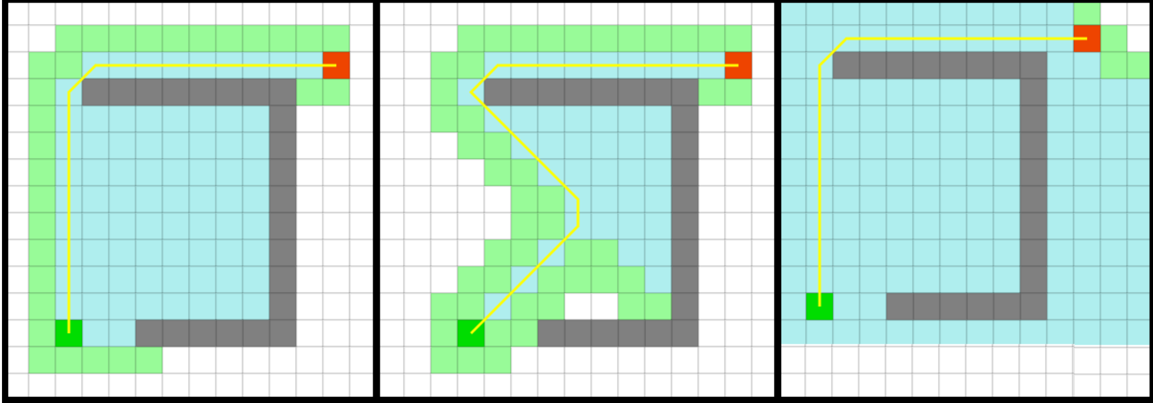


Fig. 5: A* algorithm on the left & Best-First search in the middle & Dijkstra's algorithm on the right finding the same path

node, to find the shortest path to the goal node. Best-First search uses $f(n) = h(n)$, which is the estimated cost from the current node to the goal node, to find the shortest path. A* algorithm combines these two values to find the shortest path to the goal node.

A* search algorithm constructs a path from the starting node to the goal node by following the nodes with the lowest f cost. This algorithm keeps track of alternative path nodes with their f cost. In each state, A* algorithm will expand the node with least f value till it reaches the goal node.

A* algorithm maintains two data structures to function. A* keep the nodes that have been visited but not expanded in the *openset*. These nodes are a queue of nodes that are possible to be the shortest path. *Closedset* contains already expanded nodes that have been extracted from the *openset* and examined.

A* algorithm initially inserts the starting node into the *openset*. Then the algorithm will explore all of the traversable neighbours of the current node with the smallest f cost in the *openset*, calculate their f cost and insert them into the *openset*. Since it is required to keep track of each node's path to the current location, each node also holds a pointer to its parent. The algorithm chooses the node with the smallest f cost from the *openset*. If the current node is not the goal node, the algorithm will insert the node to the *closed set*, and it will repeat the process till that either to find the goal or the *openset* is empty, which means that the algorithm failed to find a

path.

$$f(n) = g(n) + h(n) \tag{3}$$

Calculating the current f cost of node n is the total of $g(n)$ and $h(n)$. The actual cost of reaching the node n from the starting node is $g(n)$ and the estimated of cost of reaching the goal node from the node n is $h(n)$.

A* Algorithm.3 is fairly simple to implement and understand. Initially our *open set* is empty, so we add the *start* node to the *open set* and calculate the f cost for the *start* node. Our main loop, extract the *current* node with the least value of f cost from the *open set* in each iteration and examine it. If the *current* extracted node is our *goal node*, then the algorithm will return the path from the starting node to the *goal* node. To find the path from the *start* node and to our *goal* node, each node also keeps a pointer to its parent node. If the *current* node is not our *goal* node, the algorithm will remove the *current* node from the *open set*, insert it to the *closed set* and it will explore all of it's neighbours. If the neighbour is not in our *closed set*, and it is not a member of the *open set*, then the algorithm will insert it to the *open set* and set the *current* node as the parent node for it. If the neighbour node already exists in the *open set* the algorithm will calculate a new f cost for the neighbour node. If the new f cost is greater equal to the current f cost, the node will be discarded. Otherwise, the new f cost will replace the old f cost and set the *current* node as the new parent for the neighbour node. The reason for this if statement is that we do not want to insert duplicate nodes that have been already explored back into our *open set*.

A* search algorithm is guaranteed to find a solution if there is one which means that it is complete. Pathfinding solutions are exponential problems which means that the size of the search space and its complexity has an evident impact on the execution time[10]. Optimal pathfinding solutions in Real time strategy games have different definitions. Some games require to find a solution relatively fast but not necessarily the shortest path that exists since the goal is efficiency, not accuracy. Usually in

Algorithm 1 A* Algorithm

```
1: Start:
2:     open_set = {start}
3:     f(start) = h(start)
4:     closed_set = { }
5: while open_set is not empty do
6:    current = extract the node with lowest f cost from open_set
7:    if current = goal_node then
8:       return "Path found"
9:    end if
10:   open_set.remove(current)
11:   closed_set.insert(current)
12:   for each neighbour of current do
13:      if neighbour in closed_set then
14:         continue
15:      end if
16:      if neighbour not in open_list then
17:         open_set.insert(neighbour)
18:      end if
19:      if g(current) + distance(current, neighbour) < g(neighbour) then
20:         g(neighbour) = g(current) + distance(current, neighbour)
21:         f(neighbour) = g(neighbour) + heuristic_function(neighbour, goal)
22:         neighbour.setParent(current)
23:      end if
24:   end for
25: end while
26: return "Failed to Find the Path"
```

RTS games, as long as the path introduces to the agent is close enough to the actual shortest path and it is not irrational in terms of movement, it is an acceptable path.

1.6.1 Heuristic Consistency

A* algorithm chose the best node to explore from the open set using the cost function which is $f(n) = g(n) + h(n)$ where $g(n)$ is the actual distance or cost of travesing to node n and $h(n)$ is the estimated cost of reaching the goal node. If our heuristic function is consistent the cost of traversing from our node x to the next node y will be [15]:

$$h(x) \leq d(x, y) + h(y) \tag{4}$$

This means that by moving from x to y, our overall cost of reaching the goal cannot be reduced more than the estimated cost of traversing from node x to the node y. If the node that our agent is exploring is the neighbour of the node x, the value of f is consistent since the $g(n)$ will increase as much as the $h(n)$ decreases[20]. We can define our consistent heuristic as:

$$|h(x) - h(y)| \leq d(x, y) \tag{5}$$

Based on those mentioned above, we can state that if our heuristic function is consistent while exploring the neighbours of the node x, our neighbour's f value is equal or smaller to the current node's f value. Using this information, already explored nodes in the *closed set* will not be revisited [16].

Theorem 1 *A consitent heuristing will garantee a non-decreasing f value while exploring along the path.*

Based on the Theorem 1, our A* algorithm will not revisit the nodes that are already explored, and we can skip the nodes that are already explored once in Algorithm 1 line number 16. Also, we can obtain that if our heuristic function is consistent and admissible, our number of expanded nodes is optimal [15]. If our heuristic function is not consistent but admissible, nodes in the *closed set* can be revisited [18] to find the path. Based on the theorem we can state that, if our heuristic is admissible and consistent, the A* search algorithm only expand the nodes that their f value is either smaller or equal to the current expanded node in the *open set*.

1.6.2 Problem Statement

Majority of the computational power required to find a path in A* algorithm happens in inserting of a new node to the data structure and removing the node with the lowest f cost from the *open set*. Although updating the f cost which also called as decrease key for the nodes that already exist in the *open set* requires operations as well, but this operation is not as frequent as inserting and removing nodes from the *open set*

if our heuristic function is consitent[24]. The data structure of the *open set* is a critical section of this problem, and it will help the algorithm to perform better or worst based on the solution. One of the widely used solutions for this problem is the binary Min-Heap. Heap data structure is sensitive to the inserted values and requires operations to maintain the min-heap properties. We assume that pathfinding data may be closer to worst case operations and the heap is required O(log n) operations to function, and we wanted to find a better data structure for this problem.

1.6.3 Min-Heap Example

In our example, we showcase a small pathfinding problem which proves our point that the heap data structure mostly requires worst-case operations to maintain the properties of a min-heap.

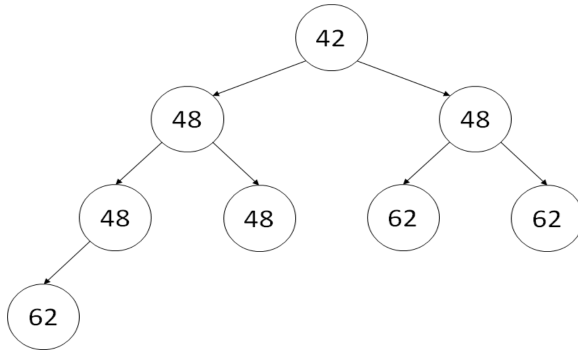


Fig. 6: Current Min-heap

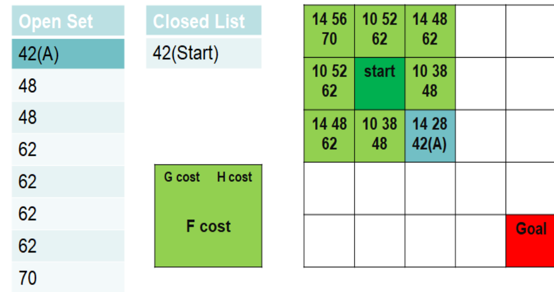


Fig. 7: Problem Itteration 1

In the example given in Fig.7, A* is trying to find the path from the start location to the goal location. After inserting the start node, A* will explore the neighbours of this node, and it will select the node with the lowest f cost which is shown as "Node A". The current state of our Min-Heap is also is represented in Fig.6.

In the second iteration A*, extract the current minimum from the min-heap and expand the neighbours of "Node B"(Fig.9). The current state of Min-Heap is shown in Fig.8.

In this iteration, A* algorithm extracts the node with the minimum f value, Since that the "Node C" (Fig. 11) is the goal node, the path is found and the algorithm

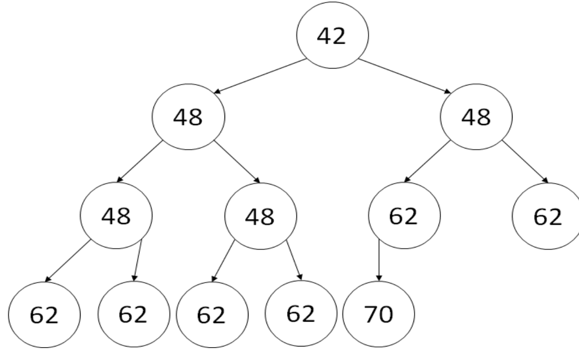


Fig. 8: Current Min-heap

Open Set	Closed List
42(B)	42(Start)
48	42(A)
48	
48	
48	
62	
62	
62	

G cost H cost
F cost

14 56 70	10 52 62	14 48 62		
10 52 62	start	10 38 48	28 34 62	
14 48 62	10 38 48	14 28 42(A)	24 24 48	
	28 34 62	24 24 48	28 14 42(B)	
				Goal

Fig. 9: Problem Itteration 2

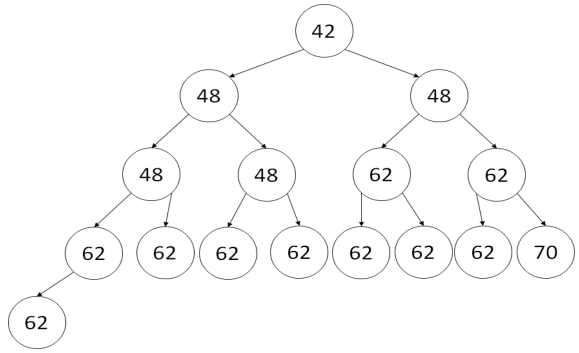


Fig. 10: Current Min-heap

Open Set	Closed List
42(C)	42(Start)
48	42(A)
48	42(B)
48	
48	
48	
48	
62	

G cost H cost
F cost

14 56 70	10 52 62	14 48 62		
10 52 62	start	10 38 48	28 34 62	
14 48 62	10 38 48	14 28 42(A)	24 24 48	42 20 62
	28 34 62	24 24 48	28 14 42(B)	38 10 48
		42 20 62	38 10 48	42 0 42(C)

Fig. 11: Problem Itteration 3

will stop. Each time that a new insertion was made to the Min-Heap, based on the data structure algorithm, the new node was inserted to the last place on the data structure. After that, it was compared to its parent, and in case of having a smaller value, the node swapped its location with its parent till that it finds its correct location. The same function happened when we removed the minimum node from the data structure. Based on the algorithm we swapped the last node's location with the node on top of the data structure, then checked if the node has a higher value than its children, we swapped its location with them, till the nodes are in their correct locations.

1.7 Thesis Contribution

As you can see in the given example, the number of operations was high due to the depth of our Min-Heap, since that we were keeping all of the nodes that A* algorithm

expanded and they were not expanded during our search to find the shortest path. If we limit the size of our data structure by limiting the depth of our Min-Heap, we could have performed less number of operation such as Swap operations and comparisons required by the Min-Heap to find the same path.

A* algorithm performance is limited due to the inserting a new node and removing the node with lowest f value operations. The algorithm inserts the expanded neighbours of the current node if they are not in the *closed set* and it will remove the node with least f value in the *open set*. Hence the performance of the data structure has a direct impact on the performance of the algorithm. Already existing solutions for this problem is using a priority-queue which usually is a Min-Heap. A Min-Heap requires $O(log\ n)$ number of operations to maintain the properties of a min-heap.

In this research, we introduced two new solutions based on the min-heap, which result in better performance in terms of operations and runtime in the right circumstances. The first solution is called Cached Min-Heap, which we took the idea from the Heap On-Top Priority Queues, but based on our own implementation and is our newly introduced algorithm. The second data structure that we implemented is the Partial Min-Heap, which is an optimistic solution for this pathfinding problem which under the right circumstances is able to find the correct optimal path with less number of operation and better runtime in comparison to the Min-Heap.

1.8 Thesis Organization

This thesis paper is organized into 6 chapters. The first contains introductory information about the pathfinding algorithms and necessary information regarding the A* algorithm and provides a small background about our work. The second chapter mainly focuses on the previously introduced solutions to the A* data structure. In the third chapter, we introduce our proposed data structures Cached min-heap and Partial min-heap and provide detailed functions of these data structures and analysis them. Our fifth chapter is a summary of our experiments combined with our test criteria and the analysis of the results. In the fifth chapter, we also compared our

results to the other existing solutions to the *open set*. The sixth chapter provides a summary of our analysis and concludes our work and the seventh chapter provide a possible blueprint of what can be the continuation of this research and how it might be improved uppon.

CHAPTER 2

Literature Review

2.1 A* Data Structure

The functionality of A* search algorithm is dependant on two data structures that are utilized in the main loop of this search algorithm. A* repeatedly extracts the node with the lowest f cost from it's *open set*, examine its neighbours to find the best subsequent node to explore till it finds the goal node. The main loop of this algorithm maintains its operations using two data structure *open set* and *closed set*, which their implementation is essential to the algorithm's performance. A* algorithm needs a data structure to maintain its *open set* to perform more efficiently. A* algorithm does not need to revisit the already expanded nodes from its *closed set* if the heuristic function is consistent.

A* search algorithm operations on *open set* data structure are as follows: "Insert", "Remove min", "Contains", and "Update Node". In each iteration, the algorithm removes the node with the lowest f value from the *open set* and explore all of the adjacent neighbours to the current node. Then the algorithm move the current node from the *open set* to the *closed set*. The algorithm will check if the neighbours of the current node are members of the *open set*. This operation is the contains operation. If they are not already a member of our *open set*, they will be inserted into the *open set*. In case of the nodes being members of the *open set*, the algorithm will decide based on their f value the next course of action. If the new f value is larger than the existing f value, the node will be discarded. Otherwise, the existing node f value need to be updated, and since the path reaching the node has been changed, the

node's parent should be changed too.

In this chapter, we discuss the already existing solutions of the *open set* and *closed set* implementations for the A* algorithm and their time complexity based n which is the number of members in each data structure.

2.2 Closed Set

If the heuristic function used in the A* algorithm is consistent, the implementation of the *closed set* will not have a direct impact on the performance of the algorithm. The algorithm already expanded the nodes that are members of the *closed set* and the purpose of keeping these nodes are preventing our algorithm to enter an infinite loop state [David Rutter]. The *closed set* can be implemented as an array or a hash table.

Operations required in the *closed set* is mainly membership tests. Implementation of the *closed set* using an array requires $O(n)$ operations to return the membership and using a hash table requires $O(1)$ operations. In this research, we used hash tables to implement our *closed set* since we were looking for an optimized solution and better performance to our pathfinding problem.

2.3 Open Set

Since that the majority of operations in A* search algorithm occur on the nodes in the *open set*, the algorithm is heavily dependant on the performance and efficiency of this data structure and researchers mainly focused on many solutions for the *open set*. Since that mostly each solution focuses on a particular problem, it is hard to compare them to each other. We mainly focused on inserting a new node, removing the min, contains, and update operation time complexity.

2.3.1 Array

Using an array to implement the *open set*, one solution is that the elements in the array are not sorted and the other solution is to be sorted.

Unsorted Array

Inserting a new node to an unsorted array takes $O(1)$ operation. We add the node at the last possible location in the array without any operations. Removing the node with the lowest f value requires $O(n)$ operations since we need to scan the array to find the node with lowest f value. Contains is the same as removing a node from the array and requires $O(n)$ operations. Updating a node's f value requires $O(n)$ operations as well since it needs to first find the node in the array, then change its f value.

Sorted Array

Since that the array requires to be sorted at any time during the operation, inserting a new node requires $O(n)$ operations. At first, the array needs to be scanned to find the correct locations of the inserted node based on the f value; then the subsequent nodes needs to be shifted to the new location. Removing the node with the lowest f value requires $O(1)$ operation since that the array is sorted and the node at the index or last location in the array is the minimum node. Finding the node in the array can be implemented using different methods. Using binary search to find the node in the array requires $O(log\ n)$ operations. Update method requires $O(n + log\ n)$ operations since that we need to find the node then locate the correct position of the node in the array.

2.3.2 Hash Table

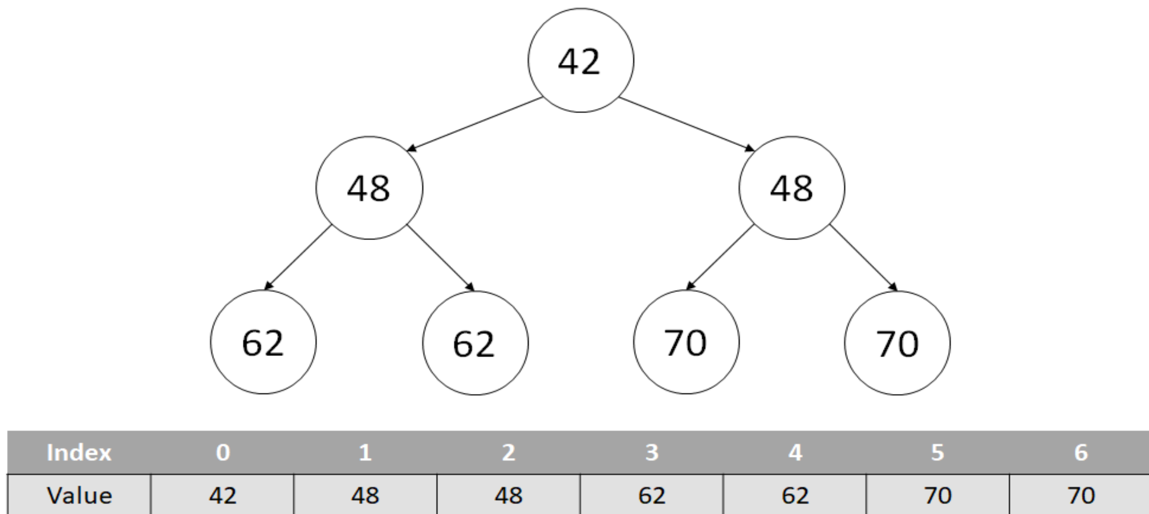
Using hash tables to implement the *open set* was suggested in this study [6] since it allows the algorithm an instant accessing time. Inserting a new node, update and contain functions need $O(1)$ operations to return the result, but the data structure

requires $O(n)$ operation to find the node with the lowest f value. The data structure should still follow the properties of a priority queue.

The main issue using a hash table is the indexing problem. In each search, items in the hash table need to acquire a hash key which requires computational power to calculate for each node. The developer needs to use the right hash algorithm to prevent duplicated keys.

2.3.3 Binary Min-Heap

Min-heap is one of the most popular solutions to implement the *open set* which was introduced in 1964 by Williams[1], is a complete binary tree which follows the properties of a priority queue. Each item in the min-heap required to have an index which allows the data structure to locate each node's location in the tree. Using an array (Fig.12) is one of the most popular ways to implement a binary min-heap. Nodes in the min-heap must have smaller or equal value than their children nodes. Min-Heap implements "Insert" and "Remove Min" functions.



Index	0	1	2	3	4	5	6
Value	42	48	48	62	62	70	70

Fig. 12: Implementation of Min-Heap using an Array

Insert

To insert (Fig. 13) a new node to the heap, this data structure inserts the new node to the last available location in the heap. Then it will check if the node is in the correct location by checking if the node's value is higher than its parent. If the node has a smaller value, the location of the newly inserted node will be swapped with its parent node. This operation usually is called the sort-up or bubble up operation. Adding a new node requires $O(log\ n)$ operations.

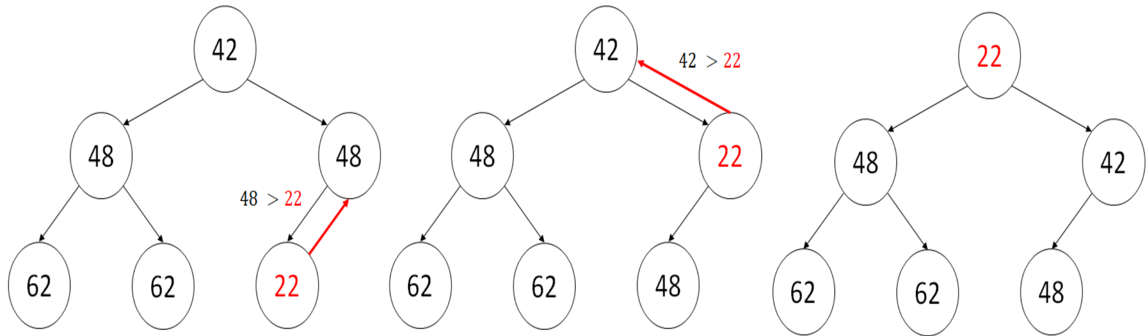


Fig. 13: Inserting to Min-Heap

Remove Min

To Remove the node with the min f value in the Min-Heap, this data structures remove the node at the top of the tree, replace it with the node at the last index in the tree. Then it will check if the node has a smaller value than any of its children. The node with the least value will be swapped with the node on top till the node find its correct location in the tree. This process also is called sort-down or bubble down operation. Removing the minimum node requires $O(log\ n)$ operations.

