9-17-2019

Performance of Buckets versus Min-Heap in the A* Search Algorithm

Sheeba Mohanraj
University of Windsor

Performance of Buckets versus Min-Heap in the A* Search Algorithm

By

Sheeba Mohanraj

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

2019

Performance of Buckets versus Min-Heap in the A* Search Algorithm

by

Sheeba Mohanraj

APPROVED BY:

M. Hlynka
Department of Mathematics & Statistics

X. Yuan
School of Computer Science

S. Goodwin, Advisor
School of Computer Science

September 17, 2019

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 anyones 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 the search for the least cost route between two points on a map. Given a map with a start node and a goal node, the pathfinding algorithm begins from the start node and searches through the map exploring its neighbours until it finds a path to the goal node. A* is the most popular pathfinding algorithm which uses heuristics for finding the least cost path from the start node to the goal node. The A* algorithm uses two parameters g cost(the actual cost of reaching the current node from the start node) and the h cost(the estimated cost of reaching the goal node from the current node). F cost is the sum of the g cost and the h cost. At each iteration, the A* algorithm chooses the node with the lowest f cost from the open list to be expanded. This node will be removed from the open list and added to the closed list. This process is repeated until reaching the final node. The A* algorithm for pathfinding is affected by its data structures, particularly by the frequent insertions and deletions in the open list. Min-Heap is the commonly used data structure for implementing the open list for A* algorithm, which takes O(log n) for insertion and deletion. Since this is very expensive, we explored the idea of implementing the open list with Buckets, which was initially introduced for improving the performance of Dijkstra's algorithm. We found that the bucket data structure produces better results as it takes O(1) for inserting a node into the open list and O(bucketsize) for deletion. We compared the performance of both the algorithm under various factors to determine which was performing better.

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 ideas for my thesis. Thanks to his plenty of guidance and encouragement, I had a great time in researching the field of pathfinding. It is my great pleasure to get an opportunity to be his student and work with him.

I would also like to thank my committee members Dr.Hlynka and Dr.Yuan for taking their time to review my thesis and for attending my thesis proposal and defense. Thanks to their valuable suggestions and guidance for improving this thesis.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Problem Statement

In any computer controlled game, the most basic requirement of an agent is to be able to successfully navigate through the game environment. The agents in the video games use the search technique, called pathfinding, to find the least cost route between any two points in the environment. The agent can be a single person, a vehicle, or a combat unit in the genre of action game, simulator or a role-playing game. Real-time strategy games often have characters sent on missions from their current location to a predetermined or player determined destination. The pathfinding problems include transit planning, telephone traffic routing, maze navigation and robot path planning[23]. If there are no additional constraints on the solution other than the primary task of finding the least cost path, there are very simple search algorithms that will always find the best path, if there exists one. Although these algorithms can find the least cost path, they are often suboptimal solutions in terms of the resources required for performing its operations.

The problem of pathfinding in commercial computer games has to be solved in real-time, often under constraints of limited memory and CPU resources[24][2]. Early solutions to the problem of pathfinding in computer games, such as depth first search, iterative deepening, breadth first search, Dijkstra's algorithm, best first search, A* algorithm, and iterative deepening A*, were soon overwhelmed by the sheer exponential growth in the complexity of the game. More efficient solutions are required so as to be able to solve pathfinding problems on a more complex environment with limited time and resources. Therefore, in this thesis, we are researching on the optimal solution for pathfinding problem on the basis of time and memory[9]

1.2 Thesis Motivation

For solving pathfinding problems, a universal problem-solving mechanism, search, is used in artificial intelligence. Search algorithms are categorized into two types; informed search and uninformed search. In the uninformed search strategy, also known as blind search, searching is performed without having any knowledge about the goal state[11]. Hence, the agent ends up searching all the possible paths in the search space until reaching the goal state. This leads to over usage of memory and time for finding the final path. Breadth-first search, Depth-first search and Iterative deepening search are some of the uninformed search strategies. To improve the efficiency of the search, the knowledge about the estimated distance(heuristic function) from the current state to the goal state was introduced. By adding the g cost, which is the cost of the path from the start node to the current node, and the h cost, which is the estimated cheapest cost from the current node to the goal node, we get the f cost that determines which node is to be expanded. Best-first search(greedy search), Dijkstra's algorithm, Iterative Deepening A* search and A* search are the well-known informed search strategies[20].

A* search is one of the best and popular technique used in pathfinding and graph traversals. The A* algorithm has been modified into different versions by the researchers to provide better performance. Among the various researches done in improving the efficiency of A*, the number of operations performed for inserting and deleting the nodes from the open and the closed lists have been the main focus, as the cost of the A* algorithm is based on the operations done in these lists.

The motivation of this thesis is to reduce the cost of frequent insertion and deletion of nodes in the open and the closed lists. There are so many data structures to test and compare results for implementing the open list in A* search. Min-Heap is the commonly used data structure for implementing the open list. We have used the bucket data structure for reducing the time taken for the overall operations in the open list. In addition, the cost of the path is reduced by calculating the heuristics using Octile distance which allows the agent to travel in 8 directions in the game environment compared to Manhattan distance. As in the Manhattan distance, if it takes two steps(horizontal and vertical movement) to reach the goal it can be replaced by one step(diagonal movement) using Octile distance.

1.3 Thesis Contribution

The A* algorithm maintains two lists: open list and closed list, which requires certain operations to find the optimal path. The amount of time and memory required for maintaining the open list and the closed list determines the performance of the algorithm. These two lists can be created using various data structures, in which, Min-Heap is one of the popular data structure used for performing operations in the open list. However, it takes O(log n) for insertion and deletion as it has to perform bubble-up and bubble-down for sorting the list. The insertion operation occurs every time the neighbour nodes are added to the open list, and the deletion operation occurs every time the node with the lowest f cost is deleted from the open list.

Another data structure that we used for implementing the open list and the closed list for the A* search, is the bucket data structure. Buckets performs the insertion operation with O(1) in the best case scenario and O(bucketsize) for the deletion operation. However, the bucket data structure was introduced for improving the Dijkstra's algorithm and not many researchers have worked on using it for the A* algorithm. The performances of both the data structures were compared and analyzed based on various factors to determine which was performing better and under what conditions. We expect buckets to outperform min-heap if the size of the buckets can be kept close to O(log n). This is due to there being 8 times as many insertions as deletions when eight-directions moves are allowed.

1.4 Thesis Organization

This thesis is organized into 6 chapters. The first chapter covers the introduction to the concept of pathfinding, and gives a brief explanation about the problem statement, the motivation for this thesis and the thesis contribution. The second chapter is about the background research and literature review. It discusses about the various graph representations, methods for calculating the heuristic function, and the concept of A* algorithm and its operations. It also explains some important data structures used for implementing open list and the closed list. The third chapter introduces the data structures used in this thesis for implementing A* and various factors considered in producing better results. The fourth chapter gives the outline of the experiments conducted, its results and the analysis. In this chapter we compare the results of both the data structures used for the implementation. The fifth chapter provides the summary of our analysis and concludes our work and the sixth chapter proposes the future work which can be done to extend our thesis.