Contain and Update

To find the node which is the Contain operation in the min-heap, firstly we need to traverse the array which requires $O(n)$ operations. If the node needs to be updated, first we find the node in the heap, then we update its value. Since that the node's value is changed, we have to check if our data structure is not breaking the properties

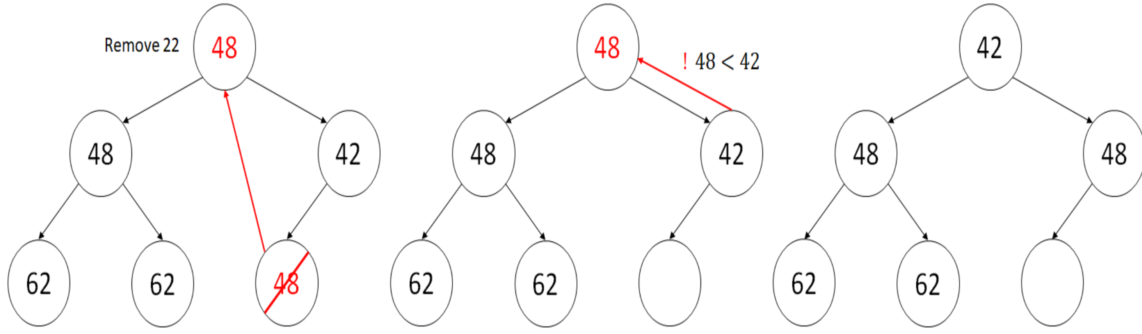


Fig. 14: Remove Min - Min-Heap

of a Min-Heap and find the correct location of the node in the heap which requires $O(log\ n)$ operations.

2.3.4 Fibonacci Heap

Despite the promising time complexities of Fibonacci heap [11], this data structure is not a popular solution to the A* shortest path problem due to the implementation complexities of the Fibonacci heap. This data structure mainly developed to improve the time complexity of Dijkstra's algorithm but the authors mentioned that it could be used as a priority queue in any problem. Fibonacci heap is a combination of heap-ordered trees. Each sub tree is a non-binary min-tree which has a pointer of its minimum member to a root list. Root list will keep track of all of the minimum members of the sub trees and has a pointer to the minimum member in the entire heap.

Insert and Remove

To add a new node to the Fibonacci heap, first, we create a singleton tree using that node then we insert the new singleton tree into the root list. If the new singleton tree is smaller than the current minimum member, we update the root list pointer. In the case of the given Example (Fig. 15), Adding the 21 element to the Fibonacci will result in the following figure. Fibonacci-heap insert requires $O(1)$ operation.

Removing the node with the least value from the Fibonacci-heap requires the

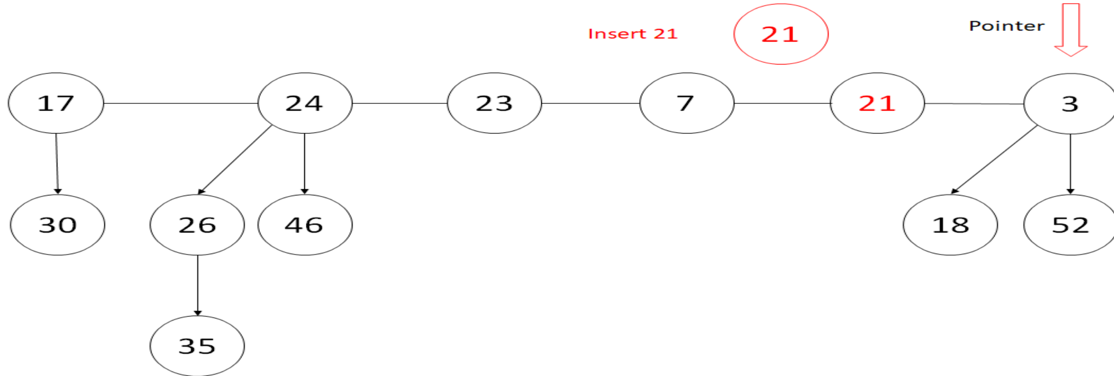


Fig. 15: Fibonacci-Heap Insert Example

following operations. First, the algorithm extracts the minimum node, then meld its children to the root list then consolidate the remaining trees so that no two roots have the same rank.

2.3.5 Multilevel Buckets

Multilevel buckets [12] is using the bucket data structure which maintains an array of buckets. In a data structure with K bucket levels, K = 0 is the lowest level, and K = K-1 is the highest level of the bucket, in which each bucket only keeps the nodes with certain value of f. The algorithm will use the ith bucket at the time and expand the nodes correspond to that bucket. If the current bucket is empty, the algorithm will use the next bucket with the values of f. In case of updating a node, the node will be removed from its current bucket, and it will be moved to the new bucket of its corresponding f value. Since that this data structure was introduced to work with Dijkstra's algorithm, there was no definition of the contain method. Based on the implementation of this data structure, inserting and removing a node will take a constant time based on the number of buckets used and the number of nodes.

2.3.6 MultiQueues

MultiQueues [21] is an array Q of multiple lock protected priority queues. In this data structure, accessing each priority queue requires that the priority queue is not locked.

Inserting a new node, will Lock the $Q[i]$ priority queue and insert the element into the priority queue. Since that the priority queue follows the properties of a binary tree, Insert requires $O(log\ n) + 1$ operations where n is the number of node in the $Q[i]$th priority queue. Operations required to delete the node with the lowest value is similar to the insert functions with the difference that the node will be removed from the priority queue with the smallest value. Operations required for the delete-min operation in this data structure is $O(log\ n) + 1$ where n is the number of nodes in the $Q[i]$th priority queue and plus one is the fixed operation of choosing the ith priority queue.

2.3.7 Heap On Top Priority Queues

A combination of Multi-level bucket data structure of Denardo and Fox [8] with binary min-heap data structure was introduced to improve the performance of Dijkstra's algorithm which is called heap on top priority queue [5]. Hot priority queue, is a data structure combination of a heap H and the k-level bucket data structure B. Elements of this data structure are either a member of H or B. Size of the heap section is finite and is set to a n element which we assumed that is defined by the developer based on the right circumstances since that the author did not mention a method of calculating the n. Buckets are required to have a fixed size of K as well, and they are unsorted arrays that are keeping the node with a specific value range. Since that H is following the properties of a min-heap, Inserting a new node into the H require $O(log\ n)$ where is n is the number of nodes currently in the heap section and if the value of the inserted node is outside of the specific range, insert requires $O(1)$ operation to complete the insertion process. Removing a node from the Hot queue requires $O(log\ n)$ operations where n is the number of nodes in the H. If the heap section is empty, the algorithm uses the next unsorted bucket and convert its members to a min-heap which it requires $O(log\ k)$ operations to create a new heap where K is the number of the elements in the current bucket. Contain operation requires $O(n)$ operation to traverse the data structure to find the node. Update method needs $O(n)$ operations to find the node and If the node is in the first bucket, it will be moved to

the heap section and it requires $O(log\ n)$ operation to locate it in the correct position. If the node is in ith bucket, the node will be removed and it will be inserted into the correct bucket.

2.4 Summary

Each data structure that was mentioned in this chapter has its own advantages and disadvantages. Based on the information gathered we could report the following table (Table. 1) as a mean of comparison for these data structures in the primary four operations needed for A* algorithm to operate.

Data Structures	Insert	Remove Min	Contains	Update
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	$O(log\ n)$	$O(log\ n)$	$O(n)$	$O(n) + O(log\ n)$
Fibonacci Heap	$O(1)$	$O(log\ n)$	$O(1)$	$O(1)$
MultiQueue	$O(log\ n) + 1$	$O(log\ n) + 1$	$O(n)$	$O(n) + O(log\ n)$
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

Performance of these data structures under the right circumstance is different. Since that search algorithms belong to the NP-hard family problems, size of the problem has a direct impact on the complexity of the solution. Performance of these data structures might not be significantly different in small size problems. Another important remark is the use of these data structures in the game industry. Although that Fibonacci heap is better in terms of time complexity than the min-heap, due to outstanding implementation challenges most of the games use min-heap instead. Hash tables offer remarkable performance as well, but developers often do not use

them since that it requires hashing operations and prevent any key collisions. Heap on top priority queue it could perform better under the right circumstances but since that the author's explanations were not clear we did not use it in our research. In our study, we proposed our implementation of the Hot queue as cached min-heap and also proposed our data structure partial min-heap and compared their result to the binary min-heap and the unsorted array.

CHAPTER 3

Cached Min-Heap and Partial Min-heap

3.1 Motivation

Based on our studies in the field of pathfinding we explored multiple solutions regarding the A* search algorithm performance. Some of the solutions mainly focused on the representation of the map, and they suggested that by decreasing the size of the map and partially representing the map can increase the performance of this algorithm. Some of the algorithms suggested different solutions based on single agent or multiple agent pathfinding solutions. The operations that are related to the *Open set* are the most resource consuming operations of this algorithm. Researchers introduced a number of solutions for this problem, and each of these solutions relates to a type of problem. Data structures such as the Hot queue, MultiQueue, and Multilevel buckets suggested that by distributing the load of nodes into multiple sections and data structures, our algorithm performs more efficiently which this idea motivated us to exploit this research. Our experiments showed us that in a normal pathfinding problem in a large fully connected graph, using an admissible and consistent heuristic, the majority of the nodes inserted into the *open set* are not a part of the solution, but the data structure performs a vast number of operations to keep them.

In our research, the agents were allowed to move in any direction with the set cost of 10 for non-diagonal and 14 as the diagonal movement. The heuristic function used was the Manhattan distance, and the white space is representing a traversable node.

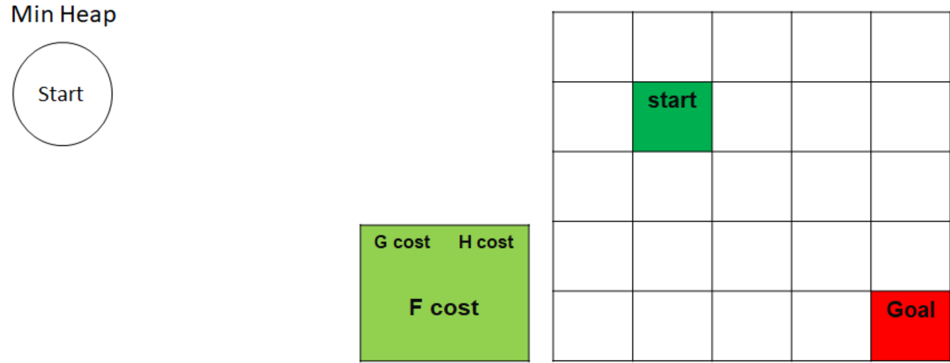


Fig. 16: Example of search environment

The green node is the start location and red is our goal location. Each cell has three values written inside which the top-left value is the G cost, top-right is the H cost, and the bottom value is the summary of given values as F cost as given in the Fig.16. In the following examples, nodes in light green color are representing the nodes in the *open set* and the nodes in blue are the nodes in the *closed set*. In our experiment we used a min-heap to implement our *open set*.

In the first iteration, our A* search algorithm extract our *start* node from the *open set* and explore its children and insert them into the *open set*. The algorithm then extracts the node with the least f(Node A) cost and explore its children.

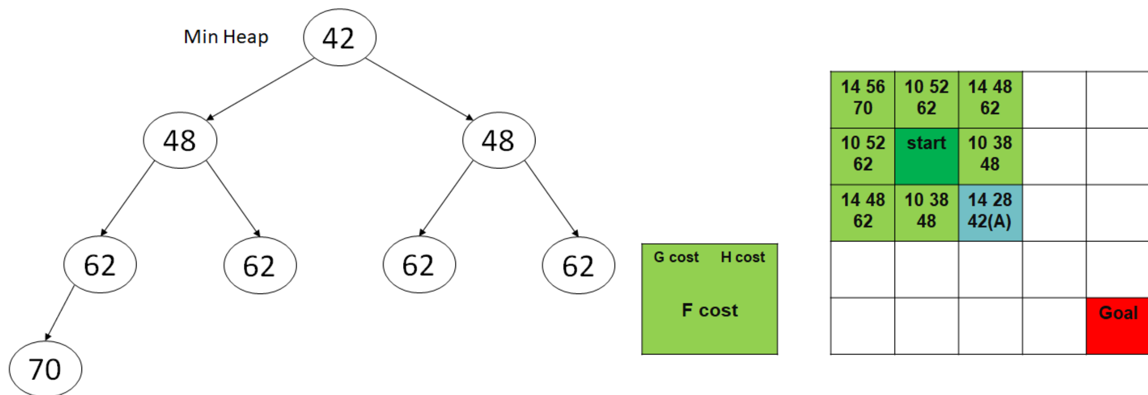


Fig. 17: Example of search environment (Exploring a node)

A* algorithm inserts the children of Node A into the *open set* and then extract the node with the least cost till it finds the correct path. If we look at the members of the min-heap (Fig. 18) before finding the goal node, we can make an important

observation. The f value of the nodes that directed us to the correct path was 42, and they did not change in the pathfinding process.

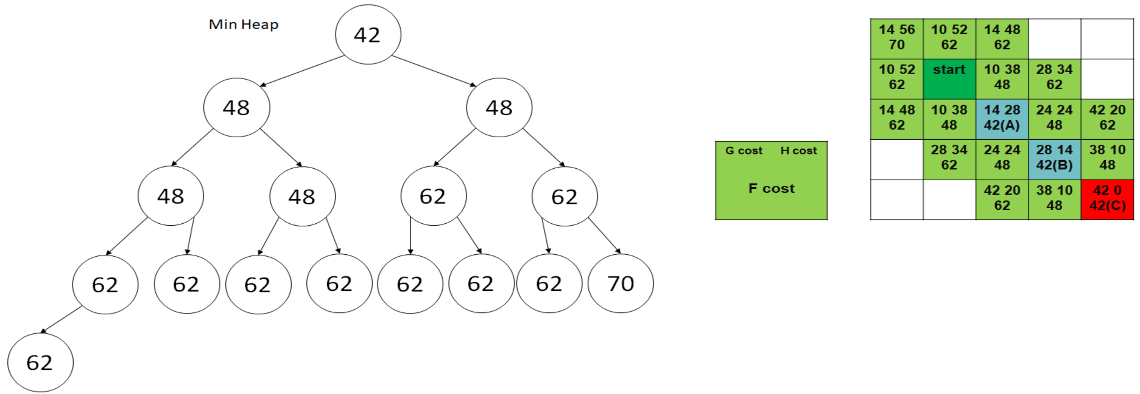


Fig. 18: Example of search environment (Exploring a node)

Data structure before finding the goal node has 16 members. Most of the members in the current heap were not explored, but the data structure kept them as members. Since that these nodes are members of this heap, the current depth of this min heap is 5. In case that we want to remove the minimum node from this data structure, min-heap will return the current minimum node, and it will replace it with the last node in the heap, and perform the required operations to find the correct location of it.

Since that the current heap has 16 members and the time complexity of the min-heap is $O(log\ n)$, removing the minimum node requires four operations. In a pathfinding problem, children of the current node have the same f value or $f + d$ values where d is the distance of the current node to the next node if our heuristic is consistent. Based on the aforesaid, binary heap expects values smaller or relevantly small values in comparison to the nodes at the bottom part of the heap. In the following example, you can see that the nodes at the bottom of the heap have values of “62” whereas the nodes inserted have values of “48” and “42”. The worst case operation for inserting a new node into the heap is $O(log\ n)$, and in this example, it requires “4” operations to locate the newly inserted node into the heap.

Based on the given example we can observe that:

- The pathfinding data (Inserted nodes and Remove min) in each iteration result

in the worst case complexity of the min-heap which is $O(log\ n)$.

- The majority of the inserted nodes are irrelevant to the pathfinding solution. These nodes only increase the size of the heap, hence expanding the time complexity of the algorithm.

It is clear that if we decrease the size of our data structure, we decrease the time complexity of our min-heap and increase its performance. Based on our second observation we came up with a theory that our pathfinding algorithm might be successful to find the correct optimal path if we limit the memory size of our data structure. Based on our observations and the concept of multiqueues combined with multi-level bucket data structure, we came up with two new data structures to implement the *open set* for the A* search algorithm, Cached min-heap and Partial min-heap.

Our cached min-heap divide the data structure into logically two sections in one array, where each element of the array has an index which we took the original idea from the heap on top priority queue. The cached min-heap first section follows the properties of a min-heap and the second section is an unsorted array. The first section of this data structure serves the A* algorithm till there are no new nodes inside, If this section is empty, then our algorithm will use quick sort to build the heap section using the members in the reserved section. Our partial min-heap is an optimistic min-heap data structure which helps the A* algorithm to find the solution performing fewer operations since it limits the search space by eliminating nodes that have a higher f value. Both data structure are explained in detail in the following sections.

3.2 Cached Min-Heap

Based on the idea of Hot queues and Multilevel buckets, we proposed a data structure called Cached min-heap. This data structure logically separates itself to two sections (Fig.19). The first part of this data structure follows the properties of a min-heap and the reserved part is an unsorted array. We assigned an index to each element in the data structure to maintain their location. We separate the cached section and the heap section using a value of d which is the depth of our data structure. The number

of elements in the heap is $2^{d} - 1$ elements.

We designed cached min-heap in a way that the elements inside the heap always have values smaller than the reserved section. To achieve this goal since that we want our algorithm to function correctly, we keep a pointer to the minimum element in our cached section. Also, we keep a pointer to the maximum node in our heap section as well.

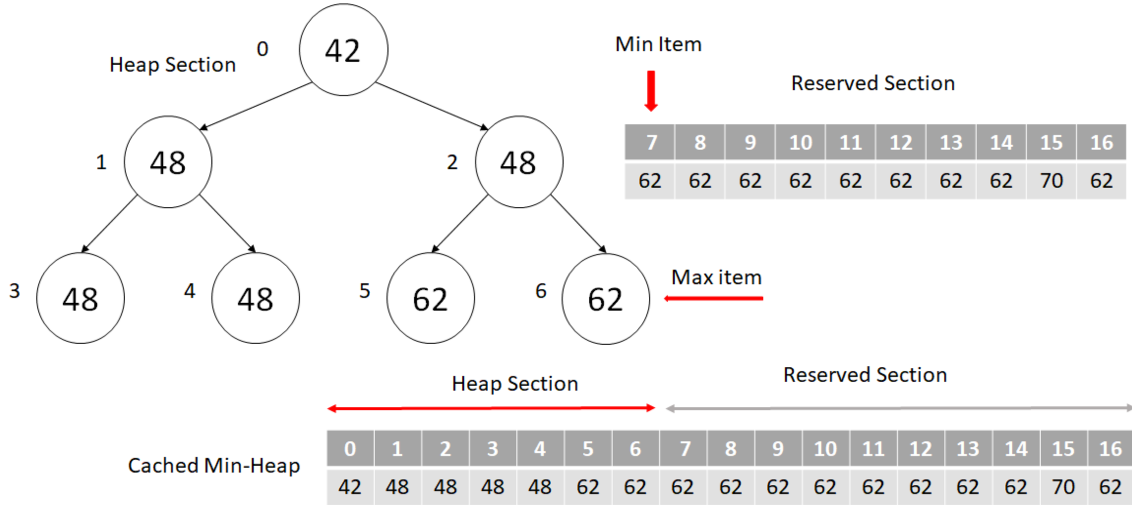


Fig. 19: Cached Min-Heap Data Structure d=3

3.2.1 Cached Min-Heap Operations

Cached min-heap is designed to improve the performance of A* algorithm data structure and satisfies the main four operations required. In the following we explained and analysed the required operation which are Insert, Remove-Min, Contains, and Update based on n where n is the number of nodes in the data structure.

Insert

For Inserting an element, cached min-heap follow certain steps to insert the new node into the data structure. If the heap has empty space and the reserved section is also empty, the new nodes will be inserted into the heap section. If the inserted node has a higher f value, the max pointer will be updated to the new node. For example

in Figure 20, inserting the new node with the value of 62 the node will be inserted to the last space available in the heap section. The new node has a higher value in comparison to its parent, so it is in the correct location. Since that the new node has higher f value than the node with the index of "4" (old max node), the *max* pointer will be set to the new node in the heap.

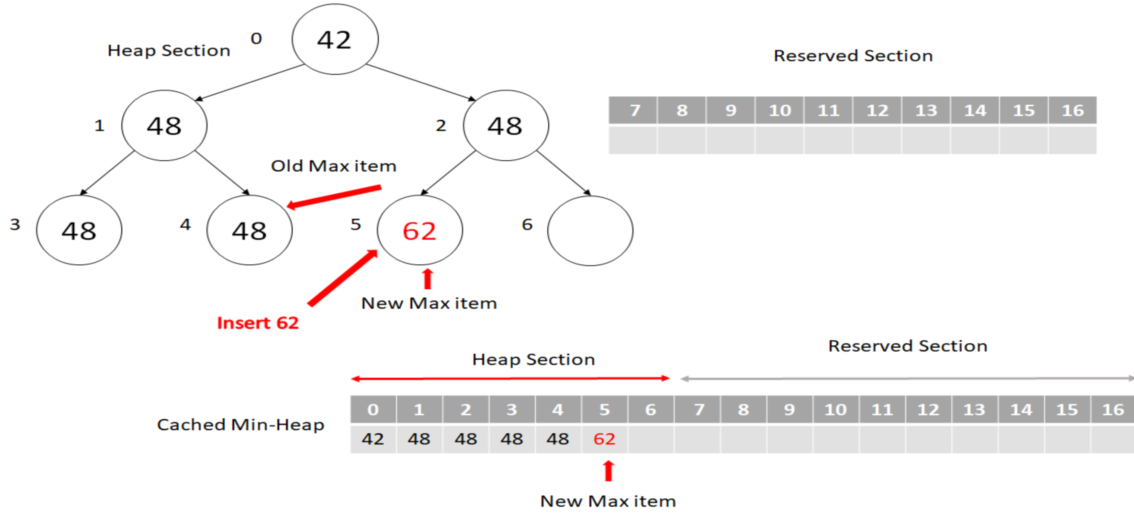


Fig. 20: Cached Min-Heap Data Structure Example d=3

If the heap has empty space but our reserved section has elements inside, the data structure checks if the new nodes f value is smaller than the current *min* item in the reserved section. If it has smaller value, the node will be inserted into the heap section. Otherwise, it will be inserted into the reserved section. For example in the Figure 21, the reserved section has "70 & 80" as members and 70 is the *min* member of our reserved section and our heap section has empty space. The algorithm will check the new member f value and compare it to the *min* member in the reserved section. Since that it has a higher value than our current *min* the node will be inserted to the reserved section.

If the heap is full and we are inserting a new node, the data structure will compare the current nodes f value, if it is smaller than the current *max* node in the heap section, the new node will replace the current *max* node in the heap and then the old *max* will be inserted into the reserved section. Since that we replaced the old *max* node, the algorithm will scan the leaf nodes in order to find the new *max* node

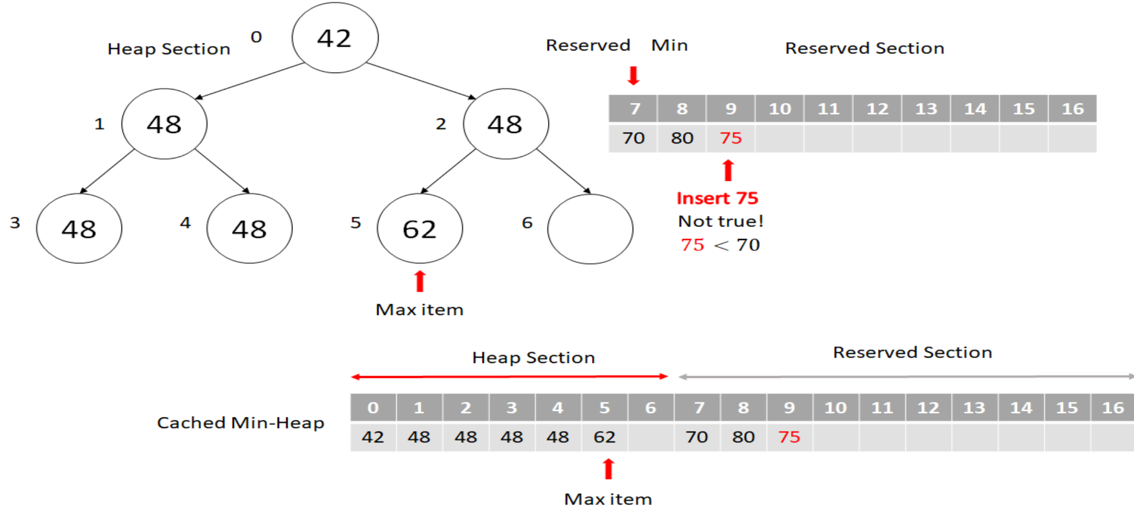


Fig. 21: Cached Min-Heap Data Structure Example d=3

and change the pointer to that node. For example in Figure.22, the new node has f value of "48" which is smaller than the current *max* node with the value of "62". The algorithm will replace the new node with the old *max* node and insert the old *max* into the reserved section.

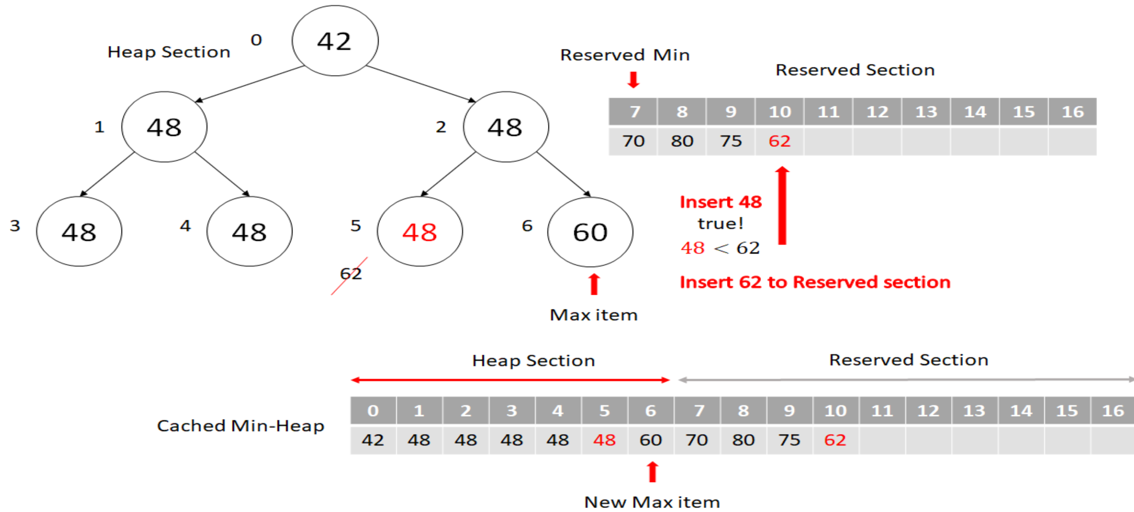


Fig. 22: Cached Min-Heap Data Structure Example d=3

In this example (Fig. 22), the algorithm will check if the newly inserted node into the reserved section is smaller than the current *min* or not if it is smaller it will replace the current *min* with the newly inserted node. Since that we replaced the old *max* node, the algorithm will scan the leaf nodes to find the new *max* node which is

the node in the index "6" in the heap section and change the *max* pointer to the new *max* node.

Since that cached min-heap data structure follows the mentioned steps to insert a new node, it is guaranteed that at any stage, the element of the heap section has smaller f value than the reserved section. The time complexity of our insert operation is $O(log\ n)$ if the heap has empty space, $O(1)$ if the heap is full and the node has higher f value than the *max* value in the heap. If the heap is full and the new node has smaller f value, inserting the node needs $O(log\ n) + 1$ operations to replace the node in the heap and place it in the reserved section and it needs $O(n/2)$ operations to find the max node in the leaf nodes of the heap section.

Remove Min

Cached min-heap requires $O(log\ n)$ operations to remove a node from the heap section since it follows the properties of a min-heap in the top section. In case that the top section of our data structure is empty, our algorithm will perform the Quicksort algorithm on the reserved section and replenish our heap section using the nodes from our reserved section. We understand that this operation is computationally heavy and requires $O(n\ log(n))$ average operation to sort the data and insert them to the heap and then it needs $O(log\ n)$ operations to remove the minimum node. A* algorithm usually inserts more nodes and extract less number of nodes during a pathfinding problem, and if our algorithm requires to perform this operation, the performance will suffer heavily. We expect that this operation does not happen many times during a pathfinding solution.

Contains

Elements in the data structure are representing objects of a node. During the operations of the A* algorithm, if the algorithm calls the contains function, the data structure is required to scan the entire entity to find the same node to find a duplicate node. Since that we are using a single array to implement our cached min-heap data structure, it requires $O(n)$ operation to check the data structure to find a duplicate

for the new node. For example in Figure.23 the algorithm needs to scan each element at the indexes to find out the new node is a duplicate node or not.

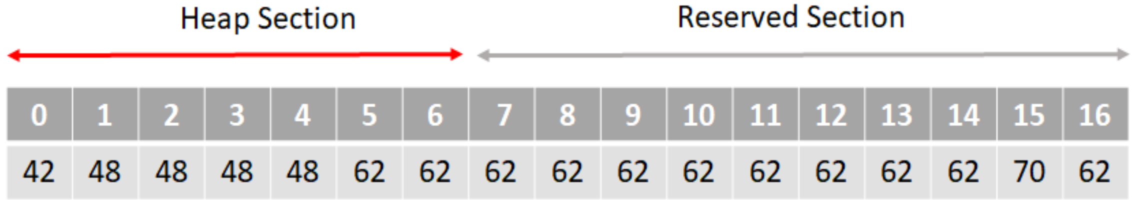


Fig. 23: Cached Min-Heap Data Structure Contains

Update

In our cached min-heap data structure we logically separate the data, so the nodes are either in the heap section of data structure or they are in the reserved section of our data structure. First, we need to locate the node in our data structure which is the same operation as the contains operation which requires $O(n)$ operations. If the located node is in our heap section, we treat it as a min-heap operation which requires $O(log\ n)$ operation to find the correct location for the new node in the heap. If the node is located in our reserved section, we first change the f value then compare it with the max node in our heap section. If the new f value is smaller than the current max, we replace the max node with the current node and check to find the new max value in our heap section. Since that we inserted a new node into the reserved section we check if the new node is smaller than the current min or not, if it is we update the new min pointer to the new min node.

Replenish Cache

During a pathfinding problem, if our heuristic function is consistent and our search environment is clear of obstacles along the way, the expanded nodes of the current node are guaranteed to have smaller f values. If our search environment contains multiple path blocks along the optimal path suggested by our heuristic, the nodes along the way are mostly won't lead us to the shortest optimal path. Aforementioned

will result in inserting nodes with a higher value than the current *min* node and performing more remove-min operations from the heap section and inserting a node in the reserved section. Since that this series of operations are a regular part of a pathfinding problem, we implemented our method to refill the cache section of our data structure. Our first solution to this problem was to perform a build-heap operation which only takes $O(log\ n)$ operations. After implementing the replenish cache method using heap sort we noticed that of our A* algorithm is not functioning correctly. We noticed that using the heap sort to replenish our cached part and slicing our data structure to two sections is not guaranteed to work correctly since we might have a node with smaller values in the reserved section that they belong to the heap part. You can see in the given example below (Fig. 24) that slicing the data structure to a depth of 3; we will have nodes that are smaller in the reserved section, so we decided to use the Quicksort algorithm to implement our replenish cache method.

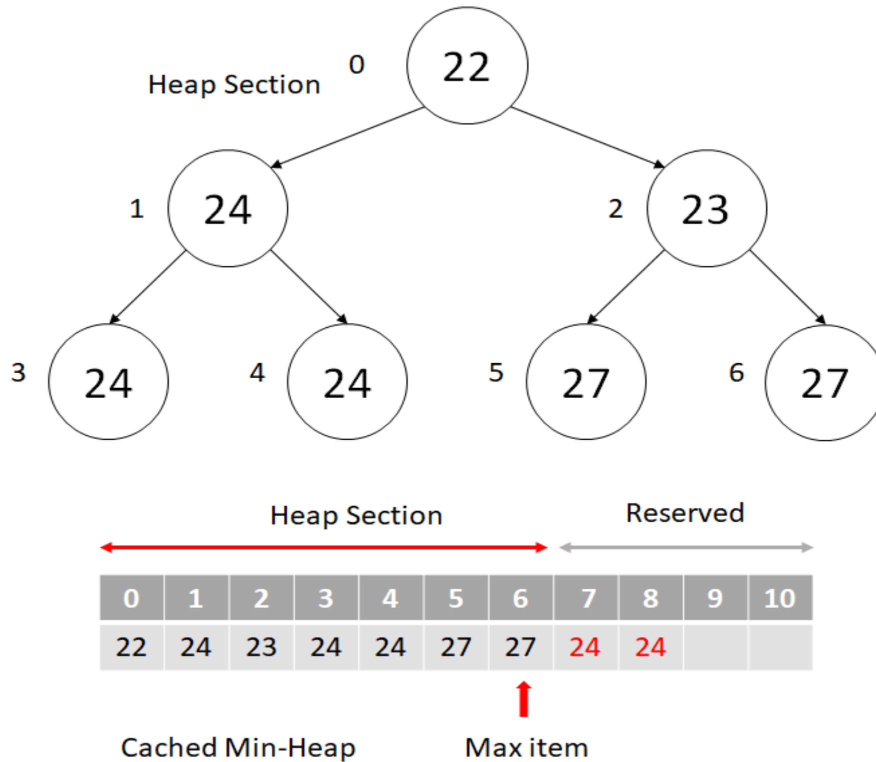


Fig. 24: Cached Min-Heap Data Structure Replenish Cache Failure Using Heap Sort

Summary

Based on our analysis of the operations of cached min-heap, we can state that the performance of this data structure depends on two main components. First is the depth of the data structure and second is the replenish cache operation. If the depth of this data structure is small, the data structure will save some operations since that the size of the heap is smaller than a min-heap but the numbers of calling the replenish cache will be increased.

3.3 Partial Min-Heap

Based on our experiments we observed that a majority of nodes in the data structure are not related to the path that A* algorithm finds. Our second observation was that the depth of the min-heap has a direct impact on the performance of the data structure. We proposed a new data structure called partial min-heap which is an optimistic data structure for the A* algorithm. We estimated that this data structure performs better than min-heap under the right circumstances with a high chance of successfully returning an optimal path. Same as cached min-heap we assigned an index to each element in the data structure, and we also set a maximum size for this data structure as well. The maximum number of elements in this data structure is $2^{\mathrm{d}} - 1$ where d is the depth of the data structure.

3.3.1 Partial Min-Heap Operations

Partial min-heap is an optimistic data structure which aims to limit the search space of the A* algorithm while following the properties of a min-heap. This data structure performs the main operations required by the A* algorithm. In the following section, we explained and analysed all four main operations of the partial min-heap.

Insert

Since that this data structure only contains a limited number of nodes at the time, First the algorithm checks if the data structure is full or has empty space. If the data structure is empty, the first inserted node will be inserted into the heap, and its value will be set to the *max* value and the index of that node will be saved. If the data structure is not full, the nodes that the A* algorithm insert them into the heap will be added to the partial min-heap same as the min-heap. In each insertion operation, we check if the current node f value is higher than the current *max* node. If it is higher, after that we locate the correct position of the node, we will set the pointer of the new *max* node to the newly inserted node. If our data structure is full, we compare the f value of the inserted node to the current *max* nodes f value. If the new node has a higher f value, we discard the new node and if it has a smaller f value compare to the current *max* node, we replace the new node with the current *max* node's location which is $O(1)$ operation to replace the node and $O(log\ n)$ operations to relocate the inserted node to its correct location. Since that we replace the *max* node, we need to find the new *max* node in the heap so we scan the leaf nodes to find the new *max* location which requires $O(n/2)$ operations.

For example in the Figure. 25, the new node has a f value of "48" which is smaller than the current *max* node with the value of "60". The algorithm will replace the new node with the old *max* node and discard the old *max* node. Then the algorithm will scan the leaf node in the current heap and find the new max item which is the node in the index "3".

Remove Min

Partial min-heap requires $O(log\ n)$ operations to remove a node from the heap section since it follows the properties of a min-heap. Same as the min-heap the top node will be extracted, and it will be replaced by the last node in the heap. Then it will be compared to its children and swap its location if it is smaller until it finds the correct location in the min-heap.

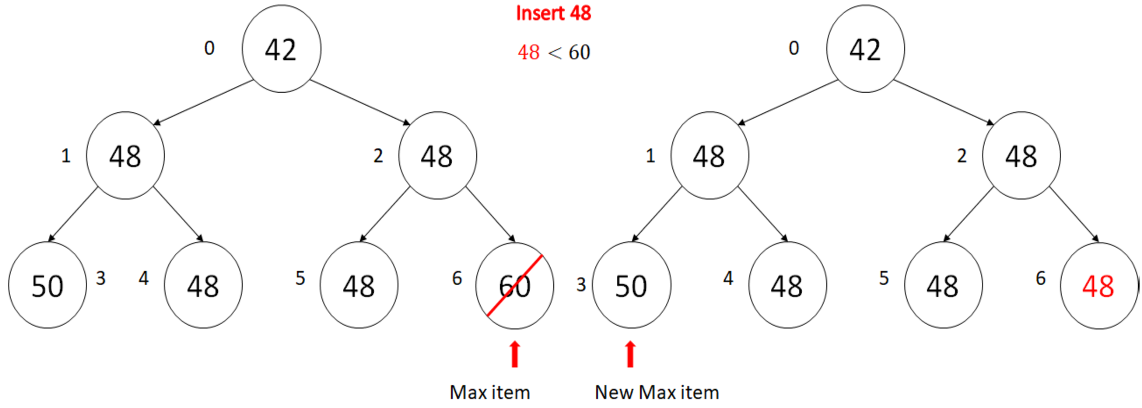


Fig. 25: Partial Min-Heap Data Structure Insert Example d=3

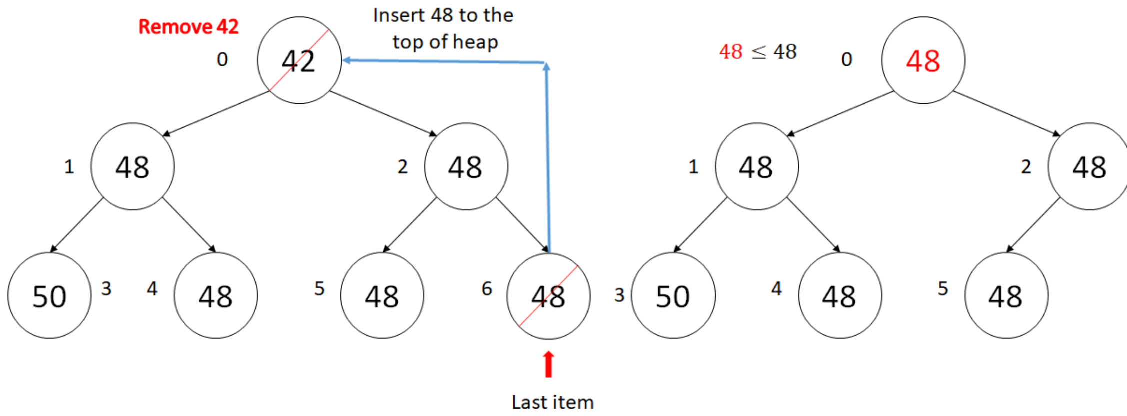


Fig. 26: Partial Min-Heap Data Structure Remove Min Example d=3

For example in the Figure. 26, first partial min-heap extract the minimum node with the value of "42", then replace that node with the last node in the index of "6" in the heap. Then compare its value with its children, since that the node has a value equal to its children no operation is required, and the node is in the correct location.

Contains

The same as the cached min-heap data structure, elements that are members of the partial min-heap are objects of type node. To locate a node in the data structure, we need to compare the node in the matter with each node in the heap to ensure if the node is already a member of the data structure or not. This operation takes $O(n)$ to scan the data structure.

Update

Scanning the entire data structure requires $O(n)$ operation to find the node that we need to update, then we change the f value of the updated node. Since that the new value might not be in the correct position in the heap and violate the properties of the min-heap, we need $O(log\ n)$ operations to locate the updated node in its right location in the heap.

3.4 Partial Min-Heap Case study

Partial min-heap is designed to decrease the size of the memory in our A* pathfinding algorithm. Since that we are techinically limiting the search space for our algorithm we will face the risk of losing the optimallity of the path or failure of finding the path.

3.4.1 Partial Min-Heap Success

In the following example, we set the depth of the partial min-heap to $d = 3$. Based on the formula that we provided, Partial min-heap at most will accept $2^3 - 1 = 7$ elements at most. Since that we limited the number of nodes so in each insert operation and remove min operation we will save extra operations compare to $d = 5$ depth. In the example given bellow we can see the final iteration of A* algorithm finding the path using a min-heap as *open set*. In this example, our min-heap accepts all of the inserted nodes. Our assumption was that using the correct d and limiting the size of the data structure will result in an optimal or sub-optimal path. If we set the limit of size of the data structure to $d = 3$ in figure 27, our pathfinding algorithm intially inserts neighbours of the starting node, since that the data structure will be filled out with the neighbours of the start node, The node with the f value of “70” will be discarded from the data structure in the first iteration.

In the second iteration shown in Figure.28, A* algorithm will chose the node “A” and explore its children. Expanding the child node, partial min-heap will discard the nodes that are already in the min-heap and replace them with higher f values and

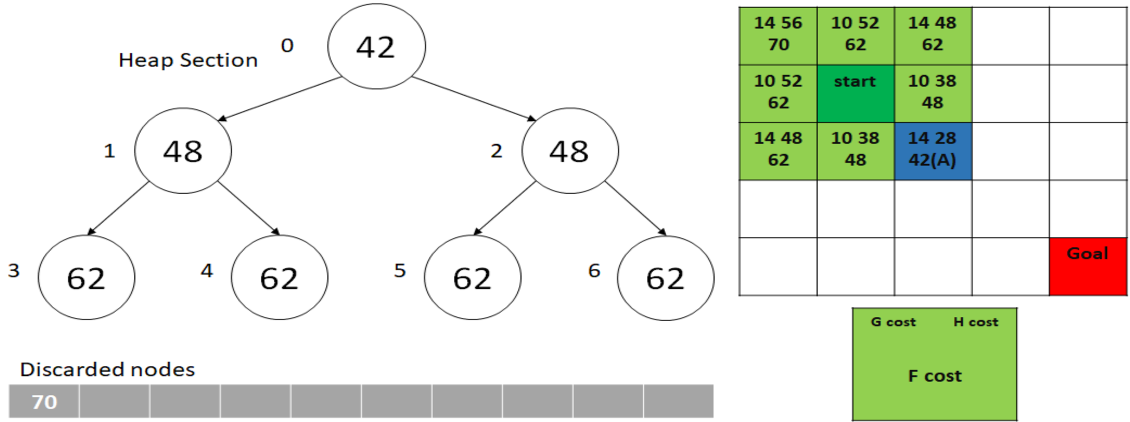


Fig. 27: Pathfinding Example

replace them with the new nodes with smaller f values.