CHAPTER 2

Background and Literature Review

2.1 Graph Representations

Search spaces of pathfinding problems are commonly represented as graphs, where states associated with the search space are represented by graph nodes, and the transition between states is captures by graph edges[10]. The architecture we choose for a game will help us determine the features that the game can support easily, and the features that will require significant effort to be implemented. It is important to know that, for most games, all feasible path planning architectures are abstractions of the space through which characters can walk in the game. This is because the physics that used to simulate the world are not directly used as the path planning representation. So, in some sense, much of the debate here is related to what representation most closely matches the underlying physics of the game world. Some of the common representations of the maps are Grids, Navigation meshes and waypoints[22].

2.1.1 Waypoints

Waypoints are a set of points that refer to the coordinates in the physical space. They were being widely used before the popularity of the navigation meshes grew. Waypoints are often used for pathfinding as they can be specific to each path or be a part of the game map. In real-time strategy games, the players can manually add the path specific waypoints. The shortest paths between each pair of waypoints can be pre-computed if the waypoints were small.

The disadvantage of waypoints is that, the lack of walkabe edges in the graph can affect the quality of the path. On the other hand, too many walkable edges will impact storage and planning complexity[5].

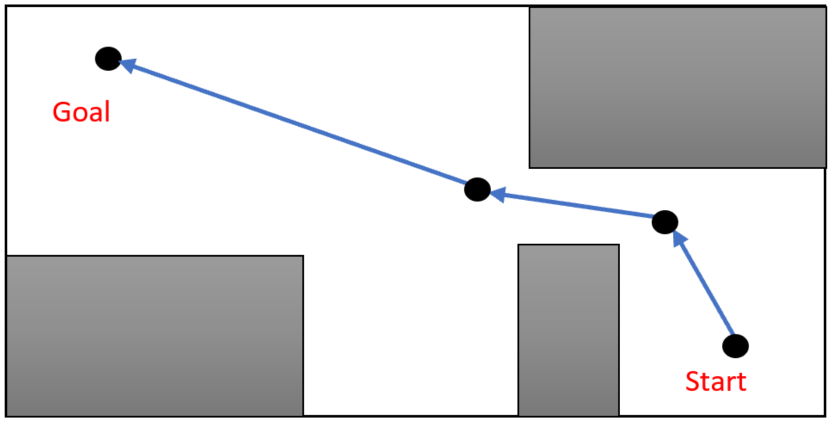

Fig. 1: Waypoints

2.1.2 Navigation Mesh

Navigation Mesh, or Navmesh, is a three-dimensional object in the game world which covers every space where entities can move around. It represents the world using convex polygons with which it is easier to correctly perform smoothing both before and during movement. In a two-dimensional environment, navmesh covers the unblocked area of a map with a minimal set of convex polygons[3].

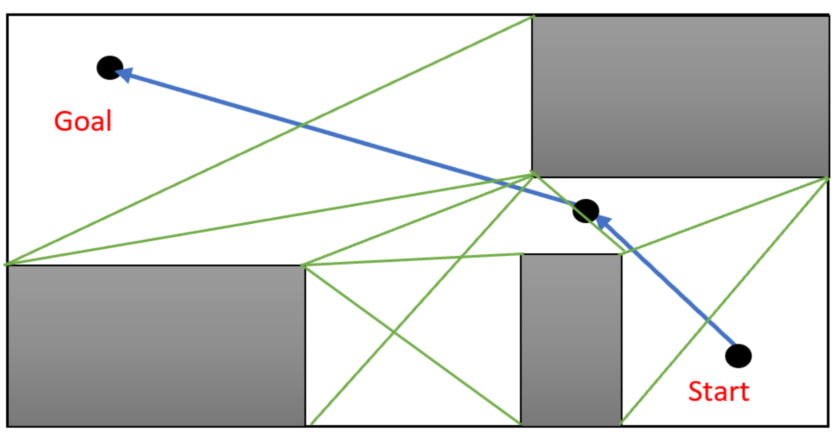

Fig. 2: Navigation Mesh

Path planning on navigation meshes is usually fast, as the representation of the world is fairly coarse. But this does not impact path quality as characters are free to walk at any angle. Although navmesh can represent large space with just few polygons, the time required to implement a navmesh is significant. Unlike waypoints where change can be made easily if known ahead of time, in navmesh, changes can be difficult or expensive to implement[6].

2.1.3 Grids

A grid is composed of vertices or points that are connected by edges to represent a graph. It represents a world via an array of blocked(non-traversable) and unblocked(traversable) cells. Some of its implementations can include slope, terrain type or other meta-information which is useful for planning[21]. The terrain costs in grids are easy to dynamically update and the localization of grids is easy. However, the grids are memory intensive in large worlds.

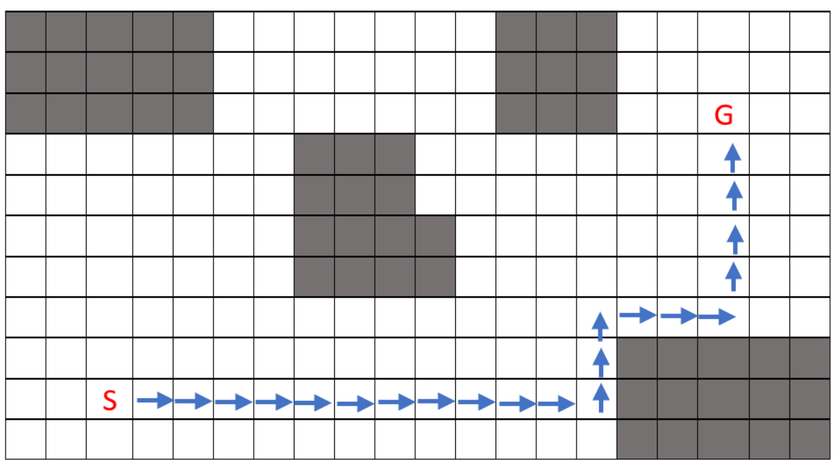

Fig. 3: Square Grid

Grids are easy to implement and can be easily modified. Some of the common grid types are square, triangular and hexagonal. Square grids are the simplest and the most popular grid graphs used for computer game developments. We have also used square grids to run our experiments as it is easy to implement and produces better results.

2.2 Heuristics

In uninformed search strategies such as Breadth First Search, Depth First Search and Iterative Deepening Search, they do not take the distance to the goal into account. They perform search operations without having any information about reaching the goal state until they stumbled upon one[19]. The heuristic function provides an informed way to guess which neighbour of a node will lead to a goal. It is an estimate of the cost of the optimal(cheapest cost) path from node n to the goal node. The performance of heuristic search is commonly measured by the number of performed node expansions. Some of the commonly used functions for calculating heuristics are Manhattan distance, Euclidean distance and Octile distance.

2.2.1 Manhattan Distance

Manhattan distance is often used to calculate the heuristic function for grid-based maps. The movement is restricted to only 4 directions(vertical and horizontal).