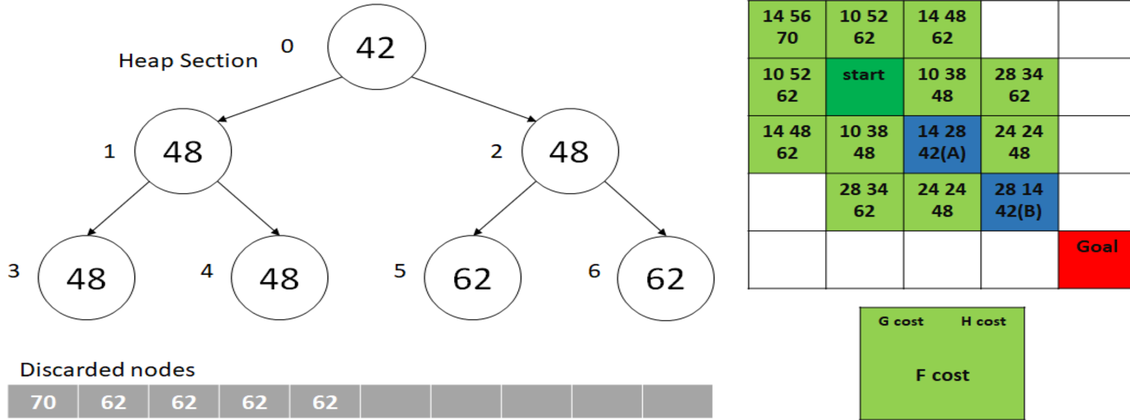


Fig. 28: Pathfinding Example

In the last iteration shown in Figure. 29 you can see that A* algorithm was able to find the optimal path using the partial min-heap data structure while discarding nine extra nodes. Our pathfinding performed less number of compare operations since that worst case for each insert and remove operation was $O(log\ 8)$ compare to the $O(log\ 15)$ operations in a normal min-heap in the last iteration of this pathfinding example. So our expectation is that the partial min-heap will let us to be able to find an optimal or suboptimal path under the right circumstances.

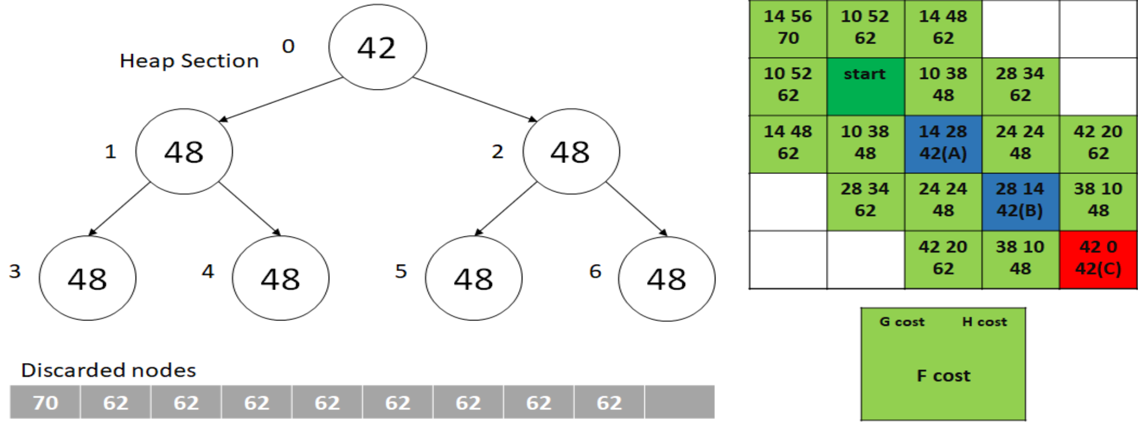


Fig. 29: Pathfinding Example

3.4.2 Partial Min-Heap Failure or Sub-optimallity

Same rule applies if we slice our data structure too small and set a small size data structure for a problem. If our data structure is too small, our A* algorithm might try to re-insert the already discarded nodes into the data structure which might result in finding a path to the goal node. In case of this issue ocurring, our solution to this problem will be suboptimal since we might have missed the branch in our graph that was resulting in the shortest path. If our data structure is smaller than a certain size or our map has large blocks along in the path, our data structure will compeletly fail to find an answer since it already discarded the nodes that will result in finding a path out of the blockage.

3.4.3 Summary

Our proposed partial min-heap is an optimistic data structure for the *open set* for the A* pathfinding algorithm. Our analysis of this data structure is that if the depth of the data structure which represents the size of it, has a crucial role in the success or failure of the pathfinding algorithm. Limiting the size of the data structure to small depths will limit the pathfinding algorithm to be able to find only small size paths. Also increasing the depth of our data structure higher than a certain depth depending on the pathfinding problem will result in changing our data structure to perform as a min-heap and it will act the same as a min-heap.

CHAPTER 4

Experiments and Results

4.1 Implementation Methods

We implemented $A*$ algorithm using $C\#$ programming language, and we used the Unity3D game engine to present our work visually. Each system and class will keep track of number of operations and time in order to enable us to evaluate performance of our algorithms. Our system is designed to find the single-source shortest path from the location of our starting node to our goal node. In each test, we change the location of our start and goal node to a walkable spot in the map.

We used the A* search algorithm as our pathfinding algorithm. In each iteration, based on the same start and goal location, A* will try to find an optimal path using four different data structures that we implemented. To have consistent and fair results in each test case, we gathered the results using all four data structures. In this class we stored primary attributes in each test case such as Type of data structure, success, number of operations, time, cost of the path that was found, chance of obstacles,etc. Since that the focus of our research was improving the performance of the *open list*, we implemented the close list as HashSet which contains object type of class Nodes.

The data structure that we compared our new algorithms cached min-heap and partially min-heap is the unsorted array and min-heap. Due to the nature of A* algorithm, while loop continues its operations till there are no other nodes to expand. We implemented the unsorted array using a List that contains object type of nodes. Using an array we manipulated a min-heap in a way that each node in the data structure has a property of heap-index. By using this property, we maintained the

structure of a tree and a min-heap. Our new approach to min-heap, partial min-heap and cached min-heap was implemented using the same technique. Cached min-heap logically separates inserted data into a min-heap and an unsorted-array. Partial min-heap follows the property of a min-heap but logically decides if the inserted data might be valuable to the algorithm or not.

4.2 Experimental Environment

Our data structure has no limitation for any search space or any condition. Since we wanted our work to be clear and useable by other researchers we implemented our search space using a fully connected squared-grid map. Based on the suggestions of [23] we used specific map sizes, and we set the Euclidian distance as our heuristic function. Agents were allowed to move in 8 directions(diagonally and non-diagonally). Cost of diagonal movements is 14, and the non-diagonal movement is 10. We generate each map with a pseudo-random percentage of obstacles, ranging from 0 to 55 percent. Typically, the possibility of finding a path is slightly less when the chance of having obstacles is higher than 45 percent.

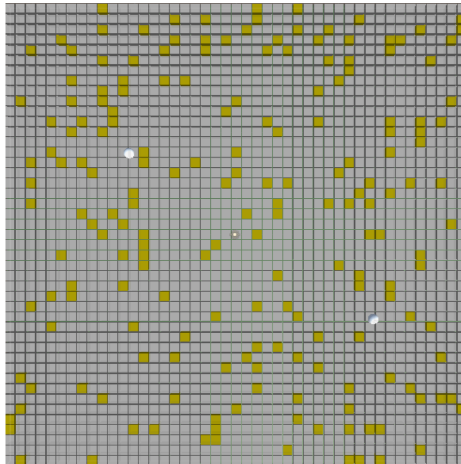


Fig. 30: Map Size 45*45 - Obstacle chance 10%

We ran our experiments on conventional map sizes that were used by commercial games such as Warcraft III and Dragon Age: origin. Our map sizes varied from 120*120 to 512*512 nodes which is actually the same size as Warcraft III map size.

In each experiment, our system generated a new map, placed obstacles based on the obstacle chance. Then our program randomly changed the location of our pathfinder agent and goal location to a walkable node on the map. Then our pathfinding algorithm will use these tools to run the A* algorithm separately using our implemented data structures. In each test case, the chance of obstacle will increase by 5 percent after every 100 iterations. We defined each iteration as 100 different pair locations in each map size with a certain chance of obstacles. Our test method will automatically increase the obstacle chance, and repeat the algorithm for each map size.

4.3 Experiment Setups

Our test method was implemented to function with minimal user input. The system generates parameters such as obstacle chance and distribution. First, we set the plane size to our test scenario in Unity3D which help us to test our algorithms with commercial game's map sizes. Our Grid script scans the plane to generate a graph of nodes which are used by our A* script to use it run the tests. We implemented our benchmarking system in a way that user can insert obstacles on the map manually as well in case that they wanted to experiment with different ideas.

We also implement a Unity GUI for our benchmark system in case that user wanted to add new data structures or in case that they wish to run multiple tests with only some of the data structures implemented, not all of them. Due to the implementation of our newly developed data structure, the user must define the depth of the partial min-heap manually in each test. It is the same case for our cached min-heap data structure as well, which requires user input to logically separate its min-heap and the unsorted data structure from each other. We set a depth of our partial min heap to 8 (2^8 - 1 nodes) by default in case that the user forgot to define this value, preventing the test to fail. Also, we set our default size of our cached min-heap to a depth of "8" and depth "8+3" for the cached section to prevent any user input failures.

So in an experiment, the user can define the number of iterations, size of the map, the chance of obstacles, showing the path found using different data structures, type

of the data structure, number of iterations. In order so that user can have more data from the test scenario we implemented the "Map" method. This method is useful since it gives the user the ability to import a specific map to experiment with or export the existing map using a text file.

4.4 Performance Evaluation

In each iteration, we gathered information that our program reported us to compare the performance of these algorithms with each other. Since that partial min-heap is not guaranteed to find the path, we collected that if the A* was successful to find the path, the time is taken to find a path, path length, cost of the path, nodes expanded, and operations in each test scenario for each data structure.

4.4.1 Success

Our proposed partial min-heap data structure is a greedy solution to the A* star algorithm. Our experiments showed us that under the right circumstances our algorithm performs better than min-heap, but in cases that the partial min-heap does not have the correct depth tailored to the search space, this algorithm will fail to find a path. To keep track of this result, we made sure that in each iteration how many times partial min-heap was successful and how many times it was not.

4.4.2 Time

We defined a stopwatch in each method to calculate the time in each iteration. We used the "System.stopwatch" to measure the elapsed time. Our stopwatch starts measuring the elapsed milliseconds after that all of the classes are initiated. We stop the stopwatch once the pathfinding method returns a path or the open set is empty which shows that the algorithm failed to find the path. We also kept track of all the iterations in parallel to the ones that were successful, to have a fair comparison.

4.4.3 Path Length and Cost

Since that our main test method places the starting node and the goal destination randomly on the map, we wanted to keep track of the path length to compare our results with other data structures. The second reason for keeping the path distance in each iteration was that our partial min-heap might result in finding a sub-optimal path. Since that this algorithm only keep track of nodes that have a higher chance of resulting in finding a path, it might result in finding a suboptimal path with a smaller number of operations. For example, the path that min-heap will find has smaller F cost, but the number of operations is higher, but partial min-heap will find another path which might have higher Fcost but in less time and number of operations. We kept track of each data-structure path to make sure that our algorithms will find the same path as well. We calculate the path cost as well by adding up the node's Fcost in the path to measure the path cost.

4.4.4 Nodes Expanded

Our implementation of A* algorithm works using a while loop and perform by first in first out. A* algorithm will extract the first node in the open set and perform the operations needed to find the path to the goal node. Since our algorithms use different properties to determine the best node in the open set to the algorithm, we kept track of the number of nodes expanded in each iteration so that we have an extra point of measurement.

4.4.5 Operations

Based on our implementation of A* algorithm and using Unity3D as our game engine, we considered to keep track of number of internal operations that each data structure performed to maintain the properties of a min-heap. In each class of data structure implemented we kept track of number of Inserts, Removes, Sort-downs, Sort-ups, Contains, and Update-Items. Based on the architecture of partial min-heap we counted special operations that this data structure performs which is swap oper-

ation. Cached min-heap logically separates inserted data into two different sections and keep track of maximum node in the current min-heap and minimum node in the cached section. If extras are full it will add nodes to the cache section and if the heap is empty it will use quick sort to replenish the heap section. In different cases based on the Fcost of the inserted node, different operations take place as well. During the process, we counted Cached Inserts, Swaps, and the number of operations needed in case of the Replenish-Cache method required to maintain a min-heap, including operations needed in the quick-sort algorithm.

Unsorted Array

In our unsorted array data structure, anytime the algorithm inserted a new node to the open list we count it as one operation. Removing the minimum from the unsorted array takes n number of operations since that it will compare the F cost of each node to one another to find the least F cost and counted it as one operation per each comparison. Contains searches the open set, comparing the new node with the nodes that are already in the open set. We counted each comparison as one operation. If the node's F cost needs to be updated, since the data structure is unsorted no operations will occur and we did not count the update-item as an operation.

Min-heap

Insertion and Removing first item in min-heap takes $O(log\ n)$ operations. Nodes that were inserted in our min-heap take a property of heap index. Using this heap index we maintain the properties of a binary min-heap. Each insertion count as one operation since it needs to get the properties of a heap. The algorithm compares the inserted item's value to its parent's value, if the added item is smaller then the item's swap location with its parent until the item is in its correct location in the min-heap. Removing the first item returns the first item and replace the last item in the heap with the location of the minimum item. Then compare it with its children and in case of higher f cost item need to be swapped. We counted each operation and comparison as one operation. Update-item and Contains methods need to scan

the min-heap to find the item and compare the items with each other, so we counted each comparison as one operation. In case of update item, sort-up or sort-down might be required, so we counted the required operations as well.

Cached Min-heap

We divided our data structure section into a min-heap and an unsorted array and created a new data structure named Cached min-heap. This data structure follows the properties of a min-heap in the top section of data, but keeps the rest of the data as well in case that it might be required to finish the pathfinding algorithm. Adding a new node into the cached min-heap takes certain amount of comparison and operations. If our heap section is not empty, nodes are inserted into our heap section without any hesitation and comparisons which is $O(log\ n)$ operations, the same as a min-heap, but we keep track the maximum node in the heap. If the heap section is full, new nodes are compared to the maximum member of the heap. If they have a smaller F cost than the maximum node, they replace the current max. Since we replace the maximum node, we run the find-max method to find the current max in the tree. Find max searches the leaf nodes to find the new maximum. This operations takes $(n-n/2)$ where n is the number of current nodes in the tree operations. Cached-insert operation takes $O(1)$ operations since it is an unsorted array, and we keep track of the Minimum node in the unsorted array as well. If the new inserted node has a smaller f cost than the current minimum in the cached section, we insert that node into the extras and that takes $O(1)$ operation. Removing the first item is the same as a min-heap which takes $O(logn)$ operations. We implemented our update item same as the min-heap with a different condition. If the node that needs to be updated is not in the tree section, we check with the maximum node's f cost. If it has a smaller f cost, we will simply replace that node since that it has a higher chance of getting used by the pathfinding algorithm.

Assuming that our min-heap section is empty, and we have nodes present in our cached part, we will run the Replenish-Cache method. This method first sorts the data in our cached section using the "quick-sort" algorithm which takes $\mathrm{O(N^2)}$ worst

case operations and inserts the nodes in the tree.

Partial Min-heap

Partial min-heap is our greedy proposed solution to solve the A* pathfinding algorithm. Partial min-heap implementation has the same operations as a standard binary heap, but it is different in the insertion process. Since we have a limited amount of memory in our implementation, we keep the maximum node in our min-heap. When we run out of memory, which means that our heap is full, we check the f cost of the current node; if it is smaller than our current max, we discard the current maximum node and replace it with the current node. After doing this operation, we will run the Find-Max method to find the current maximum node which takes $O(N^2)$ of operations.

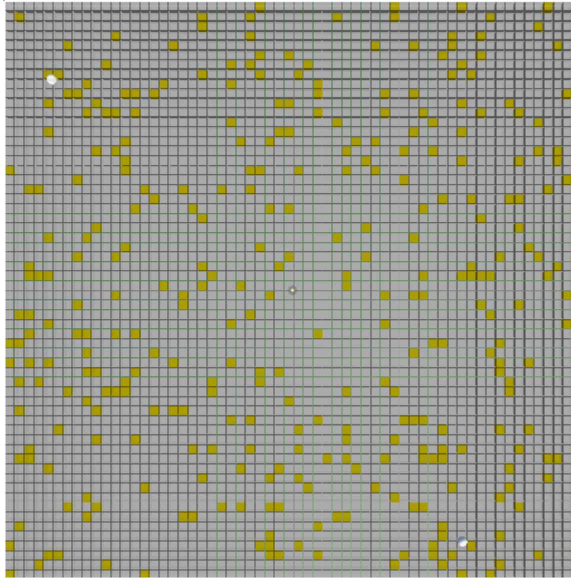


Fig. 31: Map Size 60*60 - Obstacle chance 10%

4.5 Experiment Results and Analysis

Using the benchmark system that we implemented, we ran tests and changed multiple variables in our iterations to understand how our proposed data structure performs in different situations. Since that we use random locations in our map to run our

pathfinding algorithm, we did not have control over the path length, but we found out that for each certain map size, the depth of our partial min heap which we use K to represent it in our graphs, had better performance. We ran over 260,000 experiments using our implemented data structure to find out, under what circumstances our proposed data structure perform better or worst than other existing solutions.

In our experiments we used conventional map sizes which were more popular in games. Map sizes that we used were: 120*120, 200*200, 300*300, 400*400, and 512*512 nodes. We increased the obstacle generation chance by 5 percent after 800 experiments. Our obstacle chance varied from 0 percent up to 55 percent. In each experiment with the mentions map size, we generated a new map with different obstacle distribution and different locations for our pathfinding agent and goal location. Since we were trying to find the best value of K (depth of our partial min-heap) in our data structure and its effect on the performance, with each value of K we ran 9600 number of tests. After each batch of tests we changed the value of K and ran the experiments again. In total, we generated more than 62000 maps, with different obstacle chance and distribution and map size. Using our four data structures we ran A* pathfinding algorithm, 200 times. A summary of our test plan is shown in table 2.

Obstacle Chance	Number of tests	Map Size	Data Strutcure
0 5 ... 50 55	200	120*120 200*200 300*300 400*400 512*512	Unsorted Array Min-heap Cached Min-heap Partial Min-heap

Table 2: Summary of experiments

We presented our results in the following order, Based on the gathered information, we compared our data structure's performance using the number of operations and the number of times that our Partial min-heap data structure was successful to find a path using a value of K. We also completely presented our gathered data in the appendices. We presented our data in the following, a chart will present the number

of operation that it took to find a path using an unsorted array, min-heap, cached min-heap, and partial min-heap, and another chart will present the percentage of success using each data structure.

4.6 Number of operations

Results For Map Size 120 * 120

This chart (Fig 32) presented results for the map size of 120*120 nodes and using the K=6 as the depth of our partial min-heap. Based on the gathered results we found out that, using k=6 will result in finding the path with less operation but a small success rate.

Since we experienced a large number of failures by using K with a value of 6, we changed the k with a value of 7. The charts (Fig 33) show that in this environment using K=7 we saw better success rate to compare to k=6. In most cases, partial min-heap performed with less number of operations which shows that this data structure performs better compared to other data structures. Cached min-heap performed better than heap when the obstacle density was higher than 45%. In most cases, partial min-heap performed better than all other data structures with an acceptable rate of success.

In this set of experiments, we used k with the value of 8 in our partial min-heap. The charts in (Fig 34) shows that partial min-heap had a higher success rate in comparison with k=7. Results show that the unsorted array executed more operations than the min-heap. In this test scenario, cached min-heap performed more operations than the min-heap and partial min-heap. Our results show that partial min-heap had better performance than the other data structures. We concluded that using the K of 8 as the depth of our data structure, will guarantee in the best performance in this particular map size.

	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	8321	12704	16822	32395	33103	26471	58986	68712	57196	88219	88730	75180
Min-Heap	3586	4318	5504	8817	9254	8353	15700	18915	16990	24877	26826	26955
Cached Min-Heap	5252	7980	8599	17046	13671	9811	20667	21642	16534	23047	22233	19854
Partial Min-Heap	2536	2818	3466	2683	2388	2450	2686	2643	1955	5187	5296	7561

Fig. 32: Map Size 120*120 using K=6 Full Data in Table 3

Results For Map Size 200 * 200

We initially experimented on map size of 200 * 200 nodes, using K = 6 as the depth of our partial min-heap data structure, but we faced large numbers of failures as seen in Fig. 35 even though that partial min-heap performed better than min-heap and cached min-heap. We assumed the reason for such a small number of operations is that the partial min-heap was successful to find the path in small search spaces with less number of nodes.

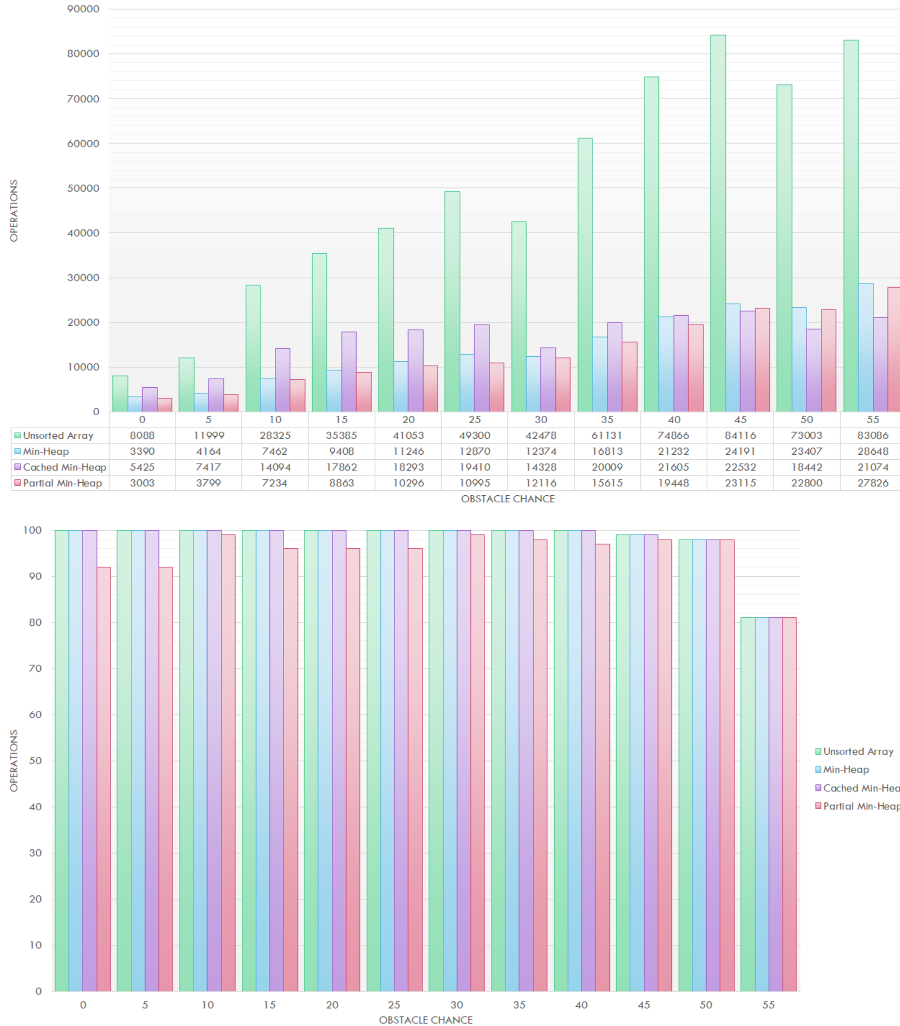
We ran our experiment and used 7 as the depth of our partial min-heap. It can be seen that the percentage of partial min-heap success increased compare to

Fig. 33: Map Size 120*120 using K=7 Full Data in Table 4

K = 6. Based on the number of operations we can state that partial min-heap still performed better than min-heap and cached min-heap in most case, but our pathfinding algorithm failed to find the path more than 50% of times.