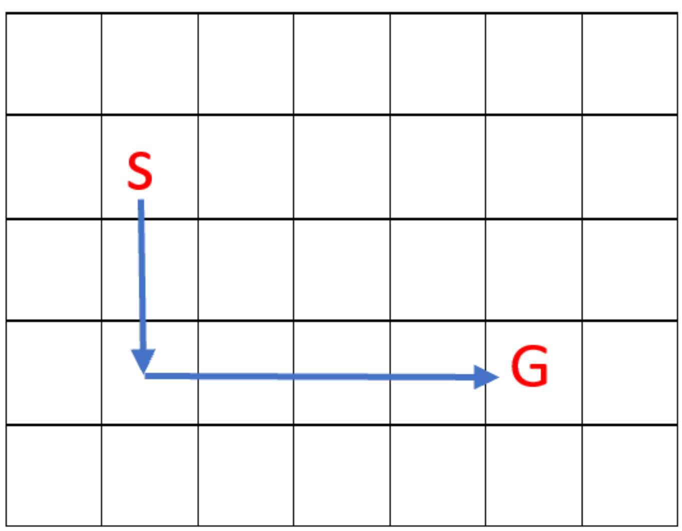

Fig. 4: Manhattan Distance

The Manhattan distance could be the best choice when the agent is only allowed to move in 4 directions. It is calculated by taking the sum of the X and Y distances to the destination.

$$h(n) = |x1 - x2| + |y1 - y2| \tag{1}$$

Manhattan distance provides a good estimate of the time to reach the destination on an open map. However, if the shortest path to the destination is circuitous, then the Manhattan distance becomes a poor estimate.

2.2.2 Euclidean Distance

The Euclidean distance is the straight-line distance between two points in the search space. Since the cost is calculated based on the direct movement by the agent from the starting point to the goal point, Euclidean distance produces accurate results in case of no obstacles. As Euclidean distance is shorter than Manhattan distance or the Diagonal(Octile) distance, we will get shorter paths.

$$h(n) = \sqrt{|x1 - x2| + |y1 - y2|} \tag{2}$$

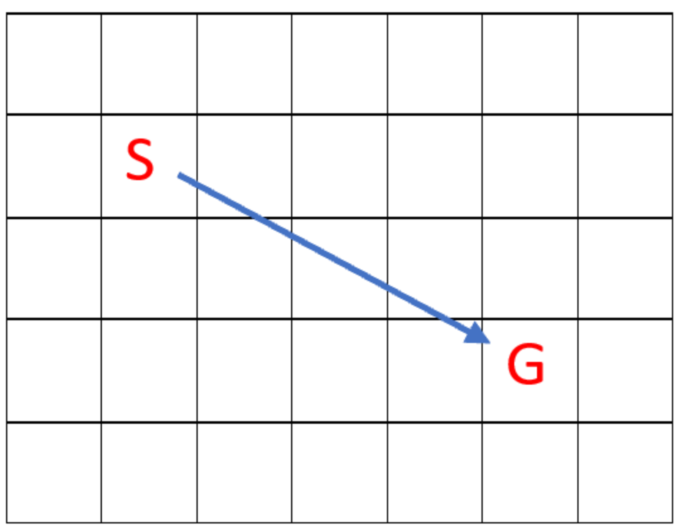

Fig. 5: Euclidean Distance

2.2.3 Octile Distance

Octile distance is the extension of Manhattan distance that allows diagonal movements[1]. It is commonly used in grid-based maps.

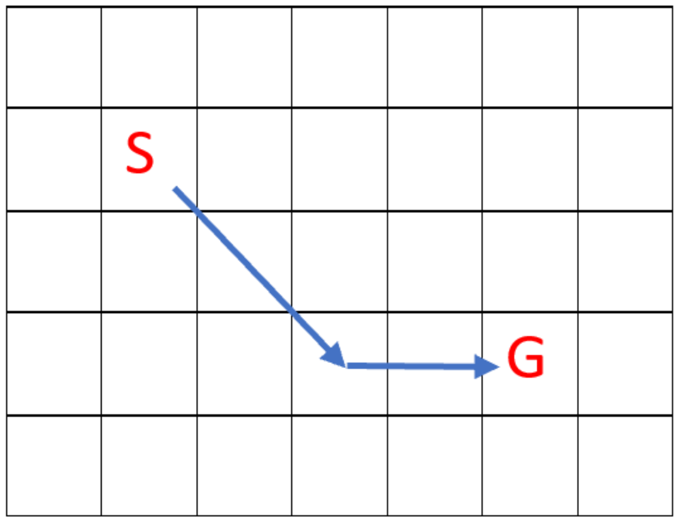

Fig. 6: Octile Distance

The Octile distance is calculated using the following formula,

$$\text{h(n)} = \text{max((x1-x2), (y1-y2))} + (\sqrt{2} - 1) * min((x1 - x2), (y1 - y2))$$

For our experiments, we used 1.4 as the value of the square root of 2, as it is the the commonly used value for experiments.

Octile distance gives a shorter path compared to the Manhattan distance as two step costs(horizontal and vertical movement) in Manhattan distance can be replaced by one step cost(diagonal movement) in Octile distance. For our research, we have used Octile distance for calculating the heuristic function on grid-based maps. We fixed the movement cost for horizontal and vertical movement to be 10 and the movement cost for diagonal movement to be 14.

2.3 A* algorithm

A* is a greedy search algorithm that can be used to find solutions to many problems with pathfinding being one of them. It is the most popular and widely used AI pathfinding informed search algorithm[14]. In a square grid having many obstacles with a given starting node and a goal node, A* finds all the possible paths to reach the goal node. This is done based on certain functions such as g(n), h(n) and f(n).

Algorithm 1 A* Algorithm

```
Input: A Graph G(V, E) with start node start and end node goal
Output: Least cost path from start to goal
Initialize:
    open_list = {start}
    closed_list = { }
    g(start) = 0
    f(start) = heuristic_function(start, goal)
while open_list is not empty do
  current = the node in open_list having the lowest f value
  if current = goal then
    return "Path found"
  end if
  open_list.delete(current)
  closed_list.insert(current)
  for each neighbour of current do
    if neighbour in closed_list then
      continue
    end if
    if neighbour not in open_list then
      open_list.insert(neighbour)
    end if
    if g(current) + distance(current, neighbour) < g(neighbour) then
      g(neighbour) = g(current) + distance(current, neighbour)
      f(neighbour) = g(neighbour) + heuristic_function(neighbour, goal)
      neighbour.setParent(current)
    end if
  end for
end while
return "Path not found"
```

In a given graph G(V,E), where V refers to the vertices and E refers to the edges, the search begins from the start node and reaches a node n on its way towards the goal node. The movement cost to move from the start node to the node n gives us the g cost, represented by g(n) and the estimated movement cost to move from the node n to the goal node gives us the h cost, represented by h(n). The sum of the two movement costs g(n) and h(n) gives us the total cost f(n). Based on the f cost, the node which is to be expanded next is chosen. In the A* search, two lists are maintained: the open list and the closed list. The open list stores the nodes that are to be expanded and the closed list stores the nodes that have already been expanded.