If we increase the depth of our partial min-heap to 8, we can see that the performance of partial heap regarding operations is still better than the min-heap and cached min-heap (Fig. 37). We can see in the graph that partial min-heap executed the pathfinding algorithm using less number of operations.

Since that we still have a large number of failures, we increased the depth of our partial min-heap data structure to 9. Using k as 9, we observed that the success rate increased exponentially. Based on our results, partial min-heap with the depth of 9



	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	8088	11999	28325	35385	41053	49300	42478	61131	74866	84116	73003	83086
Min-Heap	3390	4164	7462	9408	11246	12870	12374	16813	21232	24191	23407	28648
Cached Min-Heap	5425	7417	14094	17862	18293	19410	14328	20009	21605	22532	18442	21074
Partial Min-Heap	3003	3799	7234	8863	10296	10995	12116	15615	19448	23115	22800	27826

Fig. 34: Map Size 120*120 using K=8 Full Data in Table 5

in this map size, our pathfinding algorithm performs almost as good as a min-heap with less number of operations. Our proposed data structure perform better when the chance of having obstacles is higher since it will separate the nodes with a high value of f cost apart from the other nodes.

We also increased the size of our data structure to k = 10, and we observed that our data structure would act as a min-heap since that the memory size is larger than what is required to find the correct path and The number of operations were the same as a min-heap. For this map size, our experiments brought us to this conclusion that k = 9 is the best depth for this map size.

	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	18117	44915	76725	107630	112044	201270	177557	213989	212545	261865	236216	316125
Min-Heap	5579	9728	14975	20711	21661	33465	32181	41042	43870	53531	54162	81644
Cached Min-Heap	9887	22097	33653	38525	34047	64152	47248	48354	43235	49776	42563	58423
Partial Min-Heap	3527	7195	4738	5259	4243	3234	1878	3077	3099	4289	3846	6219

Fig. 35: Map Size 200*200 using K=6 Full Data in Table 6

Results For Map Size 300 * 300

We initially started our experiments on this map size using the k = 6 and found out that our partial min-heap performed poorly. Results showed us that in the best pathfinding samples, Partial min-heap found the correct path in 15% of iterations. This indicated us that to increase the k to 7. Using k with the size of 7 for this map size did not improve the success of partial min-heap to find the path as well, and it was only successful 33% of the times to find a path. We used k equal to 8 and nine as the depth of our partial min-heap.

On average, partial min-heap success rate using k = 8 was 51% which indicated

Fig. 36: Map Size 200*200 using K=7 Full Data in Table 7

us that k = 9 could perform better. Using k = 9 as the depth of our partial min-heap result in an average of 83% percent which is only five percent less than the min-heap in this particular iteration. We accepted this k equals 9 as the best performance of depth of partial min-heap for this map size. In every obstacle chance, partial min-heap had less number of operations compare to the min-heap. Our second proposed data structure, cached min-heap performed with more number of operations in comparison to min-heap and partial min-heap in test scenarios where the obstacle chance is less. Cached min-heap executed less operations in comparison to other data structures when the obstacle chance was higher to find the correct path. We also ran our experiments using k = 10 as the new depth which turned our partial min-heap to

Fig. 37: Map Size 200*200 using K=8 Full Data in Table 8

a min-heap and the performance of our data structures were the same in number of operations. Cached min-heap still performed better in number of operations. It is obvious that unsorted array needs the most number of operations required to find the path. Based on our experiments, we found out that partial min-heap using k=9 as the depth will result in high number of success and less number of operations compare to the min-heap, which means that it has a better performance in terms of executed operations.

	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	20722	41170	79734	130150	187226	256605	253977	257925	326427	271695	266057	316936
Min-Heap	6076	9412	15367	23044	32651	43109	41511	46778	59370	55165	59746	78707
Cached Min-Heap	5748	8523	12890	19622	25419	39479	38107	38674	46443	40530	43438	56598
Partial Min-Heap	6053	9276	15209	21778	31807	39356	35909	42519	55114	53977	58351	76722

Fig. 38: Map Size 200*200 using K=9 Full Data in Table 9

Results For Map Size 400 * 400

Our test results indicated that using partial min-heap using k = 6 & 7 was not successful to find the path more than 14% and 35% percent of the times respectively. Experiments using k = 8 as the new depth for this map size showed us (Fig. 41) that although the number of operation was less than other data structures on average partial min-heap was successful only 37% of the times.

Based on our experiments, on average partial min-heap was able to find the correct path with less number of operation in comparison to min-heap with success rate of 75% (Fig. 42). Cached min-heap executed less number of operations in comparison

	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	45121	183791	208286	265384	428984	738014	730430	718476	1018335	833101	973065	754945
Min-Heap	9849	24430	28919	39382	59783	89292	92114	97009	129516	125484	154930	152589
Cached Min-Heap	17927	46077	38746	40493	63002	146637	130075	115109	150392	98108	113269	108842
Partial Min-Heap	8188	22509	19043	25227	26531	24997	24554	33941	33177	45251	65213	103852

Fig. 39: Map Size 300*300 using K = 8 Full Data in Table 10

to other data structure to find the path when the number of obstacles was higher than 40%.

We used k = 10 as the depth of partial min-heap with this map size, and our results were improved compared to k = 9 by an average of 20%. Figure 43 shows us that partial min-heap performs better in number of executed operations in comparison to min-heap in this map size with all of different obstacle densities. Cached min-heap also performed better than partial min-heap when our obstacle chance was higher than 30%. As expected, using k=10 as the depth of partial min-heap for this map size will turn our data structure into a min-heap, and the performance of our data structure will be almost similar regarding operations.

Fig. 40: Map Size 300*300 using K = 9 Full Data in Table 11

Results For Map Size 512 * 512

The number of existing nodes in a map with the size of 512 * 512 is 2^{18} nodes, which make our search space very large. Our pathfinding algorithm mostly requires to expand a large number of nodes to find the correct path. We expected the highest number of operations in this map size. We initially started our experiments using k equal to 6 and 7, which resulted in poor performance of at most 6% and 20% success in finding a path respectively. We increased our k to 8 which had adequate performance rate when the chance of having obstacles are either low or high. Partial min-heap had 45% success rate when there were no obstacles in the map with 50%

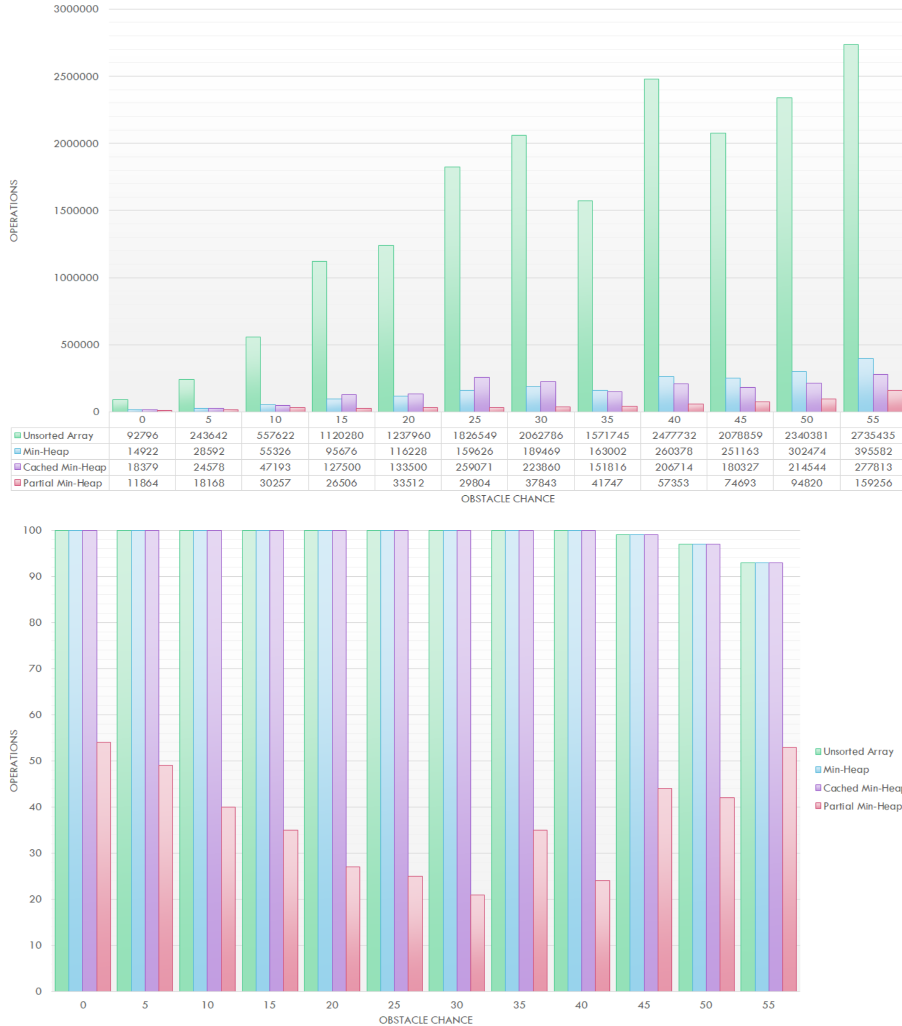


Fig. 41: Map Size 400*400 using K=8 Full Data in Table 12

less number of average operations compare to min-heap. Partial min-heap had only 25% success rate in this map size with the depth of 8.

Partial min-heap on average was 57% of the times successful to find the correct path using k equals to 9. Cached min-heap performed better in the number of executed operations in comparison to min-heap when the number of obstacles was higher than 40%. The unsorted array had the most number of performed operations in comparison to other data structure in all of the test cases.

We ran multiple numbers of test iterations using partial min-heap using k equals to 10 and 11 on this map size to get a better understanding of our data since this map size is one of the most common map sizes in most of the commercially developed

Fig. 42: Map Size 400*400 using K=9 Full Data in Table 13

games. Based on the results that we gathered, Partial min-heap using k=10 performs better in the number of executed operations in comparison to min-heap and will achieve the success rate of 89.5% to find the correct path.

Results in figure 45 show that using $k = 11$ turns our partial min-heap to a normal min-heap and the success rate of this data structure is the same as a min-heap.

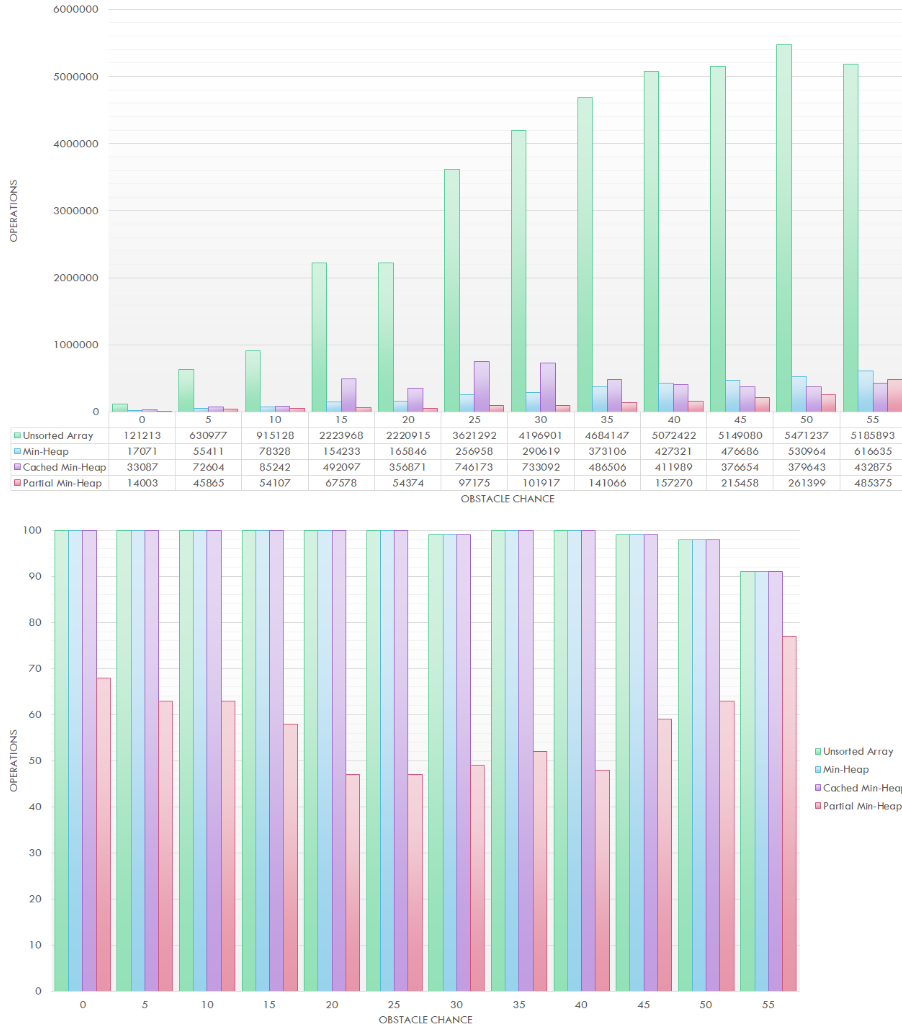
4.7 Runtime

We also gather the runtime information that on average how much our pathfinding algorithm was able to find the correct path and present this information in the

Fig. 43: Map Size 400*400 using K=10 Full Data in Table 14

following section based on each map size.

Experiments show that partial min-heap and min-heap run time was almost similar but in some cases, partial min-heap found the path faster by a small margin. Although that cached min-heap found the path executing less number of operations as shown in the Fig 45 but it took more time to find the path. Using smaller value for k as the depth of our partial min-heap will result in having a better time compared to min-heap, but our pathfinding algorithm will not find the path in the majority of iterations.



	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	121213	630977	915128	2223968	2220915	3621292	4196901	4684147	5072422	5149080	5471237	5185893
Min-Heap	17071	55411	78328	154233	165846	256958	290619	373106	427321	476686	530964	616635
Cached Min-Heap	33087	72604	85242	492097	356871	746173	733092	486506	411989	376654	379643	432875
Partial Min-Heap	14003	45865	54107	67578	54374	97175	101917	141066	157270	215458	261399	485375

Fig. 44: Map Size 512*512 using K=9 Full Data in Table 15

4.8 Summary

Based on our analysis on this certain map size, using different chance and distribution of obstacles, we came to this conclusion that Partial min-heap mostly performs better in term of number of operations in comparison to other data structures in most of the cases, with a chance of not finding a path. Using the correct K in each map size will result in better performance than the min-heap data structure with a small chance of failure. Cached min-heap perform better in terms of number of operations when the chance of obstacles are higher than certain percentage in each map but runtime of this data structure is higher than min-heap and partial min-heap. In general, unsorted

	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	129144	589478	1345530	2132006	2400230	3831368	4371062	3235347	4289284	4627432	5196288	3183882
Min-Heap	17343	52155	90505	149581	192769	260246	285071	269076	376704	428573	536272	412963
Cached Min-Heap	38936	73940	217274	339979	326654	873226	825142	361938	321927	327727	381370	290680
Partial Min-Heap	16315	51724	64580	100199	158436	188852	182623	220482	347115	403729	525741	403988

Fig. 45: Map Size 512*512 using K = 10 Full Data in Table 15

array performed worst in terms of operation and runtime compare to other three data structures. Based on our observations and experiments, we do not recommend using unsorted array as the data structure for the finding the shortest path using A* algorithm. Min-heap performance is guaranteed to find the path in any situation and map size. Cached min-heap is designed to logically separate data into two sections that might will help the pathfinding the algorithm faster in some cases. The issue with cached min-heap is Replenish-cache which is time consuming. Replenish-cache operations consume a lot of operations and times since it uses an average case o(n log n) operations to re-fill the min-heap section of the data structure. Partial min-heap performs better in the number of executed operations in most of the iterations and

	0	5	10	15	20	25	30	35	40	45	50	55
Unsorted Array	149451	381413	917446	1689380	1911819	2637751	3573103	4671923	3901518	4272846	5582816	4582653
Min-Heap	18829	38394	76822	139643	151773	191283	263556	352846	353663	418784	565952	559534
Cached Min-Heap	57884	49033	78801	171058	237814	588809	560485	652194	287853	313069	402567	393125
Partial Min-Heap	18593	37985	75927	137786	149692	176282	251337	339504	347301	410904	554971	547538

Fig. 46: Map Size 512*512 using K = 11 Full Data in Table 16

obstacle chances. Partial min-heap runtime is better than min-heap in most of the cases, but it requires a correct estimation of the depth of the tree to correctly find the correct path. In cases that the memory is limited, we recommend using partial min-heap since it limits the search space for the A* algorithm and delivers an acceptable success rate.

Fig. 47: Run Time Map Size 400*400 using K=9 & 10

Fig. 48: Run Time Map Size 512*512 using K=10 & 11

CHAPTER 5

Conclusion

In this thesis, we tried to explore a way to improve the efficiency of A* pathfinding algorithm's data structure. We found out that A* algorithm *open set* contains a number of unnecessary nodes that only increase the number of operations required for this algorithm to find an optimal path which they are mostly not useful information to our algorithm. We proposed two new data structures called partial min-heap which is a optimistic approach to find a solution for this algorithm and cached min-heap which is based on the already existed approach of Heap on top Priority queues. We also compared the performance of these data structure with already existing solutions for this problem and analyzed them in different map sizes and density of obstacles.

In partial min-heap, we limited the size of the data structure for this algorithm and kept track of the location of our node with the max f cost in the heap so that in case of an overflow the newly inserted node with the smaller f cost to replace the current maximum node. Our approach to this problem resulted in better performance in time and number of executed operations, but the take away was success rate. We found out that partial min-heap performed poorly when the size of the suggested memory was too small, and it was only successful in a small number of problems which we found out that in those cases the size of the path was small too. So we suggested that in our performance evaluation chapter that which size of partial min-heap will result in better performance in terms of both number of operations and time based on each map size.

Our second proposed data structure, Cached min-heap was our approach developing a distributed data structure based on the heap on top priority queues. This

data structure logically separated the inserted data into two parts in one single array. First part follows the properties of a min-heap and the second part of it is an unsorted array. This solution is guaranteed to find the best path, but the performance of it is not stable. In some cases, cached min-heap performs better than min-heap and in some instances performs worst than min-heap in terms of operations, but since that insert operation will require multiple comparisons, it will not perform as quick as min-heap in runtime.

In conclusion, our proposed data structures could perform better in term of operations and runtime. Our partial min-heap, under the right circumstances, out-perform min-heap in runtime and number of operations with an only small number of failures which still give us the time to run the algorithm with min-heap to find the correct path in case of failures.

CHAPTER 6

Future Work

Our proposed Cached min-heap data structure helps us to limit the size of used memory to find the path to the goal node with less number of operations, but it suffers in runtime. Based on our experiments this data structure suffers the most when the replenish cache is getting called, and it needs to sort the unsorted array and fill the empty heap. Also, since that comparison operations for this data structure in the insert method is time-consuming, we still think that this data structure can still be better than partial min-heap and min-heap if we can further optimize these operations.

One of the solutions that we can propose for future work is detailed research in further distribution in the second part of the data structure. This act will limit the size of the nodes that it needs to be sorted and can save valuable number of operations. The second solution could be further improving the implementation of the insert method in Cached min-heap to increase the time efficiency.

APPENDICES

In this chapter we presented all of our results in test cases.