In Algorithm 1, the search begins by adding the start node to the open list. The total cost f(n) is calculated by the following formula,

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

The nodes adjacent to the start node are added to the open list, and the node with the lowest f cost is selected from the open list to be expanded next. The expanded node is then removed from the open list and added to the closed list. While the open list is not empty, the next node with the lowest f cost is selected. If the current node was the goal node, the process terminates and back-traces the path from the goal node to the start node. If the current node was not the goal node, it is removed from the open list and added to the closed list. Then all the possible neighbours of the current node is checked if it is present in the closed list and the open list. If the neighbour nodes are not present in either of the lists, it is then added to the open list and the process continues.

On the other hand, if the neighbour node is present in the open list, then the f value of the neighbour will be calculated and compared to the node(duplicated node) already in the open list. If the neighbour's f value is lesser than the duplicated node in the open list, the duplicated node will be replaced by the neighbour node in the open list. However, if the f value of the neighbour node is greater than the duplicate node, the neighbour node will be discarded. This process is repeated until the goal node is reached. A* algorithm is always said to find the optimal path if one exists. The different data structures used for the open list also determines the memory and the time efficiency of the algorithm. In this thesis, we have analyzed the different usages of the data structures and how it has impacted on the results.

2.4 Properties of A* Algorithm

2.4.1 Optimality

There are several search algorithms for solving the problem of pathfinding, but not all the algorithms guarantee finding the least cost path from the start node to the goal node. A* search algorithm always finds all the possible least cost paths from the start node to the goal node. This path is called an Optimal path. If A* returns a solution, that solution is guaranteed to be optimal(least cost) if all the cost estimates(heuristics) are optimistic[15]. Among all the optimal algorithms that begins from the start node and uses the same heuristic h, A* expands the minimal number of paths[7].

2.4.2 Admissibility

A search heuristic h(n) is an estimate of the cost of the optimal path from the node n to the goal node. A* is optimal if h(n) is an admissible heuristic, that is, provided that h(n) never overestimates the cost to reach the goal[8][26]. Admissible heuristics are by nature optimistic because it assumes the cost of solving the problem is less than it actually is. Since g(n) is the exact cost to reach n, it is clear that f(n) never overestimates the true cost of a solution through n. The heuristic function h(n) is admissible if for every node n,

$$h(n) <= h^*(n)$$

Straight-line distance is an example of an admissible heuristic because the shortest path between any two points is a straight line, so the straight-line distance cannot be an overestimate.

2.4.3 Consistency

A heuristic function h(n) is consistent if for every node n and every successor n' generated by an action a[18],

$$h(n) <= c(n,a,n') + h(n')$$

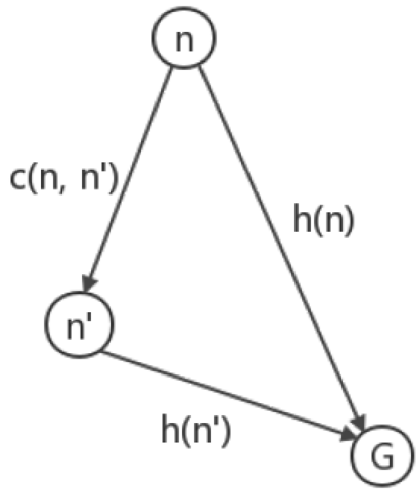

Fig. 7: Consistency Diagram

This is a form of triangle inequality, which depicts that each side of a triangle cannot be longer than the sum of the other two sides. This implies that f(n) never decreases along a path from the root. Considering f* to be the cost of the optimal solution path. A* expands all nodes n with f(n) <f*, and may expand some nodes right on the goal contour before selecting a goal node. The optimality of the A* algorithm can be proved by the following theorems.

Theorem 1 *If the heuristic is consistent, f value along any path is non-decreasing[7].*

Based on Theorem 1, the heuristic is consistent when the f values of the expanded nodes by the A* algorithm are never decreasing. This theorem also indicates that the A* algorithm will not revisit(no need to check for the duplicates) the nodes that have already been visited and added to the closed list.

Theorem 2 *If the heuristic is admissible and consistent, A* could find an optimal path[7].*

The heuristic is said to be admissible when only those nodes whose f values are less(or equal) to the optimal cost path C*, that is, f(n) is less than or equal to C* are expanded. Based on both the theorems, we can prove that our implementation of A* is admissible and consistent, so it provides us an optimal solution.

2.5 A* Operations

The A* algorithm maintains two lists for performing its operations. One is the open list and the other is the closed list. The open list keeps track of those nodes that need to be examined, while the closed list keeps track of those nodes that have already been examined. The open list performs operations such as, insertion, deletion, contains and update. For each iteration, the node with the lowest f-cost is removed from the open list(deletion), and the presence of their neighbour nodes are checked in the open list before adding them(contains). Then those neighbour nodes are added to the open list(insertion). After inserting the nodes to the open list, they are sorted based on the f-cost(update). In this thesis, we focus on how these operations impact on the performance of the A* algorithm.

2.5.1 Open List

Initially, the open list contains only the start node, whereas the closed list is empty. After expanding the start node, it is removed from the open list and added to the closed list. Later the adjacent nodes of the start node are added to the open list and the nodes with the lowest f cost is selected from the open list, and it is expanded. After expanding, the current node is added to the closed list that maintains the list of nodes that have already been visited, and the neighbours of the current node is added to the open list. This process is done repeatedly in the main loop until the current node is the same as the goal node.

The performance of the algorithm mainly depends on the number of insertions, deletions, contains and update done in the open list. As the data structures have a major impact on the time complexity of the algorithm, in our research, we have used difference data structures: Min-Heap and Buckets for the implementation of the open and closed list.

2.5.2 Closed List

Comparing to the open list, the closed list does not perform as many operations. The nodes which are expanded are added to the closed list to make sure that those nodes do not have to be expanded again by the agent. Before every insertion of the node in the open list, the closed list is checked for the presence of that node in it. Since the closed list does not perform much operations, it does not necessarily impact the performance of the algorithm.

2.6 Data Structures

In this chapter, we discuss about the various data structures used for the implementation of open and closed list. Among the various data structures, some of the commonly known ones are the array(sorted and unsorted), hash table, Min-Heap and buckets. The time and memory required for performing the operations of the open list determines the efficiency of the A* algorithm. In this thesis, we have used Min-Heap and bucket data structure for the implementation.

2.6.1 Array

The open list can be implemented using array in two ways, sorted and unsorted. In the unsorted array, the number of operations required for inserting a node into an array takes O(1) as it is stored in the last location in the array. For the deletion of the node, it takes O(n) as it has to scan the unsorted array to find the node to be deleted. It takes O(n) for the checking the presence of the node in the open list and updating it, as it has to scan through the array to find the node.

In the sorted array, the number of operations required for inserting a node in the array takes O(n) as it has to sort the array every time a node is added. However, for the deletion of the node, it only takes O(1) because the array is already sorted and the location of the node with the lowest f-cost will be at the end of the array. When we use binary search for checking whether the node is in the array, we require O(log n) operations. If the node is already in the open list, it takes O(log n) to find the node and O(n) to update the node in the list.

2.6.2 Hash Table

In the hash table, the values are stored as index which is a key/value pair. However, in case the keys are larger, we have to convert the large keys into smaller keys by using hash functions. By using the key, we can access the element in O(1) time. Hence insertion of a node will take O(n). Hash tables become quite inefficient when there are collisions; when two keys are hashed to the same slot(index). Due to this collision, the scanning of the array list for the node consumes more time and leads to O(n) operations for removing the node from the array [16].

2.6.3 Min-Heap

Min-Heap is the most commonly used data structure for the implementation of the open list. It is a complete binary tree in which the value of the parent node is always smaller than or equal to its children node[17]. This data structure allows us to perform the insertion and deletion in logarithmic time. Min heap is mostly implemented using array. For inserting and removing a node from the array, we perform two operations, up-heap(bubble up) and down-heap(bubble down).

For inserting a node to the heap, the new node should be placed at the last position just beyond the rightmost node at the bottom level of the tree or as the leftmost position of a new level, if the bottom level is already full. After inserting it at the last position, the value of the new node is compared to the value of its parent node to verify if its value is larger than the parent node. If not, the positions will be swapped. This insertion process takes O(log n).

For removing a node from the heap, we delete the node at the first position(root) of the tree and replace the position with the node at the left position at the bottommost level. After replacing, we compare the value of the node with its children node to check if the value is smaller than its children. If not, we the positions will be swapped. Hence, the deletion of the node from the heap takes O(log n).

For checking if the open list contains the node, it takes O(n) as it has to scan through the entire tree. If the node is already in the open list, it takes O(n) to find it and O(log n) to update its value in the open list.

2.6.4 Multilevel Buckets

A bucket is a list of nodes whose labels fall within a given range. In a multilevel bucket, the lowest bucket level is 0 and the highest bucket level is k-1[13]. When using the bucket for the implementation of open list, if each node is inserted into a separate bucket it only takes O(1) operation for insertion of the node. Whereas removing the node from the bucket depends on the size of the bucket. In this case, if n number of nodes are entered in each bucket, it takes O(1) to remove the node at the lowest bucket level. Although, scanning through the buckets to find the node still takes O(n) to go through the entire tree. In this thesis, we have discussed about the various possibilities of performing the insertion and deletion operations in the bucket, and have compared it with Min-Heap data structure.

2.7 Summary

From the above discussions about the various data structures that can be used for the implementation of the open list, it is difficult to narrow down to one specific data structure that would perform well in all the four operations of the open list and the closed list. Each data structure has its own pros and cons in every operation of the open list.

Data Structure	Insert	Delete	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)
Buckets	O(1)	O(k) or O(n/k)	O(n)	O(n) or O(n)+O(n/k)
Min-Heap	O(log n)	O(log n)	O(n)	O(n)+O(log n)

Table 1: Time Complexity Comparison of Different Data Structures

In the above table, when we look at the sorted and unsorted array, the insertion operations takes only O(1) in the unsorted array as it is easier to insert a node in an unsorted list whereas it takes O(n) in the sorted array since the array list is sorted every time an element is inserted into the array. However, the deletion operation takes O(n) in the unsorted array and O(1) in the sorted array since in the sorted list, it is easier to locate the element compared to the unsorted list where we need to compare the value of each element to find the lowest valued node.

For the insert operation, among the five data structures we have compared, sorted array is the highest with O(n) operations for performing the insert. When looking at the hash table, the insertion of the node in the list seems to take O(n) under the worst-case scenario, which is still lesser than the Min-Heap. But the hash table still has its own disadvantages when it encounters collision. The Min-Heap data structure seems to take more time to perform insertion and deletion compared to the sorted array, unsorted array and the hash table, which is O(log n).

The bucket data structure seems to be performing better than the Min-Heap in certain cases, though the size of the buckets and the number of nodes determine the time complexity of the bucket data structure. In this thesis, we discuss in details about the differences in the time complexity of using the buckets and Min-Heap for the implementation of the open list.

CHAPTER 3

Min-Heap and Buckets

3.1 Motivation

Based on our research, we came to know that there are various ways in which pathfinding problems can be solved. Different algorithms have been proposed and many improvements have been made to them to optimize the path and reduce the time and space complexities. Some of the popular search algorithms such as Dijkstra's algorithm and A* algorithm have also been modified to serve this purpose. For our research, we focused on improving the performance of the A* search algorithm. When it comes to A* algorithm, the use of heuristics have made a noticeable improvement in finding optimal(lowest cost) paths. Among the commonly used methods for calculating the heuristic function, in our research, we have used Octile distance which gives a shorter path compared to Manhattan distance, as the Octile distance allows the movement of the agent in 8 possible directions(north, south, east, west, northeast, northwest, southeast and southwest), whereas Manhattan distance allows only 4-way movement(east, west, north and south). For our experiment, we have assigned the step cost of diagonal movement to be 14 and the step cost of horizontal and vertical movement to be 10.

Many of the researches done for improving the A* algorithm have focused on open list as it takes up too much resources for performing its operations(insert, delete, contains and update). As we discussed in the previous chapter, there are many data structures introduced for improving the efficiency of the open list. Though every data structure has its own advantages and disadvantages, our focus is on Min-Heap and buckets. In the following example, we have taken a square grid with a start node and a goal node.

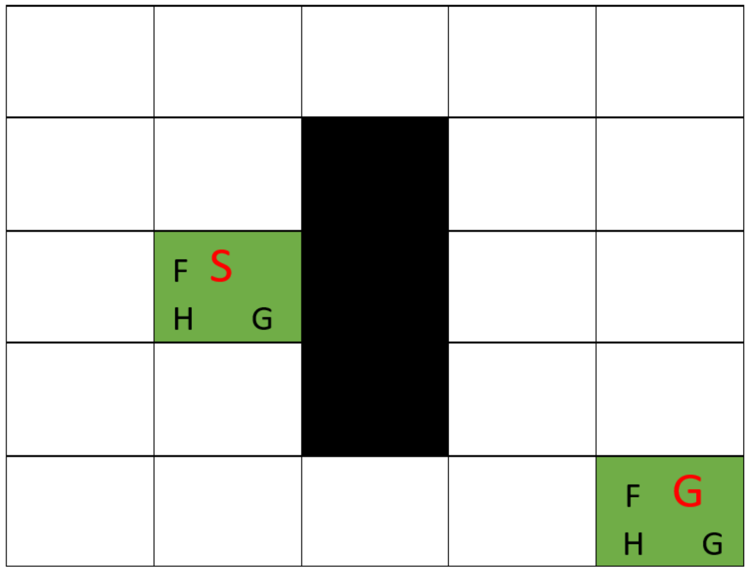

Fig. 8: An example for pathfinding using A* algorithm

In Fig 8, the white nodes represent the traversable(unblocked) nodes and the black nodes represent the block/obstacle in the grid. Initially, we begin our search from the start node that is marked as S in Fig 8. The start node is first added to the open list, and the closed list will be empty. After expanding the start node, the neighbours of the start node are added to the open list if they are not already in the open list, and the start node is added to the closed list as shown in Fig 9. The blue nodes represent the neighbour(adjacent) nodes which are added to the open list. Among the nodes added to the open list, the node with the lowest f cost is selected for expansion.