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	5252	5252	3216.834752	37.44708362	13223	394	0	Cached Min Heap
5	100	0	7980	7980	10494.22827	58.78702424	50526	116	0	Cached Min Heap
10	100	0	8599	8599	8728.924213	59.34646702	38086	34	0	Cached Min Heap
15	100	0	17046	17046	33953.96953	60.69591414	223115	176	0	Cached Min Heap
20	100	0	13671	13671	20515.34408	54.65256995	148564	83	0	Cached Min Heap
25	100	0	9811	9811	14047.89105	39.57316302	89743	95	0	Cached Min Heap
30	100	0	20667	20667	25355.00307	52.0602432	115948	80	1	Cached Min Heap
35	100	0	21642	21642	26151.02081	40.5316326	107321	98	2	Cached Min Heap
40	99	1	16369	16534	18424.15689	38.89475197	93766	93	1	Cached Min Heap
45	100	0	23047	23047	25411.81854	61.4206515	117963	320	2	Cached Min Heap
50	95	5	27058	22233	19240.32637	43.93865072	83488	655	4	Cached Min Heap
55	80	20	25543	19854	19995.5112	87.42063689	88121	131	4	Cached Min Heap
0	100	0	3586	3586	1677.000313	40.95119428	6647	450	0	Heap
5	100	0	4318	4318	3703.993418	60.86044214	21618	127	0	Heap
10	100	0	5504	5504	4208.449096	64.87256042	19877	36	0	Heap
15	100	0	8817	8817	10583.60635	102.876656	70053	199	0	Heap
20	100	0	9254	9254	9169.9957	95.76009451	52943	94	0	Heap
25	100	0	8353	8353	7821.33645	88.43832003	35929	103	0	Heap
30	100	0	15700	15700	13879.49713	117.8112776	51732	88	1	Heap
35	100	0	18915	18915	18325.89238	135.3731597	68102	111	1	Heap
40	99	1	16820	16990	15319.21413	123.7708129	65801	106	1	Heap
45	100	0	24877	24877	22379.02279	149.5961991	98141	375	2	Heap
50	95	5	33393	26826	20494.18479	143.1579016	90346	795	3	Heap
55	80	20	35127	26955	26971.60156	164.2303308	116310	149	4	Heap
0	100	0	8321	8321	5671.052388	75.30639009	23211	354	1	List
5	100	0	12704	12704	15915.86294	126.1580871	84689	91	1	List
10	100	0	16822	16822	17070.27345	130.6532566	87158	31	2	List
15	100	0	32395	32395	54851.90039	234.2048257	387112	154	4	List
20	100	0	33103	33103	43037.4304	207.4546466	262315	82	4	List
25	100	0	26471	26471	33344.10124	182.6036726	156033	82	3	List
30	100	0	58986	58986	66747.18494	258.3547657	303631	85	7	List
35	100	0	68712	68712	82011.02499	286.3756711	334703	103	8	List
40	99	1	56624	57196	64248.18274	253.4722524	302366	103	6	List
45	100	0	88219	88219	103316.45	321.4287634	507322	412	10	List
50	95	5	113037	88730	84294.24458	290.3347113	368977	1141	13	List
55	80	20	98612	75180	88899.6521	298.1604469	392508	159	11	List
0	34	66	2461	2536	1402.284072	37.44708362	5015	442	0	Partial Min Heap
5	43	57	2687	2818	3455.91422	58.78702424	16747	125	0	Partial Min Heap
10	35	65	3181	3466	3522.003148	59.34646702	12938	35	0	Partial Min Heap
15	27	73	3182	2683	3683.993993	60.69591414	18735	195	0	Partial Min Heap
20	29	71	3011	2388	2986.903402	54.65256995	15002	90	0	Partial Min Heap
25	26	74	3717	2450	1566.035231	39.57316302	5924	102	0	Partial Min Heap
30	27	73	4067	2686	2710.268922	52.0602432	8662	85	0	Partial Min Heap
35	26	74	4733	2643	1642.813241	40.5316326	6606	107	0	Partial Min Heap
40	26	74	5149	1955	1512.801731	38.89475197	5908	102	0	Partial Min Heap
45	25	75	6765	5187	3772.496431	61.4206515	14697	363	0	Partial Min Heap
50	26	74	8936	5296	1930.605027	43.93865072	8675	765	0	Partial Min Heap
55	39	61	11834	7561	7642.367755	87.42063689	25673	145	1	Partial Min Heap

Table 3: Map Size 120*120 using K=6 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	5400	5400	4014.142212	39.38956713	19901	280	0	Cached Min Heap
5	100	0	8337	8337	10060.53355	56.7165168	71174	430	0	Cached Min Heap
10	100	0	12586	12586	21781.11611	84.55455934	165461	160	0	Cached Min Heap
15	100	0	14016	14016	19451.94703	74.42694437	124156	322	0	Cached Min Heap
20	100	0	14445	14445	25129.04071	68.55881544	195586	278	0	Cached Min Heap
25	100	0	11329	11329	17215.50019	72.28116706	110302	38	0	Cached Min Heap
30	100	0	16185	16185	26562.86461	73.41395635	191277	34	1	Cached Min Heap
35	99	1	16901	15285	21528.47288	91.49851672	129597	89	1	Cached Min Heap
40	99	1	21298	19747	29123.27539	89.50319731	189616	116	2	Cached Min Heap
45	100	0	16637	16637	17433.8548	104.5536029	69705	324	2	Cached Min Heap
50	97	3	19918	17662	16597.35908	128.5160127	72217	59	3	Cached Min Heap
55	83	17	25959	22899	20775.14819	129.9964688	108133	45	4	Cached Min Heap
0	100	0	3500	3500	1868.569565	43.22695415	8144	320	0	Heap
5	100	0	4830	4830	3449.18119	58.72973003	18920	504	0	Heap
10	100	0	6743	6743	7208.668396	84.90387739	49636	178	0	Heap
15	100	0	7793	7793	6856.424155	82.80352743	35432	364	0	Heap
20	100	0	9476	9476	9714.559681	98.56246588	58650	316	0	Heap
25	100	0	9111	9111	9560.571186	97.77817336	43668	40	0	Heap
30	100	0	12900	12900	15722.6408	125.389955	109239	36	1	Heap
35	99	1	16272	13796	13329.11828	115.4518007	69212	98	1	Heap
40	99	1	21140	18716	20905.67976	144.5879655	112812	129	2	Heap
45	100	0	18948	18948	17522.99261	132.3744409	63801	385	2	Heap
50	97	3	26187	22916	20836.96656	144.3501526	96700	65	3	Heap
55	83	17	35165	30632	27722.83271	166.5017499	146637	51	4	Heap
0	100	0	8497	8497	7024.30378	83.81111967	30121	233	1	List
5	100	0	14183	14183	14469.45556	120.2890501	81637	434	2	List
10	100	0	22418	22418	33229.50266	182.2896121	246122	127	3	List
15	100	0	27172	27172	32623.41235	180.6195237	183168	304	3	List
20	100	0	33095	33095	47385.92035	217.6830732	345290	286	4	List
25	100	0	29866	29866	40144.19451	200.360162	207113	31	3	List
30	100	0	47427	47427	80266.13201	283.3127812	638914	30	5	List
35	99	1	57185	47733	62480.37898	249.9607549	346041	99	6	List
40	99	1	80395	68997	103186.4697	321.2265085	649893	89	9	List
45	100	0	61452	61452	70099.46347	264.7630327	275964	395	7	List
50	97	3	80854	69497	75941.20896	275.5743257	345908	59	9	List
55	83	17	105082	90999	102628.0736	320.3561668	580671	69	11	List
0	68	32	3705	2811	1551.537999	39.38956713	6198	313	0	Partial Min Heap
5	72	28	4686	4171	3216.763278	56.7165168	16418	492	0	Partial Min Heap
10	73	27	6403	6305	7149.473505	84.55455934	42383	174	0	Partial Min Heap
15	59	41	6613	5711	5539.370048	74.42694437	28365	358	0	Partial Min Heap
20	65	35	7133	5456	4700.311175	68.55881544	22273	311	0	Partial Min Heap
25	73	27	7223	5610	5224.567112	72.28116706	25461	39	0	Partial Min Heap
30	68	32	8387	5959	5389.608987	73.41395635	23657	35	0	Partial Min Heap
35	73	27	11607	8678	8371.978561	91.49851672	33646	94	1	Partial Min Heap
40	75	25	11943	9169	8010.822329	89.50319731	47939	125	1	Partial Min Heap
45	75	25	14696	11216	10931.45587	104.5536029	46442	375	1	Partial Min Heap
50	83	17	23283	17382	16516.36553	128.5160127	94263	63	2	Partial Min Heap
55	73	27	29978	22950	16899.08189	129.9964688	69348	47	4	Partial Min Heap

Table 4: Map Size 120*120 using K=7 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	5425	5425	4398.036197	38.94219529	19107	204	0	Cached Min Heap
5	100	0	7417	7417	7002.994166	50.38775676	29989	256	0	Cached Min Heap
10	100	0	14094	14094	31969.19346	110.9191938	291472	120	0	Cached Min Heap
15	100	0	17862	17862	27070.76922	101.4107962	164172	193	1	Cached Min Heap
20	100	0	18293	18293	29408.08836	98.53283475	222633	93	1	Cached Min Heap
25	100	0	19410	19410	34666.95004	100.5579776	295468	200	1	Cached Min Heap
30	100	0	14328	14328	18257.44973	113.9010316	89706	245	1	Cached Min Heap
35	100	0	20009	20009	23479.27461	114.6450611	110760	26	1	Cached Min Heap
40	100	0	21605	21605	22712.35249	125.8265357	102951	84	2	Cached Min Heap
45	99	1	22307	22532	22016.28515	134.8304297	116028	242	2	Cached Min Heap
50	98	2	21192	18442	19887.27622	151.7364614	125621	125	3	Cached Min Heap
55	81	19	27986	21074	20952.51666	166.8248212	98468	476	5	Cached Min Heap
0	100	0	3390	3390	1892.30576	43.50064092	8512	222	0	Heap
5	100	0	4164	4164	2727.630805	52.22672501	12726	292	0	Heap
10	100	0	7462	7462	13172.30006	114.7706411	125584	132	0	Heap
15	100	0	9408	9408	10521.70641	102.5753694	59235	218	0	Heap
20	100	0	11246	11246	10586.67483	102.8915683	59651	102	1	Heap
25	100	0	12870	12870	13698.91775	117.0423759	86845	227	1	Heap
30	100	0	12374	12374	13181.49436	114.8106892	69718	270	1	Heap
35	100	0	16813	16813	14640.46231	120.9977781	66382	30	1	Heap
40	100	0	21232	21232	17746.97921	133.2177886	78253	95	2	Heap
45	99	1	23949	24191	19214.14551	138.6150984	87797	280	2	Heap
50	98	2	27034	23407	23598.79796	153.6190026	130460	135	3	Heap
55	81	19	38502	28648	28591.96548	169.091589	134148	580	5	Heap
0	100	0	8088	8088	7299.564141	85.43748674	32331	148	1	List
5	100	0	11999	11999	11868.62429	108.9432159	47019	214	1	List
10	100	0	28325	28325	80276.214	283.3305737	785465	92	3	List
15	100	0	35385	35385	50393.94687	224.4859614	298211	171	4	List
20	100	0	41053	41053	55367.55723	235.3031178	404188	86	5	List
25	100	0	49300	49300	76714.95379	276.9746447	590823	212	6	List
30	100	0	42478	42478	55268.18694	235.0918691	294579	273	5	List
35	100	0	61131	61131	69234.90432	263.1252636	364288	28	7	List
40	100	0	74866	74866	78883.22231	280.8615714	362225	99	8	List
45	99	1	83276	84116	84017.49761	289.8577196	402876	319	9	List
50	98	2	86831	73003	91902.14662	303.1536683	563683	125	9	List
55	81	19	109640	83086	104053.5136	322.5732685	493281	725	12	List
0	92	8	3780	3003	1516.494574	38.94219529	7194	217	0	Partial Min Heap
5	92	8	4522	3799	2538.926031	50.38775676	12584	285	0	Partial Min Heap
10	99	1	7347	7234	12303.06756	110.9191938	115669	129	0	Partial Min Heap
15	96	4	9283	8863	10284.14958	101.4107962	58261	213	0	Partial Min Heap
20	96	4	11163	10296	9708.719524	98.53283475	54192	100	0	Partial Min Heap
25	96	4	11935	10995	10111.90687	100.5579776	52854	222	1	Partial Min Heap
30	99	1	12200	12116	12973.445	113.9010316	68232	266	1	Partial Min Heap
35	98	2	16133	15615	13143.49004	114.6450611	56172	28	1	Partial Min Heap
40	97	3	20768	19448	15832.31708	125.8265357	66701	91	2	Partial Min Heap
45	98	2	23182	23115	18179.24476	134.8304297	85816	273	2	Partial Min Heap
50	98	2	26318	22800	23023.95372	151.7364614	127406	133	3	Partial Min Heap
55	81	19	37326	27826	27830.52098	166.8248212	130552	564	4	Partial Min Heap

Table 5: Map Size 120*120 using K=8 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	9887	9887	11008.94745	41.31842231	51014	774	0	Cached Min Heap
5	100	0	22097	22097	37790.99425	91.87713388	285640	277	1	Cached Min Heap
10	100	0	33653	33653	64298.08834	76.0464564	394500	92	2	Cached Min Heap
15	100	0	38525	38525	67233.92183	74.89063931	385115	823	2	Cached Min Heap
20	100	0	34047	34047	57062.43382	56.57735876	345339	1050	2	Cached Min Heap
25	100	0	64152	64152	130081.0977	43.76159936	803904	685	4	Cached Min Heap
30	100	0	47248	47248	78690.45145	42.49230976	512218	135	4	Cached Min Heap
35	100	0	48354	48354	62360.94783	45.81713248	383033	349	5	Cached Min Heap
40	100	0	43235	43235	53360.75942	47.22930699	287356	294	5	Cached Min Heap
45	99	1	49279	49776	62202.85715	54.35768298	312906	30	6	Cached Min Heap
50	96	4	54349	42563	45390.13864	48.90818346	273640	202	8	Cached Min Heap
55	83	17	65515	58423	71998.71268	64.26153388	354229	210	12	Cached Min Heap
0	100	0	5579	5579	3184.549434	56.43181225	14705	885	0	Heap
5	100	0	9728	9728	8677.859688	93.1550304	57315	318	0	Heap
10	100	0	14975	14975	14393.69023	119.9737064	61492	100	1	Heap
15	100	0	20711	20711	20179.29136	142.0538326	98916	971	2	Heap
20	100	0	21661	21661	22351.96778	149.505745	90190	1263	2	Heap
25	100	0	33465	33465	38360.0597	195.8572432	150618	798	3	Heap
30	100	0	32181	32181	34483.58778	185.6975707	137602	151	3	Heap
35	100	0	41042	41042	37050.06585	192.4839366	155766	406	4	Heap
40	100	0	43870	43870	41315.3177	203.2616976	170545	338	5	Heap
45	99	1	52995	53531	50878.37142	225.5623449	215562	33	6	Heap
50	96	4	70669	54162	50164.1579	223.9735652	241573	229	9	Heap
55	83	17	92103	81644	102216.1406	319.7125907	501324	240	11	Heap
0	100	0	18117	18117	16919.29387	130.074186	86069	954	3	List
5	100	0	44915	44915	60431.84969	245.8289033	408802	238	7	List
10	100	0	76725	76725	97678.86804	312.5361868	470361	85	11	List
15	100	0	107630	107630	135442.9893	368.0257998	701466	1268	15	List
20	100	0	112044	112044	145152.247	380.988513	560843	1830	15	List
25	100	0	201270	201270	307051.3777	554.1221685	1601601	871	25	List
30	100	0	177557	177557	237249.4403	487.0825805	1212204	147	22	List
35	100	0	213989	213989	244428.6415	494.3972507	1240607	440	26	List
40	100	0	212545	212545	253490.2913	503.4781934	1164280	350	25	List
45	99	1	259247	261865	323715.7643	568.9602484	1473964	29	30	List
50	96	4	319652	236216	287403.0936	536.0998915	1498788	227	36	List
55	83	17	355602	316125	487395.6369	698.1372622	2379069	266	40	List
0	36	64	2824	3527	1707.212022	41.31842231	6631	868	0	Partial Min Heap
5	23	77	3660	7195	8441.407731	91.87713388	41006	311	0	Partial Min Heap
10	15	85	3023	4738	5783.063531	76.0464564	18947	99	0	Partial Min Heap
15	13	87	3850	5259	5608.607856	74.89063931	20688	955	0	Partial Min Heap
20	13	87	3648	4243	3200.997525	56.57735876	12894	1237	0	Partial Min Heap
25	15	85	3551	3234	1915.077579	43.76159936	7027	786	0	Partial Min Heap
30	22	78	3472	1878	1805.596389	42.49230976	6649	148	0	Partial Min Heap
35	15	85	4583	3077	2099.209629	45.81713248	7558	399	0	Partial Min Heap
40	11	89	5836	3099	2230.607439	47.22930699	8003	332	0	Partial Min Heap
45	16	84	6820	4289	2954.757699	54.35768298	11955	31	0	Partial Min Heap
50	15	85	8613	3846	2392.01041	48.90818346	10140	224	0	Partial Min Heap
55	19	81	13016	6219	4129.544737	64.26153388	13928	234	1	Partial Min Heap

Table 6: Map Size 200*200 using K=6 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	12631	12631	11817.62101	49.53283464	49428	170	0	Cached Min Heap
5	100	0	20711	20711	28716.31292	90.45868229	150670	140	1	Cached Min Heap
10	100	0	26687	26687	49819.15647	108.6604412	382349	327	1	Cached Min Heap
15	100	0	45250	45250	95562.51149	95.30008361	659178	230	3	Cached Min Heap
20	100	0	48403	48403	120355.8356	66.11251834	954659	194	3	Cached Min Heap
25	100	0	44283	44283	78346.05123	82.80529087	396904	216	3	Cached Min Heap
30	100	0	40602	40602	77007.67791	76.56781804	494209	68	3	Cached Min Heap
35	100	0	52251	52251	71801.28698	83.51478298	428922	148	5	Cached Min Heap
40	100	0	57284	57284	71552.47932	97.86830855	357976	220	6	Cached Min Heap
45	100	0	50753	50753	54365.02079	97.55218962	273770	241	7	Cached Min Heap
50	95	5	62375	51995	49865.3197	125.4315987	240883	393	10	Cached Min Heap
55	90	10	53318	47910	47172.11778	149.190263	261259	771	9	Cached Min Heap
0	100	0	6409	6409	3238.538515	56.90815859	12888	192	0	Heap
5	100	0	10095	10095	9012.156901	94.93238068	48687	153	0	Heap
10	100	0	14049	14049	14261.54274	119.4217013	81470	358	1	Heap
15	100	0	23780	23780	25379.43652	159.3092481	138885	253	2	Heap
20	100	0	26313	26313	28674.96685	169.3368443	167813	225	2	Heap
25	100	0	27220	27220	25214.87766	158.791932	139763	247	3	Heap
30	100	0	29019	29019	32705.66981	180.8470896	155261	77	3	Heap
35	100	0	42359	42359	40990.41131	202.4608883	173359	166	4	Heap
40	100	0	55158	55158	51157.87613	226.1810694	256671	255	6	Heap
45	100	0	57435	57435	50155.73262	223.9547557	183730	277	6	Heap
50	95	5	83360	68301	60163.12704	245.2817299	252265	473	10	Heap
55	90	10	73442	65271	62113.80991	249.226423	319265	942	9	Heap
0	100	0	22716	22716	17366.83195	131.7832765	67062	133	4	List
5	100	0	46900	46900	57141.58039	239.0430513	280937	119	7	List
10	100	0	66814	66814	88047.78789	296.728475	503045	307	10	List
15	100	0	132904	132904	193020.0152	439.3404321	1167835	213	18	List
20	100	0	143307	143307	220168.9576	469.2216508	1522408	212	19	List
25	100	0	142049	142049	184341.3208	429.3498816	1147189	234	18	List
30	100	0	148277	148277	221776.2879	470.9312985	1014027	56	18	List
35	100	0	221493	221493	267933.8248	517.623246	1302878	175	27	List
40	100	0	283593	283593	333873.4596	577.8178429	1694061	265	34	List
45	100	0	281176	281176	303069.8593	550.5178102	1253473	387	32	List
50	95	5	377184	303943	338608.3818	581.9006632	1507122	524	43	List
55	90	10	292133	259278	325470.6825	570.5003791	1902103	1388	32	List
0	51	49	5023	4462	2453.501707	49.53283464	9653	187	0	Partial Min Heap
5	43	57	7308	8675	8182.773201	90.45868229	32180	151	0	Partial Min Heap
10	37	63	8568	10100	11807.09148	108.6604412	62980	354	0	Partial Min Heap
15	38	62	9835	8746	9082.105936	95.30008361	33526	250	0	Partial Min Heap
20	34	66	10549	6136	4370.865081	66.11251834	17951	218	0	Partial Min Heap
25	32	68	10239	7460	6856.716197	82.80529087	26475	242	0	Partial Min Heap
30	40	60	11033	6846	5862.63076	76.56781804	21841	74	0	Partial Min Heap
35	36	64	12624	9104	6974.718977	83.51478298	27244	163	1	Partial Min Heap
40	32	68	16668	10924	9578.205819	97.86830855	39078	249	1	Partial Min Heap
45	42	58	18698	12144	9516.4297	97.55218962	39694	267	2	Partial Min Heap
50	42	58	32181	20171	15733.08596	125.4315987	67436	456	3	Partial Min Heap
55	52	48	37964	25899	22257.73458	149.190263	105954	910	4	Partial Min Heap

Table 7: Map Size 200*200 using K=7 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	11590	11590	12351.22461	52.0402304	57277	342	0	Cached Min Heap
5	100	0	20374	20374	27388.72944	92.38403779	161793	802	1	Cached Min Heap
10	100	0	22743	22743	33631.24892	108.1182462	247184	904	1	Cached Min Heap
15	100	0	61188	61188	117122.307	142.5036528	907643	181	3	Cached Min Heap
20	100	0	32207	32207	61619.78666	123.5417462	376097	4	2	Cached Min Heap
25	100	0	46263	46263	100759.4695	136.4286628	775947	240	3	Cached Min Heap
30	100	0	52410	52410	71510.73033	141.3549399	308416	343	4	Cached Min Heap
35	100	0	47572	47572	56380.72622	166.4761571	266625	545	5	Cached Min Heap
40	100	0	55338	55338	68279.98818	155.137069	418055	46	6	Cached Min Heap
45	99	1	57686	53649	61942.14792	190.7744502	306196	273	7	Cached Min Heap
50	96	4	54040	43229	42595.49625	207.8113202	253057	210	9	Cached Min Heap
55	90	10	64454	52816	45741.53274	248.7733947	257920	1438	11	Cached Min Heap
0	100	0	5937	5937	3242.054191	56.93903926	15037	392	0	Heap
5	100	0	9892	9892	8257.080898	90.86848132	55274	917	0	Heap
10	100	0	13577	13577	13373.79866	115.6451411	69926	1104	1	Heap
15	100	0	25733	25733	29691.81072	172.3131182	205100	211	2	Heap
20	100	0	21130	21130	23670.1046	153.8509168	135998	5	2	Heap
25	100	0	28753	28753	31279.14581	176.8591129	180485	272	3	Heap
30	100	0	35928	35928	29841.19753	172.7460493	126760	402	4	Heap
35	100	0	43642	43642	36204.32336	190.2743371	149804	648	5	Heap
40	100	0	51665	51665	43773.63337	209.2214936	192728	51	6	Heap
45	99	1	65932	59819	53452.90375	231.1988403	203770	326	8	Heap
50	96	4	73822	58095	54560.98842	233.5829369	270499	247	9	Heap
55	90	10	90125	73177	64093.76872	253.1674717	365553	1758	11	Heap
0	100	0	20675	20675	18461.43778	135.8728736	84609	315	3	List
5	100	0	44923	44923	51578.91649	227.1099216	308188	1067	7	List
10	100	0	66866	66866	87389.93013	295.6178786	449824	1500	9	List
15	100	0	151754	151754	233406.5823	483.1217055	1844975	154	21	List
20	100	0	107456	107456	172674.0823	415.5407107	1081030	5	14	List
25	100	0	153993	153993	241109.1746	491.0286902	1662105	250	20	List
30	100	0	190883	190883	208686.3584	456.8220204	843536	429	24	List
35	100	0	214532	214532	222279.2622	471.465017	1002744	791	26	List
40	100	0	263446	263446	290558.7719	539.0350377	1549643	53	31	List
45	99	1	325810	294575	333553.9653	577.5413104	1507999	403	37	List
50	96	4	324497	249284	309449.6804	556.2820151	1657643	335	36	List
55	90	10	360014	291266	312250.2109	558.7935316	1834580	2638	40	List
0	81	19	6459	5006	2708.18558	52.0402304	13160	383	0	Partial Min Heap
5	76	24	10260	9274	8534.810439	92.38403779	50824	910	0	Partial Min Heap
10	80	20	13092	11765	11689.55517	108.1182462	57542	1082	1	Partial Min Heap
15	67	33	20602	17836	20307.29106	142.5036528	86688	205	2	Partial Min Heap
20	75	25	17981	16196	15262.56306	123.5417462	70230	4	1	Partial Min Heap
25	75	25	22804	18982	18612.78003	136.4286628	73177	267	2	Partial Min Heap
30	69	31	27059	23114	19981.21904	141.3549399	83596	394	2	Partial Min Heap
35	78	22	34829	31787	27714.31088	166.4761571	115369	632	3	Partial Min Heap
40	71	29	39549	29637	24067.51018	155.137069	100700	49	4	Partial Min Heap
45	81	19	54249	41534	36394.89084	190.7744502	167290	316	6	Partial Min Heap
50	92	8	66183	50374	43185.54482	207.8113202	246809	237	8	Partial Min Heap
55	89	11	87019	70131	61888.2019	248.7733947	356634	1718	11	Partial Min Heap

Table 8: Map Size 200*200 using K=8 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	5748	5748	5026.765811	57.8829359	35373	136	0	Cached Min Heap
5	100	0	8523	8523	9289.540758	94.71872054	55212	320	0	Cached Min Heap
10	100	0	12890	12890	12817.93791	122.8770919	64395	488	1	Cached Min Heap
15	100	0	19622	19622	27869.42984	147.5174516	239391	410	2	Cached Min Heap
20	100	0	25419	25419	21528.66775	165.5438832	88523	489	3	Cached Min Heap
25	100	0	39479	39479	70298.84734	190.5957954	657790	376	4	Cached Min Heap
30	100	0	38107	38107	65049.81851	194.4316699	542528	108	4	Cached Min Heap
35	100	0	38674	38674	41669.55093	188.7252195	250950	322	5	Cached Min Heap
40	99	1	50937	46443	44321.07429	216.9423278	257120	280	8	Cached Min Heap
45	100	0	40530	40530	36755.86476	222.6709175	169408	46	6	Cached Min Heap
50	92	8	70129	43438	42044.81979	239.5891925	200233	466	14	Cached Min Heap
55	90	10	68241	56598	46137.82743	251.7629127	204923	193	13	Cached Min Heap
0	100	0	6076	6076	3243.843661	56.954751	13471	153	0	Heap
5	100	0	9412	9412	9046.430423	95.11272482	70400	352	0	Heap
10	100	0	15367	15367	15236.60632	123.436649	82378	547	1	Heap
15	100	0	23044	23044	23942.49371	154.7336218	120111	471	2	Heap
20	100	0	32651	32651	27905.40852	167.0491201	116163	559	3	Heap
25	100	0	43109	43109	41057.66306	202.6269061	200986	439	5	Heap
30	100	0	41511	41511	45653.97841	213.6679162	222659	118	4	Heap
35	100	0	46778	46778	40343.049	200.8557915	182781	375	5	Heap
40	99	1	66161	59370	50586.92733	224.9153781	188184	334	8	Heap
45	100	0	55165	55165	50605.70717	224.957123	234555	50	7	Heap
50	92	8	98531	59746	58666.45577	242.211593	279230	549	13	Heap
55	90	10	95774	78707	64903.88788	254.7624146	288782	230	13	Heap
0	100	0	20722	20722	16862.3298	129.8550338	72300	94	3	List
5	100	0	41170	41170	57738.265	240.2878794	478633	280	6	List
10	100	0	79734	79734	107135.7271	327.3159438	582211	533	11	List
15	100	0	130150	130150	176332.8609	419.9200649	1070355	389	18	List
20	100	0	187226	187226	198136.9922	445.1258162	836959	566	24	List
25	100	0	256605	256605	343773.0576	586.3216332	2277250	448	33	List
30	100	0	253977	253977	388691.8273	623.4515437	2442070	115	33	List
35	100	0	257925	257925	296172.2458	544.2170943	1522742	482	33	List
40	99	1	364121	326427	369713.6384	608.0408197	1624697	392	45	List
45	100	0	271695	271695	315777.6008	561.9409229	1553529	50	32	List
50	92	8	471084	266057	356278.0398	596.890308	1801056	807	61	List
55	90	10	388638	316936	326683.8494	571.5626382	1604685	283	48	List
0	100	0	6053	6053	3350.434269	57.8829359	16254	149	0	Partial Min Heap
5	99	1	9454	9276	8971.636021	94.71872054	69488	349	0	Partial Min Heap
10	99	1	15350	15209	15098.77972	122.8770919	81340	542	1	Partial Min Heap
15	99	1	22591	21778	21761.39852	147.5174516	118512	461	2	Partial Min Heap
20	99	1	32222	31807	27404.77726	165.5438832	114133	550	3	Partial Min Heap
25	97	3	40929	39356	36326.75722	190.5957954	161024	429	4	Partial Min Heap
30	96	4	37825	35909	37803.67426	194.4316699	185539	116	4	Partial Min Heap
35	96	4	44795	42519	35617.20849	188.7252195	156674	366	5	Partial Min Heap
40	96	4	63943	55114	47063.97357	216.9423278	185076	325	7	Partial Min Heap
45	100	0	53977	53977	49582.33748	222.6709175	229676	48	6	Partial Min Heap
50	92	8	96063	58351	57402.98117	239.5891925	273227	530	14	Partial Min Heap
55	90	10	93265	76722	63384.56423	251.7629127	282075	220	13	Partial Min Heap

Table 9: Map Size 200*200 using K=9 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	17927	17927	21180.5015	64.73169839	82973	1142	0	Cached Min Heap
5	100	0	46077	46077	65807.24207	149.894721	422817	684	3	Cached Min Heap
10	100	0	38746	38746	89702.4825	139.0441867	844208	309	3	Cached Min Heap
15	100	0	40493	40493	58489.18912	148.6798249	488985	208	4	Cached Min Heap
20	100	0	63002	63002	91382.74326	170.6385846	537700	1000	6	Cached Min Heap
25	100	0	146637	146637	277498.8552	148.5965883	2102559	296	12	Cached Min Heap
30	100	0	130075	130075	270682.9522	159.8302901	2051331	44	12	Cached Min Heap
35	100	0	115109	115109	198098.0386	169.3514589	1099160	1206	12	Cached Min Heap
40	100	0	150392	150392	212878.1312	162.6366386	1127654	681	17	Cached Min Heap
45	100	0	98108	98108	104924.2131	183.3348153	571915	760	15	Cached Min Heap
50	97	3	120612	113269	100040.7455	228.6652357	425680	596	20	Cached Min Heap
55	87	13	152405	108842	83811.95813	272.1519902	446466	1252	28	Cached Min Heap
0	100	0	9849	9849	4401.296712	66.34226942	18757	1353	1	Heap
5	100	0	24430	24430	22938.97713	151.4561888	106698	784	3	Heap
10	100	0	28919	28919	28304.04378	168.2380569	164301	346	3	Heap
15	100	0	39382	39382	37436.68108	193.4856095	185979	238	4	Heap
20	100	0	59783	59783	58548.38487	241.9677352	252283	1167	7	Heap
25	100	0	89292	89292	75237.25033	274.2940946	293828	331	10	Heap
30	100	0	92114	92114	84894.48118	291.3665753	341203	48	10	Heap
35	100	0	97009	97009	91927.7983	303.1959734	465667	1488	11	Heap
40	100	0	129516	129516	116139.9487	340.7931171	497146	795	15	Heap
45	100	0	125484	125484	114301.6043	338.0852028	464401	953	15	Heap
50	97	3	165705	154930	133382.0398	365.2150597	536236	730	21	Heap
55	87	13	216471	152589	118705.4571	344.5365831	637116	1603	28	Heap
0	100	0	45121	45121	31348.7251	177.0557119	129097	1664	9	List
5	100	0	183791	183791	213814.5722	462.4008783	983769	774	29	List
10	100	0	208286	208286	266280.3283	516.0235734	1408960	291	31	List
15	100	0	265384	265384	311254.4094	557.9017919	1632835	181	38	List
20	100	0	428984	428984	544570.582	737.9502571	3392918	1667	58	List
25	100	0	738014	738014	800034.5848	894.4465243	4129442	268	98	List
30	100	0	730430	730430	897148.9635	947.179478	4842053	32	94	List
35	100	0	718476	718476	929330.2443	964.0177614	5260234	2375	89	List
40	100	0	1018335	1018335	1209836.023	1099.925462	5543035	1102	123	List
45	100	0	833101	833101	995238.4047	997.6163615	4521046	1437	97	List
50	97	3	1050151	973065	1086863.588	1042.5275	4705561	1052	121	List
55	87	13	1087321	754945	738914.3733	859.6012874	4075554	2599	123	List
0	61	39	9940	8188	4190.192777	64.73169839	16236	1327	0	Partial Min Heap
5	52	48	18070	22509	22468.42738	149.894721	92747	769	1	Partial Min Heap
10	56	44	19948	19043	19333.28585	139.0441867	94664	341	1	Partial Min Heap
15	51	49	29146	25227	22105.69032	148.6798249	91679	232	3	Partial Min Heap
20	43	57	33818	26531	29117.52656	170.6385846	152615	1153	3	Partial Min Heap
25	35	65	29939	24997	22080.94606	148.5965883	85412	326	3	Partial Min Heap
30	44	56	33031	24554	25545.72165	159.8302901	90371	46	3	Partial Min Heap
35	51	49	41015	33941	28679.91664	169.3514589	110670	1453	4	Partial Min Heap
40	44	56	49514	33177	26450.67622	162.6366386	96031	783	6	Partial Min Heap
45	55	45	60419	45251	33611.6545	183.3348153	144886	918	7	Partial Min Heap
50	59	41	88242	65213	52287.79004	228.6652357	187204	707	11	Partial Min Heap
55	68	32	185267	103852	74066.70574	272.1519902	284653	1548	25	Partial Min Heap

Table 10: Map Size 300*300 using K=8 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	22894	22894	27904.63174	73.50771816	113606	808	1	Cached Min Heap
5	100	0	35068	35068	69781.99146	132.5461615	611257	692	2	Cached Min Heap
10	100	0	46964	46964	90579.06236	178.2474506	727165	551	4	Cached Min Heap
15	100	0	59214	59214	145981.7177	197.6093931	1031634	234	4	Cached Min Heap
20	100	0	91999	91999	196668.2783	224.4976965	1199167	1004	8	Cached Min Heap
25	100	0	130170	130170	241699.8876	220.6729168	1223282	424	10	Cached Min Heap
30	100	0	158528	158528	324539.7973	264.0453487	1869935	142	13	Cached Min Heap
35	98	2	179054	168624	229222.2572	289.5989189	1097443	687	18	Cached Min Heap
40	99	1	122445	123682	132829.3179	306.9172467	641591	144	16	Cached Min Heap
45	97	3	136969	141204	179491.5081	340.028056	1089201	51	21	Cached Min Heap
50	93	7	163134	131554	111603.9374	357.3657433	474444	692	28	Cached Min Heap
55	89	11	130650	118048	113209.6016	396.4755429	502441	599	24	Cached Min Heap
0	100	0	10603	10603	5198.998665	72.10408217	20271	951	1	Heap
5	100	0	19809	19809	18134.12331	134.662999	84923	768	2	Heap
10	100	0	32122	32122	32859.03602	181.2706154	161261	637	3	Heap
15	100	0	37637	37637	52035.08186	228.1119941	249737	267	4	Heap
20	100	0	60733	60733	67027.57589	258.8968441	295312	1153	7	Heap
25	100	0	75751	75751	72399.82979	269.0721647	296539	478	9	Heap
30	100	0	96652	96652	92439.05214	304.0379123	441915	163	11	Heap
35	98	2	135088	116827	100675.8966	317.2946527	372692	805	16	Heap
40	99	1	133595	134944	104309.7674	322.9702268	433673	161	16	Heap
45	97	3	159917	164861	160163.5417	400.2043749	696320	57	19	Heap
50	93	7	224925	177867	143426.7934	378.7173001	541816	868	29	Heap
55	89	11	184779	165990	160674.2324	400.8419045	717607	728	23	Heap
0	100	0	52052	52052	39508.59206	198.7676836	153168	1008	11	List
5	100	0	132647	132647	168921.2026	411.0002464	949706	807	22	List
10	100	0	233209	233209	308754.3239	555.6566601	1882557	641	36	List
15	100	0	292234	292234	536722.7627	732.6136518	2649724	208	42	List
20	100	0	489517	489517	727390.8294	852.8721061	4373318	1467	67	List
25	100	0	600134	600134	788817.6683	888.1540792	3636656	448	82	List
30	100	0	777839	777839	1041513.537	1020.545705	5305221	165	101	List
35	98	2	1112409	942252	1027030.266	1013.425017	4096235	975	139	List
40	99	1	957882	967557	934183.9269	966.5319068	3940866	147	117	List
45	97	3	1196240	1233233	1567761.566	1252.102858	7847976	51	144	List
50	93	7	1495173	1167685	1221469.782	1105.201241	4670416	1300	177	List
55	89	11	1011373	898525	1134710.057	1065.227702	5683965	950	117	List
0	88	12	12184	9992	5403.384628	73.50771816	23037	932	0	Partial Min Heap
5	88	12	20722	18608	17568.48492	132.5461615	79966	756	2	Partial Min Heap
10	83	17	31237	30092	31772.15363	178.2474506	156200	627	3	Partial Min Heap
15	92	8	33963	30132	39049.47226	197.6093931	176711	262	3	Partial Min Heap
20	87	13	52626	45359	50399.21576	224.4976965	249597	1133	5	Partial Min Heap
25	80	20	59250	50700	48696.53621	220.6729168	203152	469	6	Partial Min Heap
30	83	17	76620	73087	69719.94617	264.0453487	349415	158	8	Partial Min Heap
35	74	26	108637	83630	83867.53385	289.5989189	342048	790	13	Partial Min Heap
40	89	11	119770	117735	94198.19632	306.9172467	355738	158	14	Partial Min Heap
45	88	12	138100	127335	115619.0789	340.028056	431684	55	17	Partial Min Heap
50	89	11	214424	160512	127710.2745	357.3657433	503727	830	28	Partial Min Heap
55	89	11	180339	162081	157192.8561	396.4755429	701723	703	24	Partial Min Heap

Table 11: Map Size 300*300 using K=9 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	18379	18379	35898.34318	71.94692173	284758	894	1	Cached Min Heap
5	100	0	24578	24578	23975.83305	128.0258517	128513	1014	2	Cached Min Heap
10	100	0	47193	47193	73845.20575	216.1153192	574904	524	6	Cached Min Heap
15	100	0	127500	127500	615314.0872	206.5498215	6168978	464	12	Cached Min Heap
20	100	0	133500	133500	351226.4041	149.3089772	3101255	1591	14	Cached Min Heap
25	100	0	259071	259071	673732.5151	136.0477908	4432340	1033	22	Cached Min Heap
30	100	0	223860	223860	377853.4181	156.3846225	2241751	659	24	Cached Min Heap
35	100	0	151816	151816	294982.149	159.3840315	2346101	607	20	Cached Min Heap
40	100	0	206714	206714	227092.3517	181.3845753	1954732	1709	31	Cached Min Heap
45	99	1	180327	161457	153425.2833	201.962503	647993	863	31	Cached Min Heap
50	97	3	214544	183315	162151.5663	221.996758	575272	2396	38	Cached Min Heap
55	93	7	277813	230007	218943.778	311.7752984	1020285	853	51	Cached Min Heap
0	100	0	14922	14922	7054.094184	83.98865509	32169	1037	2	Heap
5	100	0	28592	28592	26463.51496	162.6761045	148706	1168	3	Heap
10	100	0	55326	55326	70279.22967	265.1023004	374907	608	6	Heap
15	100	0	95676	95676	163565.1641	404.431903	1132755	532	11	Heap
20	100	0	116228	116228	130144.6382	360.7556489	661965	1836	13	Heap
25	100	0	159626	159626	163012.4822	403.748043	589460	1213	19	Heap
30	100	0	189469	189469	169519.5522	411.7275218	711465	763	22	Heap
35	100	0	163002	163002	168686.2496	410.7143163	654532	730	19	Heap
40	100	0	260378	260378	194901.7222	441.4767516	947720	2163	31	Heap
45	99	1	251163	223203	214049.9637	462.6553401	898734	1073	32	Heap
50	97	3	302474	256423	228813.5231	478.3445653	807235	3071	39	Heap
55	93	7	395582	325809	313260.0734	559.6964118	1462711	1096	52	Heap
0	100	0	92796	92796	73130.37089	270.4262763	325541	928	20	List
5	100	0	243642	243642	315347.0949	561.5577396	1727637	1422	40	List
10	100	0	557622	557622	983872.1054	991.9032742	6452529	544	83	List
15	100	0	1120280	1120280	2758804.034	1660.96479	21816269	590	161	List
20	100	0	1237960	1237960	1946601.023	1395.206445	11325133	3017	173	List
25	100	0	1826549	1826549	2479208.739	1574.550329	10133718	1715	250	List
30	100	0	2062786	2062786	2336360.649	1528.515832	9695050	853	268	List
35	100	0	1571745	1571745	2225980.608	1491.972053	11645969	863	199	List
40	100	0	2477732	2477732	2519306.322	1587.232284	15367051	3531	305	List
45	99	1	2078859	1861325	2370728.783	1539.717111	11877280	1483	246	List
50	97	3	2340381	2006124	2258252.548	1502.748332	8433668	5094	273	List
55	93	7	2735435	2251023	2862453.361	1691.878648	12813568	1690	313	List
0	54	46	11864	11119	5176.359546	71.94692173	22361	1016	1	Partial Min Heap
5	49	51	18168	20217	16390.61872	128.0258517	76514	1151	1	Partial Min Heap
10	40	60	30257	36831	46705.83117	216.1153192	255687	596	3	Partial Min Heap
15	35	65	26506	31900	42662.82874	206.5498215	156169	527	2	Partial Min Heap
20	27	73	33512	26577	22293.17068	149.3089772	95073	1817	3	Partial Min Heap
25	25	75	29804	16376	18509.00138	136.0477908	77278	1194	3	Partial Min Heap
30	21	79	37843	30036	24456.15014	156.3846225	71286	750	4	Partial Min Heap
35	35	65	41747	24687	25403.26949	159.3840315	106531	712	5	Partial Min Heap
40	24	76	57353	37729	32900.36415	181.3845753	118702	2105	7	Partial Min Heap
45	44	56	74693	60969	40788.85263	201.962503	165963	1033	9	Partial Min Heap
50	42	58	94820	62888	49282.56057	221.996758	196176	2968	12	Partial Min Heap
55	53	47	159256	129472	97203.8367	311.7752984	394637	1052	21	Partial Min Heap

Table 12: Map Size 400*400 using K=8 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	18333	18333	38101.05182	76.72646536	293295	1706	1	Cached Min Heap
5	100	0	26272	26272	26134.44467	177.4237686	119553	222	3	Cached Min Heap
10	100	0	57243	57243	152661.9162	228.9530868	1514991	152	7	Cached Min Heap
15	100	0	65946	65946	97823.83993	241.4548621	787447	1104	9	Cached Min Heap
20	100	0	83769	83769	115516.9734	228.7738583	776333	376	11	Cached Min Heap
25	100	0	179999	179999	500413.0597	283.7337907	4103568	861	18	Cached Min Heap
30	100	0	169700	169700	469350.4578	308.5549269	4655493	1605	20	Cached Min Heap
35	100	0	214322	214322	403330.8382	321.0278097	2123422	686	23	Cached Min Heap
40	99	1	184734	186600	215794.876	361.392638	1543015	61	28	Cached Min Heap
45	100	0	215576	215576	198290.1556	362.5163312	1126110	2027	35	Cached Min Heap
50	98	2	249451	236478	191438.1475	432.1073146	977797	1377	44	Cached Min Heap
55	92	8	279299	252868	209298.272	518.6725622	792854	2894	51	Cached Min Heap
0	100	0	13700	13700	6813.985845	82.5468706	32651	2010	1	Heap
5	100	0	30662	30662	30637.15713	175.0347312	154254	254	3	Heap
10	100	0	65877	65877	143510.8653	378.8282794	1394471	172	8	Heap
15	100	0	78656	78656	90143.5941	300.2392281	423304	1302	9	Heap
20	100	0	98116	98116	113603.7007	337.0514808	485293	436	12	Heap
25	100	0	139878	139878	142817.9086	377.9125674	631330	1017	16	Heap
30	100	0	160744	160744	156557.9978	395.6741055	635814	2015	19	Heap
35	100	0	182971	182971	176839.3276	420.5226838	700761	817	22	Heap
40	99	1	235475	237854	226099.8696	475.4996	1079004	66	29	Heap
45	100	0	293379	293379	257867.5322	507.8065894	1091850	2541	37	Heap
50	98	2	350542	330851	269494.5691	519.1286633	1380486	1722	45	Heap
55	92	8	397740	358729	299567.6723	547.3277558	1133616	3736	52	Heap
0	100	0	80420	80420	65383.51535	255.702005	328694	2994	17	List
5	100	0	272021	272021	365219.9846	604.3343318	1711622	174	44	List
10	100	0	753273	753273	2507212.61	1583.418015	24853883	118	114	List
15	100	0	775584	775584	1147685.72	1071.300948	5663488	1848	111	List
20	100	0	990572	990572	1469840.413	1212.369751	7272784	386	140	List
25	100	0	1462427	1462427	2072156.13	1439.498569	12602826	1254	200	List
30	100	0	1548135	1548135	2032930.94	1425.808872	12403013	3385	204	List
35	100	0	1906813	1906813	2584468.322	1607.628167	11916477	1058	243	List
40	99	1	2306954	2330257	2968168.38	1722.837305	17271434	58	282	List
45	100	0	2705835	2705835	3251955.99	1803.31805	16137495	3864	325	List
50	98	2	2855831	2718813	2908183.811	1705.339793	15313563	2544	336	List
55	92	8	2666917	2431558	2607544.751	1614.789383	10427013	6759	305	List
0	84	16	14790	12186	5886.950487	76.72646536	23521	1973	1	Partial Min Heap
5	77	23	29881	29945	31479.19368	177.4237686	143537	248	3	Partial Min Heap
10	75	25	51751	50086	52419.51595	228.9530868	243379	167	5	Partial Min Heap
15	76	24	61277	53339	58300.45041	241.4548621	299013	1285	6	Partial Min Heap
20	73	27	74461	49839	52337.47824	228.7738583	258224	425	8	Partial Min Heap
25	68	32	88991	78612	80504.86399	283.7337907	334760	999	10	Partial Min Heap
30	70	30	114117	93813	95206.14289	308.5549269	363619	1967	13	Partial Min Heap
35	72	28	110543	99807	103058.8546	321.0278097	443841	803	13	Partial Min Heap
40	70	30	147961	126251	130604.6388	361.392638	569459	64	17	Partial Min Heap
45	73	27	200962	169783	131418.0904	362.5163312	659954	2469	24	Partial Min Heap
50	84	16	299748	251386	186716.7313	432.1073146	658789	1669	39	Partial Min Heap
55	89	11	384486	327548	269021.2268	518.6725622	1092736	3623	51	Partial Min Heap