In the below example, the node with the lowest f value(14) is selected for expansion. Before adding this current node to the closed list, we verify if the closed list already contains the current node. If it contains the current node and if the f cost of the current node is smaller than the duplicate node in the list, the value will get updated. If not, the node will get added to the closed list.

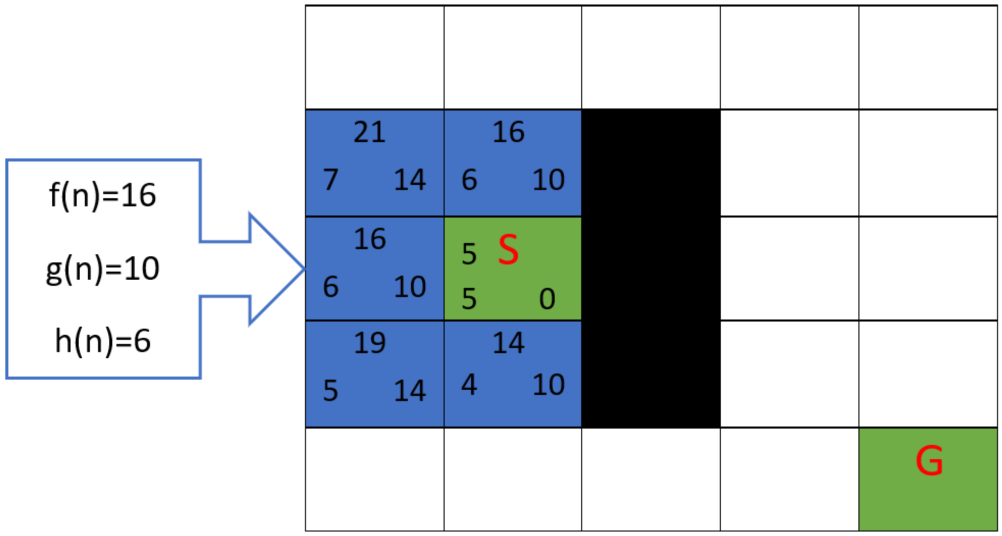

Fig. 9: An example for pathfinding using A* algorithm

As shown in Fig 10, after the node with f value 14 is selected to be expanded, the f values of its adjacent nodes are calculated and added to the open list if they are not already in the open list or the closed list. From the existing nodes in the open list, the node with the smallest f value is again selected for the next expansion.

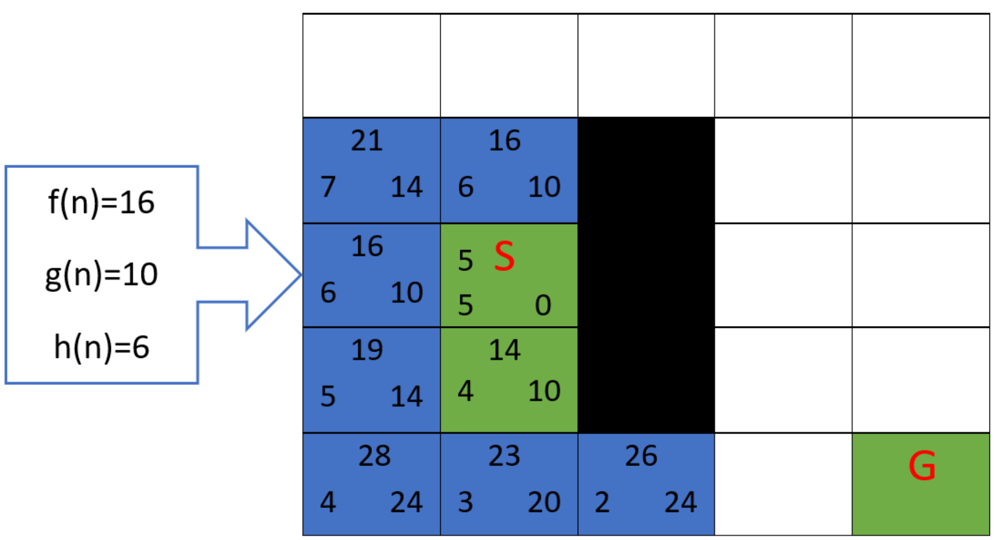

Fig. 10: An example for pathfinding using A* algorithm

This process is continued until the current node is the same as the goal node. Once the current node is the same as the goal node, we return the final optimal(least cost) path found. There are possibility of situations where the obstacles are higher in a map that no path can be found from the start node to the goal node. As we move closer to the goal node, the f cost is increased but the h cost(which is an estimated cost of reaching the goal node from the current node) is decreased.

In Fig 10, the green nodes represent the optimal(least cost) path found from the start node to the goal node. As the agent moves towards the goal, the g cost of the nodes increases while the h cost decreases with becoming zero at the goal node as seen in Fig 11. It can also be seen that there are some nodes which are not expanded in the grid. Since the A* algorithm always chooses the nodes with the lowest f value, the nodes whose f values are much higher will be ignored; it means those paths are longer and costlier to travel.

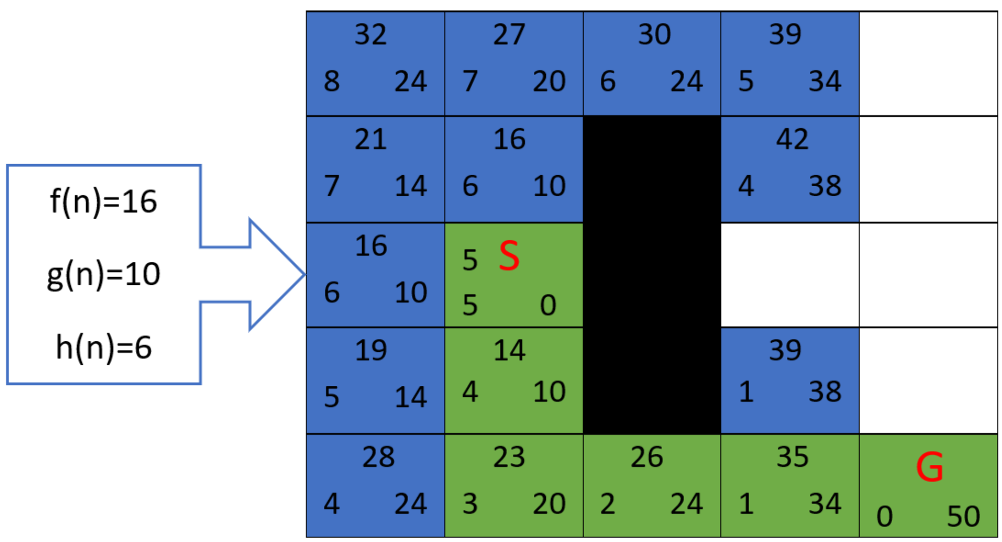

Fig. 11: An example for pathfinding using A* algorithm

So, there is no need for the agent to visit all the nodes in the grid to find the shortest(least cost) path. This is one of the advantages of using A* algorithm for pathfinding. The efficiency of the algorithm is determined by finding the optimal(least cost) path using the A* algorithm that requires a large number of insertions and deletions from the open list and the closed list. For this purpose, a data structure which can reduce the number of operations required for the insertion and deletion has to be used. This motivated us to use the commonly used Min-Heap data structure for A* algorithm and compare it with the Bucket data structure which was mainly introduced for the Dijkstra's algorithm.

When we insert a new node into the open list using the Min-Heap data structure, the node gets added to the last position at the bottom-level of the tree and is then compared with its parent; this process is called up-heap. Likewise, for removing a node from the open list, we perform the down-heap operation where the leaf node is moved to the first position(root) and the node in the root position(node with the lowest f cost) is removed. From this it is clear that the insertion and the deletion operations performed in the open list takes O(log n) using Min-Heap. This led us to consider the bucket data structure where the insertion might take O(1) in the best-case scenario and O(k) for deletion, where k is the size of the bucket. We compared the performance of both the data structures by measuring it based on various factors such as runtime, memory, number of operations and the number of buckets.

3.2 Min-Heap

Min-Heap, as we discussed in the previous chapter, is one of the popular data structures used for implementing the open and closed list. Although the closed list performs only insert and contains operations compared to open list which performs insert, remove, contains and update. A Min-Heap is a binary tree in which each node in the tree is smaller than its children. Each node in the tree has an index for identifying its location in the tree. We have stored the nodes of the heap in ArrayLists so that when we test our algorithm in larger maps, the elements can be added to the ArrayList as it does not require a fixed size to be declared when creating an object. In the following we discuss in detail about how the operations are formed by the Min-Heap.

3.2.1 Operations of Min-Heap

The four main operations of the open list such as: Insertion, Deletion, Contains and Update, and the two operations of closed list such as Insertion and Contains, can be performed using the Min-Heap data structure.

In Fig 12, we have taken an array of 6 elements which will be built into a Min-Heap. The elements in the array are stored in an unsorted manner. Each node has an array index to identify its position. The array index starts from location 0. The first element of the array, 2, is added to the first location of the Min-Heap. When adding the second element of the array to the heap, the second node(child node) is compared to the first node(parent). Since 5 is greater than 2, it is added to the second location in the heap. Likewise, other elements are added to the heap. Every time a node is added/removed from the heap, we check if the value of the current node is greater than its parent node.

Array Index	0	1	2	3	4	5
Input	2	5	7	8	10	9

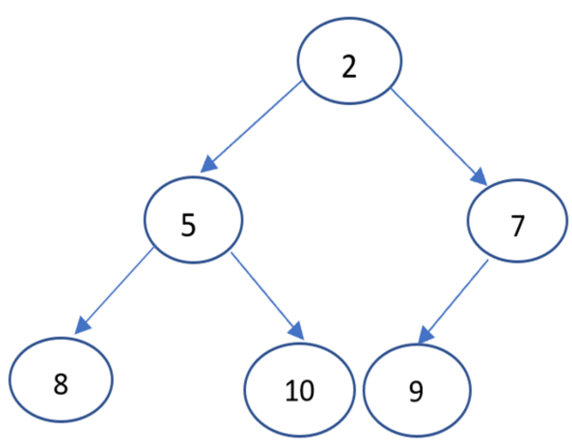

Fig. 12: Example of Min-Heap data structure

If not, we perform the up-heap operation for insertion and down-heap operation for deletion. In the following we explain in detail how the operations are performed based on Min-Heap data structure.

Insertion:

When inserting a node into the open list, according to the Min-Heap data structure, the new node is added to the position p just beyond the rightmost node at the bottom level of the tree or as the leftmost position of a new level, if the bottom level is already full. Every time a new node is added to the last position of the tree, its f cost is compared to the f cost of its parent. Min-Heap is a complete binary tree, where the parent node should always be smaller than its children nodes. When the condition is not satisfied, the position of the parent node gets switched with its child node. This swapping is continued until the condition is met.

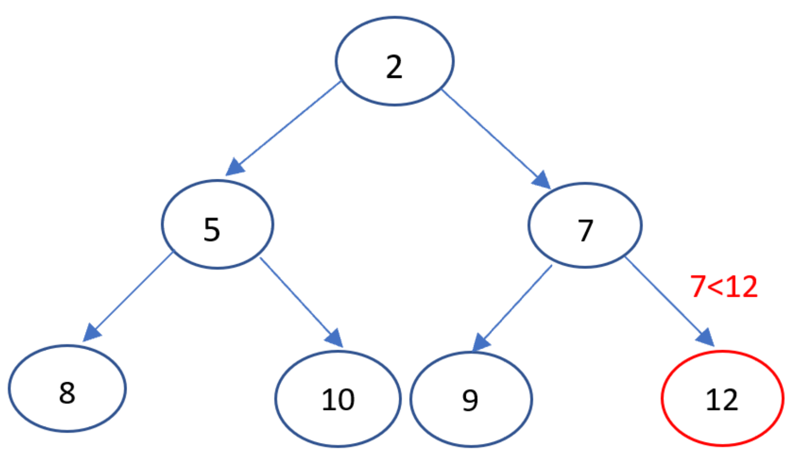

Fig. 13: Example of Min-Heap data structure

Fig 13 shows the example of insertion operation performed in the Min-Heap. When adding the new element 12 to the heap, it is added to the last position of the heap and its value is compared to the parent node. If it is lesser than the parent node, the positions of both the nodes will be swapped. In the above example, since the child(12) is greater than its parent(7), the position of the nodes will remain the same. If the child node was smaller than the parent node, the child node's position would have been swapped with the parent node and the current child node will again be compared with its parent node. This takes O(log n) operations for inserting the new node to the end of the heap as a leaf node and comparing its value to its parent node and swapping positions until it satisfies the condition of the binary tree.

Deletion:

When deleting a node from the heap, the leaf node at the last position p at the rightmost position at the bottom-most level of the tree is removed. To preserve the entry from the last position p, we add it to the root r of the tree. The value of the current node added to the root of the tree is compared to the value of its child node. If the value of the current node is greater than the child node, the positions of both the nodes get swapped. This process is continued until the tree satisfies the conditions of the binary tree.

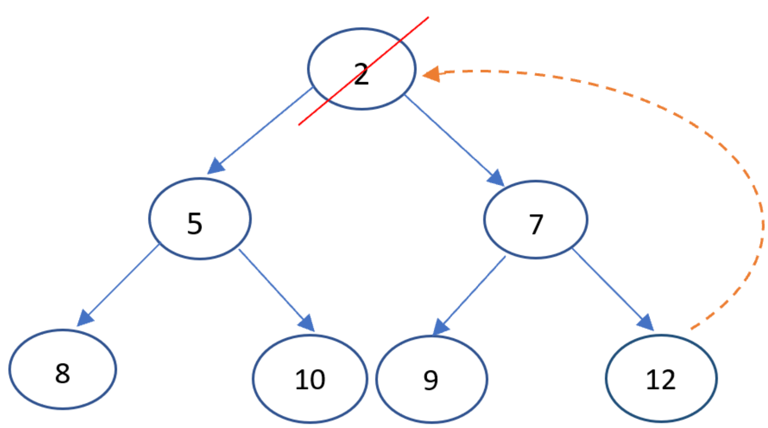

Fig. 14: Example of Min-Heap data structure

In Fig 14, the first element of the tree which has the smallest value is removed from the tree and the first position is replaced with the element 12 from the last bottom-most position of the tree.