Table 13: Map Size 400*400 using K=9 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	15570	15570	25968.0567	82.85323241	172503	880	1	Cached Min Heap
5	100	0	26624	26624	27872.9467	163.9559575	159982	407	2	Cached Min Heap
10	100	0	64749	64749	176494.4456	433.2962479	1516320	524	8	Cached Min Heap
15	100	0	70402	70402	130263.6188	278.1635219	935119	608	9	Cached Min Heap
20	100	0	294063	294063	1464274.745	314.2317948	12043183	290	21	Cached Min Heap
25	100	0	303154	303154	984787.6919	350.320962	8289638	1241	24	Cached Min Heap
30	100	0	135642	135642	158533.9221	377.2006995	994738	1343	19	Cached Min Heap
35	100	0	171151	171151	198629.4858	408.8058851	1155932	170	24	Cached Min Heap
40	100	0	196859	196859	209505.18	436.016358	1255525	4165	29	Cached Min Heap
45	98	2	196066	200066	172015.9607	485.0850577	682831	1203	33	Cached Min Heap
50	97	3	237453	206884	164543.3143	476.922584	637332	1988	42	Cached Min Heap
55	91	9	278753	237417	183883.3782	507.0268258	778065	1328	54	Cached Min Heap
0	100	0	13352	13352	7109.72536	84.31918738	29804	1037	1	Heap
5	100	0	29688	29688	27193.84906	164.9055762	129159	474	4	Heap
10	100	0	74267	74267	200421.6777	447.6847972	1857087	594	9	Heap
15	100	0	81461	81461	124293.8669	352.5533532	1013411	702	9	Heap
20	100	0	123352	123352	156263.1491	395.3013396	1013182	333	14	Heap
25	100	0	162907	162907	159755.6798	399.6944831	635300	1515	19	Heap
30	100	0	160253	160253	147696.6894	384.3132699	572134	1647	19	Heap
35	100	0	201998	201998	175992.1433	419.5141754	759554	197	24	Heap
40	100	0	244437	244437	211040.8455	459.3918214	861513	5513	29	Heap
45	98	2	271315	276851	239666.5933	489.5575486	944953	1519	33	Heap
50	97	3	334587	289391	231903.1511	481.5632368	901878	2521	43	Heap
55	91	9	397234	336323	262617.3891	512.4620856	1113192	1691	53	Heap
0	100	0	79519	79519	65089.33761	255.126121	295286	1171	17	List
5	100	0	258281	258281	316327.6344	562.4301152	1438510	485	42	List
10	100	0	938444	938444	3972333.516	1993.071378	38732966	605	140	List
15	100	0	879483	879483	2001635.192	1414.791572	17832835	790	127	List
20	100	0	1460488	1460488	3148209.361	1774.319408	22972308	251	208	List
25	100	0	1964554	1964554	2807194.454	1675.468428	16484882	2291	271	List
30	100	0	1553298	1553298	1772840.02	1331.480387	9085343	2534	204	List
35	100	0	1923439	1923439	2270469.119	1506.807592	9858718	159	242	List
40	100	0	2425506	2425506	2889114.101	1699.739422	13264194	11412	301	List
45	98	2	2400533	2449521	2753692.342	1659.425305	11774738	2293	293	List
50	97	3	2598973	2247669	2286377.104	1512.077083	9643404	4039	304	List
55	91	9	2644121	2217199	2287027.621	1512.292175	10921632	2612	306	List
0	98	2	13924	12926	6864.658121	82.85323241	29889	1017	1	Partial Min Heap
5	100	0	29372	29372	26881.556	163.9559575	127725	464	2	Partial Min Heap
10	99	1	71999	70563	187745.6385	433.2962479	1716062	587	7	Partial Min Heap
15	98	2	76344	68529	77374.94491	278.1635219	401615	691	8	Partial Min Heap
20	96	4	108463	99927	98741.62086	314.2317948	395631	325	12	Partial Min Heap
25	86	14	140676	124231	122724.7764	350.320962	519202	1485	16	Partial Min Heap
30	95	5	154066	152028	142280.3677	377.2006995	534533	1612	18	Partial Min Heap
35	96	4	192229	188293	167122.2517	408.8058851	747117	191	23	Partial Min Heap
40	96	4	231775	222137	190110.2645	436.016358	846863	5349	28	Partial Min Heap
45	98	2	266114	271544	235307.5132	485.0850577	928502	1473	33	Partial Min Heap
50	97	3	327516	283473	227455.1511	476.922584	883330	2432	42	Partial Min Heap
55	91	9	388013	328667	257076.2021	507.0268258	1090476	1625	52	Partial Min Heap

Table 14: Map Size 400*400 using K=10 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	33087	33087	58728.45223	86.81254517	293319	534	1	Cached Min Heap
5	100	0	72604	72604	102992.0083	214.0952593	665832	819	6	Cached Min Heap
10	100	0	85242	85242	109345.7555	252.8757542	689762	1522	9	Cached Min Heap
15	100	0	492097	492097	3612268.61	260.7393203	36353532	1201	29	Cached Min Heap
20	100	0	356871	356871	1178527.436	239.208389	10361507	1504	25	Cached Min Heap
25	100	0	746173	746173	1848326.622	312.1407001	10989029	7816	46	Cached Min Heap
30	99	1	771604	733092	1901169.814	321.2236732	14177394	1190	58	Cached Min Heap
35	100	0	486506	486506	766980.3576	357.2166547	4237598	2958	51	Cached Min Heap
40	100	0	411989	411989	618520.5793	366.7454866	4383201	630	54	Cached Min Heap
45	99	1	408382	376654	449944.1493	406.5076136	3247649	2261	65	Cached Min Heap
50	98	2	404706	379643	330419.4284	462.1730909	1586372	3817	69	Cached Min Heap
55	91	9	472789	432875	328600.4119	593.4462598	1597126	4053	88	Cached Min Heap
0	100	0	17071	17071	8526.993586	92.3417218	35029	590	2	Heap
5	100	0	55411	55411	53390.97256	231.0648666	229825	930	7	Heap
10	100	0	78328	78328	84019.50331	289.8611794	368507	1796	9	Heap
15	100	0	154233	154233	268984.1957	518.636863	2280165	1410	19	Heap
20	100	0	165846	165846	195937.5946	442.6483871	939573	1837	19	Heap
25	100	0	256958	256958	233864.0007	483.5948725	888943	9531	30	Heap
30	99	1	356919	290619	309043.4312	555.9167484	1534967	1361	45	Heap
35	100	0	373106	373106	345733.0613	587.9906983	1472913	3783	45	Heap
40	100	0	427321	427321	354569.9791	595.4577895	1552897	758	52	Heap
45	99	1	525248	476686	419237.0956	647.4852088	1715859	2890	66	Heap
50	98	2	567725	530964	457540.7231	676.4175656	2018812	4997	70	Heap
55	91	9	676586	616635	471621.3058	686.7469009	2292941	5409	90	Heap
0	100	0	121213	121213	97486.35225	312.2280453	385407	525	27	List
5	100	0	630977	630977	714179.8076	845.0915972	3221411	1079	100	List
10	100	0	915128	915128	1144098.625	1069.62546	6045919	2652	139	List
15	100	0	2223968	2223968	6804581.609	2608.559298	64434471	2075	330	List
20	100	0	2220915	2220915	3402855.675	1844.683083	21845010	2830	319	List
25	100	0	3621292	3621292	4669493.769	2160.901147	23520301	34224	499	List
30	99	1	5088222	4196901	6619283.316	2572.796789	42503718	2005	658	List
35	100	0	4684147	4684147	5607494.667	2368.014921	25340676	7147	596	List
40	100	0	5072422	5072422	5560225.609	2358.013064	23214519	1034	624	List
45	99	1	5812504	5149080	6145541.316	2479.020233	28436576	5038	694	List
50	98	2	5813538	5471237	6279223.678	2505.837919	31571793	8951	680	List
55	91	9	5660511	5185893	5020235.939	2240.588302	26138935	10739	654	List
0	68	32	18606	14003	7536.418	86.81254517	30312	579	1	Partial Min Heap
5	63	37	44161	45865	45836.78004	214.0952593	171850	915	4	Partial Min Heap
10	63	37	55436	54107	63946.14708	252.8757542	261983	1771	6	Partial Min Heap
15	58	42	95613	67578	67984.99314	260.7393203	266852	1397	10	Partial Min Heap
20	47	53	71515	54374	57220.65336	239.208389	278843	1801	8	Partial Min Heap
25	47	53	102429	97175	97431.81666	312.1407001	356813	9433	12	Partial Min Heap
30	49	51	124908	101917	103184.6482	321.2236732	450176	1345	15	Partial Min Heap
35	52	48	155725	141066	127603.7384	357.2166547	436633	3683	19	Partial Min Heap
40	48	52	182845	157270	134502.2519	366.7454866	590499	736	22	Partial Min Heap
45	59	41	285210	215458	165248.4399	406.5076136	604717	2809	36	Partial Min Heap
50	63	37	333162	261399	213603.9659	462.1730909	867651	4832	42	Partial Min Heap
55	77	23	606632	485375	352178.4633	593.4462598	1241686	5222	81	Partial Min Heap

Table 15: Map Size 512*512 using K=9 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	38936	38936	77819.42845	96.55528871	481723	386	2	Cached Min Heap
5	100	0	73940	73940	158087.7484	232.7268686	1191855	1238	6	Cached Min Heap
10	100	0	217274	217274	1516599.697	293.0842623	15274525	2660	14	Cached Min Heap
15	100	0	339979	339979	1112354.995	358.8988907	8004632	1354	23	Cached Min Heap
20	100	0	326654	326654	848691.455	412.6915392	5951347	300	27	Cached Min Heap
25	100	0	873226	873226	2109774.965	458.3424363	13655722	576	52	Cached Min Heap
30	100	0	825142	825142	1713183.315	421.1366985	10218591	636	52	Cached Min Heap
35	100	0	361938	361938	842343.0821	500.6095019	7304187	509	37	Cached Min Heap
40	99	1	318708	321927	456154.7807	548.2023594	3871134	1515	46	Cached Min Heap
45	100	0	327727	327727	358121.6251	620.2879692	2221254	467	52	Cached Min Heap
50	95	5	424675	381370	281516.8742	625.5437236	1330595	9982	75	Cached Min Heap
55	89	11	385244	290680	266130.2874	611.7291218	1124800	8420	71	Cached Min Heap
0	100	0	17343	17343	9708.733637	98.53290637	41601	448	2	Heap
5	100	0	52155	52155	54162.43109	232.7282344	281114	1458	6	Heap
10	100	0	90505	90505	224087.7523	473.3790789	2136372	3207	11	Heap
15	100	0	149581	149581	211327.8877	459.7041306	1332563	1579	17	Heap
20	100	0	192769	192769	201892.9515	449.3249954	850786	335	23	Heap
25	100	0	260246	260246	241523.323	491.4502244	1018062	679	31	Heap
30	100	0	285071	285071	285218.819	534.0588161	1198732	726	34	Heap
35	100	0	269076	269076	282202.4257	531.2272826	1205480	612	32	Heap
40	99	1	372937	376704	325169.5621	570.236409	1320028	1852	45	Heap
45	100	0	428573	428573	406213.3485	637.3486868	1462480	533	52	Heap
50	95	5	602061	536272	398642.3053	631.3812678	1881958	13310	77	Heap
55	89	11	552988	412963	381865.4468	617.952625	1616376	11319	73	Heap
0	100	0	129144	129144	114574.5811	338.488672	488389	373	28	List
5	100	0	589478	589478	768871.9957	876.8534631	4535576	2034	95	List
10	100	0	1345530	1345530	5381558.644	2319.818666	53211436	5483	200	List
15	100	0	2132006	2132006	3890802.044	1972.511608	27125621	2354	307	List
20	100	0	2400230	2400230	3118439.845	1765.910486	16416680	310	338	List
25	100	0	3831368	3831368	4815486.318	2194.421636	24549225	775	532	List
30	100	0	4371062	4371062	5984339.899	2446.291049	30692642	736	577	List
35	100	0	3235347	3235347	4521703.169	2126.429676	27409861	841	412	List
40	99	1	4246391	4289284	4903773.869	2214.446628	22573024	2748	528	List
45	100	0	4627432	4627432	5680925.039	2383.469119	21995116	607	553	List
50	95	5	5911350	5196288	5146010.612	2268.482006	27672397	32768	692	List
55	89	11	4154633	3183882	3990661.907	1997.664113	17215684	25171	480	List
0	94	6	19114	16315	9322.923777	96.55528871	47052	439	1	Partial Min Heap
5	96	4	52261	51724	54161.79539	232.7268686	273908	1440	5	Partial Min Heap
10	94	6	69787	64580	85898.38479	293.0842623	416635	3170	7	Partial Min Heap
15	86	14	124283	100199	128808.4137	358.8988907	819868	1560	14	Partial Min Heap
20	88	12	178127	158436	170314.3065	412.6915392	714073	326	20	Partial Min Heap
25	77	23	212264	188852	210077.789	458.3424363	1001108	668	24	Partial Min Heap
30	74	26	203331	182623	177356.1188	421.1366985	619325	716	24	Partial Min Heap
35	88	12	233448	220482	250609.8734	500.6095019	1182459	594	27	Partial Min Heap
40	95	5	351040	347115	300525.8268	548.2023594	1297629	1808	43	Partial Min Heap
45	98	2	411055	403729	384757.1647	620.2879692	1435228	523	52	Partial Min Heap
50	95	5	589822	525741	391304.9501	625.5437236	1848558	12941	76	Partial Min Heap
55	89	11	540335	403988	374212.5185	611.7291218	1583437	10979	72	Partial Min Heap

Table 16: Map Size 512*512 using K=10 Full Data

Obs Chance	Total Success	Total Failed	OverAll Avg Ops	Success Ops	Ops Variance	Ops SD	Max Ops	Min Ops	Overall Time	Success Nodes Expanded
0	100	0	57884	57884	90025.49605	98.28234522	296307	48	2	Cached Min Heap
5	100	0	49033	49033	82066.63835	188.5768864	538704	514	4	Cached Min Heap
10	100	0	78801	78801	110770.4491	306.6745851	738170	872	8	Cached Min Heap
15	100	0	171058	171058	279368.3657	373.8565735	1754931	1048	17	Cached Min Heap
20	100	0	237814	237814	567884.8498	401.6243296	3763349	1037	21	Cached Min Heap
25	100	0	588809	588809	1686185.931	414.1794445	12665397	382	35	Cached Min Heap
30	100	0	560485	560485	1298508.873	482.7399889	6551005	401	42	Cached Min Heap
35	100	0	652194	652194	1365797.849	534.0696566	9822667	637	53	Cached Min Heap
40	99	1	284975	287853	374784.8628	572.3703088	2983161	1553	43	Cached Min Heap
45	98	2	382386	313069	274157.6119	577.3439262	1469541	660	65	Cached Min Heap
50	97	3	449262	402567	297652.559	642.7481711	1167699	7793	81	Cached Min Heap
55	87	13	442651	393125	324324.6707	675.4912055	1588777	4139	82	Cached Min Heap
0	100	0	18829	18829	9769.780442	98.8421997	38974	52	2	Heap
5	100	0	38394	38394	35961.48871	189.6351463	141821	592	5	Heap
10	100	0	76822	76822	95258.26298	308.6393737	442246	1032	9	Heap
15	100	0	139643	139643	141729.8556	376.4702586	578118	1217	17	Heap
20	100	0	151773	151773	163595.8335	404.4698178	714601	1210	18	Heap
25	100	0	191283	191283	193213.5315	439.5606119	889962	442	22	Heap
30	100	0	263556	263556	241241.9505	491.1638734	1089152	454	31	Heap
35	100	0	352846	352846	297928.9922	545.8287206	1199808	779	43	Heap
40	99	1	350126	353663	333430.0723	577.4340415	1380703	1969	42	Heap
45	98	2	521290	418784	339365.5111	582.5508657	1594370	784	65	Heap
50	97	3	636407	565952	420855.1693	648.7335118	1646967	10317	80	Heap
55	87	13	634601	559534	465566.7339	682.3245077	2271113	5306	83	Heap
0	100	0	149451	149451	123442.1796	351.3433927	467890	32	32	List
5	100	0	381413	381413	475791.9915	689.7767693	2541446	510	62	List
10	100	0	917446	917446	1446421.66	1202.672715	9051016	1355	138	List
15	100	0	1689380	1689380	2102953.565	1450.156393	9157813	1729	244	List
20	100	0	1911819	1911819	2613552.055	1616.648402	14004014	1684	271	List
25	100	0	2637751	2637751	4034025.435	2008.488346	25218134	456	365	List
30	100	0	3573103	3573103	4553304.479	2133.847342	21287530	475	468	List
35	100	0	4671923	4671923	5581586.829	2362.538218	33475551	993	594	List
40	99	1	3862503	3901518	4588543.508	2142.088585	19856941	3238	475	List
45	98	2	5527481	4272846	4699919.894	2167.929864	21430198	958	663	List
50	97	3	6115063	5582816	5194332.609	2279.107854	19625015	23626	717	List
55	87	13	5025977	4582653	5048210.332	2246.822274	25372515	9837	574	List
0	100	0	18593	18593	9659.419382	98.28234522	38535	50	1	Partial Min Heap
5	100	0	37985	37985	35561.24208	188.5768864	140202	579	3	Partial Min Heap
10	100	0	75927	75927	94049.30114	306.6745851	433493	1016	8	Partial Min Heap
15	100	0	137786	137786	139768.7375	373.8565735	571518	1204	15	Partial Min Heap
20	100	0	149692	149692	161302.1021	401.6243296	703571	1191	16	Partial Min Heap
25	98	2	185223	176282	171544.6122	414.1794445	758264	432	21	Partial Min Heap
30	98	2	256114	251337	233037.8968	482.7399889	1071541	447	30	Partial Min Heap
35	99	1	340665	339504	285230.3981	534.0696566	1177672	756	39	Partial Min Heap
40	99	1	343828	347301	327607.7704	572.3703088	1355584	1911	41	Partial Min Heap
45	98	2	511076	410904	333326.0092	577.3439262	1565387	767	65	Partial Min Heap
50	97	3	623450	554971	413125.2115	642.7481711	1617056	10029	80	Partial Min Heap
55	87	13	620385	547538	456288.3687	675.4912055	2225831	5159	82	Partial Min Heap

Table 17: Map Size 512*512 using K=11 Full Data

REFERENCES

[1] (1964). Algorithms. *Commun. ACM*, 7(6):347–349.

[2] Aviram, N. and Shavitt, Y. (2015). Optimizing dijkstra for real-world performance. *Networking and Internet Architecture - arXiv preprint arXiv*.

[3] Botea, A., Müller, M., and Schaeffer, J. (2004). Near optimal hierarchical pathfinding. *Journal of game development*, 1(1):7–28.

[4] Cazenave, T. (2006). Optimizations of data structures, heuristics and algorithms for path-finding on maps. *2006 IEEE Symposium on Computational Intelligence and Games*, pages 27–33.

[5] Cherkassky, B. V. and Goldberg, A. V. (1996). Heap-on-top priority queues. *Society for Industrial and Applied Mathematics*, 96(42).

[6] Cui, X. and Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.

[David Rutter] David Rutter, M. C. G. T. Why should i use a closed list in the a* pathfinding algorithm? Accessed: 2018-11-16.

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

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

[10] Erdtman, S. and Fylling, J. (2008). Pathfinding with hard constraints - mobile systems and real time strategy games combined.

[11] Fredman, M. L., Sedgewick, R., Sleator, D. D., and Tarjan, R. E. (1986). The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129.

[12] Goldberg, A. V. and Silverstein, C. (1997). Implementations of dijkstra's algorithm based on multi-level buckets. In *Network optimization*, pages 292–327. Springer.

[13] Graham, R., McCabe, H., and Sheridan, S. (2003). Pathfinding in computer games. *The ITB Journal: Article 6*, 4(2):6.

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

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

[16] Holte, R. C., Perez, M. B., Zimmer, R. M., and MacDonald, A. J. (1996). Hierarchical a *: Searching abstraction hierarchies efficiently. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, pages 530–535. AAAI Press.

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

[18] Martelli, A. (1977). On the complexity of admissible search algorithms. *Artificial Intelligence*, 8(1):1–13.

[19] Mathew, G. E. and Malathy, G. (2015). Direction based heuristic for pathfinding in video games. pages 1651–1657.

[20] Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA.

[21] Rihani, H., Sanders, P., and Dementiev, R. (2014). Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *arXiv preprint arXiv:1411.1209*.

[22] Sturtevant, N. and Buro, M. (2005). Partial pathfinding using map abstraction and refinement. In *AAAI*, volume 5, pages 1392–1397.

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

[24] Zhang, Z., Sturtevant, N., Holte, R., Schaeffer, J., and Felner, A. (2009). A* search with inconsistent heuristics. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 634–639.

VITA AUCTORIS

NAME:	Mohsen Tavakoli
PLACE OF BIRTH:	Mashhad, Iran
YEAR OF BIRTH:	1993
EDUCATION:	Islamic Azad University of Mashhad, B.Sc., Information Technology, Mashhad, Iran, 2015
	University of Windsor, M.Sc in Computer Science, Windsor, Ontario, 2018