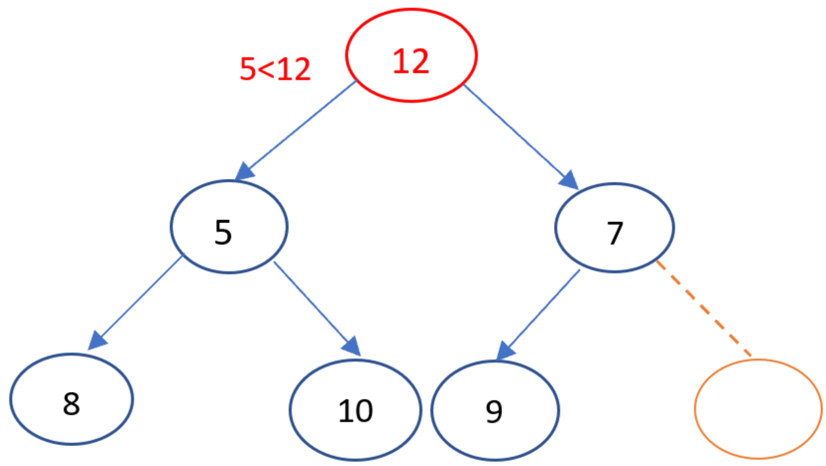

Fig. 15: Example of Min-Heap data structure

After replacing, the down-heap operation is performed where the parent node is compared to the child node to check if the parent node is lesser than its child node as shown in Fig 15 and Fig 16. This is continued until the heap satisfies the condition.

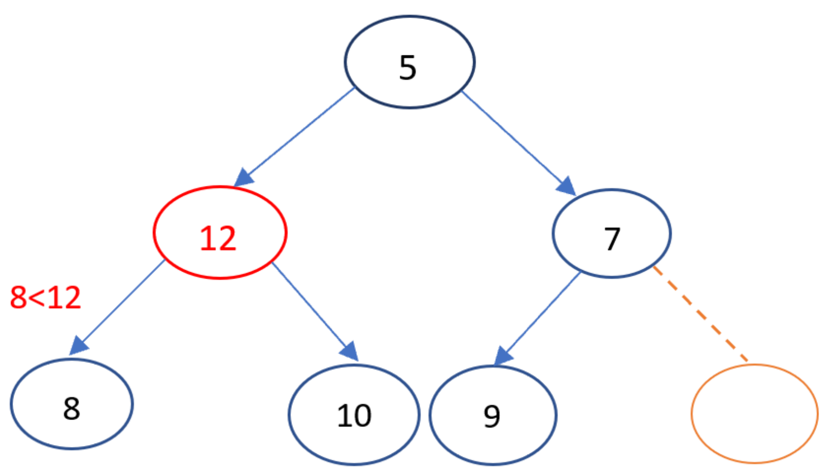

Fig. 16: Example of Min-Heap data structure

Based on the down-heap operation, the check is performed for verifying the values of the parent node and the child node. In Fig 15, since the value of the node in the first position, 12, is greater than its child node, which is 5, their positions get swapped in the tree. Then the current node, 12, is again compared with its child nodes whose values are 8 and 10. Since it is greater than the child node, the positions get swapped again until we get a binary tree.

The deletion operation takes O(log n) operations to remove a node from the tree as it has to remove the first node of the tree and replace its position with the node from the bottom-most level of the tree, and compare the values of the nodes with its child nodes and perform swapping wherever necessary.

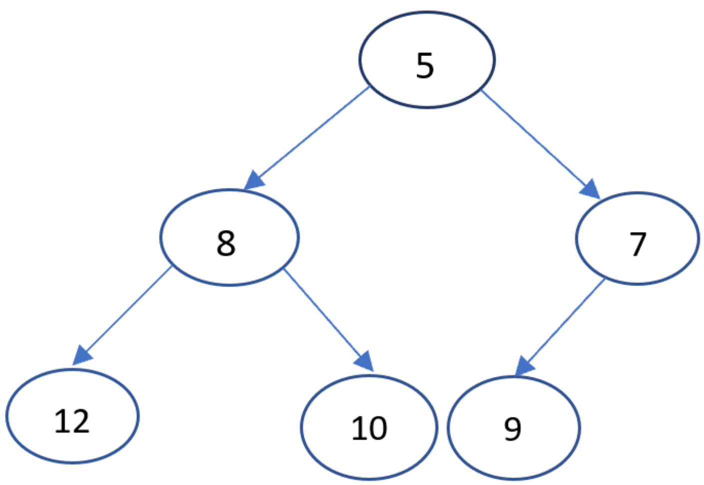

Fig. 17: Example of Min-Heap data structure

Contains:

The contains operation is performed for checking the existence of the current node in the open or the closed list. This way the duplicate entry of the node in the list can be prevented. When the contains operation is performed in the heap, before inserting the node into the heap, we check if the node is already present in the heap. This is done by comparing the node to the already existing nodes in the heap with the help of some attributes like a pointer to its predecessor or a specific location of the node in the map. If the node is not present, we then add the node to the heap. This operation takes O(n) as it has to check the entire tree from top or from bottom to locate the node we are searching for.

Update:

In the update operation, we check the heap for the existence of the node in the data structure and update its value if its value is smaller than the existing value. We search for the node in the heap(contains) and replace the node with the updated new value which is lesser than the current value. That is, we perform the contains operation for finding the current node in the tree and when the node is found, we compare the value of the current node with the already existing node. If the value of the current node is lower than the value of the already existing node(duplicate) node, we replace or update the node with its new lowest value.

This operation is expensive compared to the other operations of the open list, as it involves searching for the node in the entire tree(contains) and then updating the value of the node with its new lowest value found. The contains operation takes O(n) and updating its value takes O(log n).

3.3 Buckets

The bucket data structure is quite different from the Min-Heap in the way the nodes are stored in the open list. Buckets are sets arranged in a sorted manner. It stores all the nodes whose values fall within a certain range[25]. We maintain different buckets with different ranges. Each node is added to the bucket depending on the range the value of the node comes under.

As an example, we have taken an unsorted array of 8 elements which are assumed to be the f values of the nodes we come across while searching for the final optimal(least cost) path based on the A* search algorithm using the bucket data structure. As shown in Fig 16, we have an unsorted array of 8 elements(0-7) which will be added to the buckets. For these 8 elements we have taken 5 buckets with each bucket holding values of different ranges.

Suppose, the first bucket will store values between 0 and 10, second bucket will store the values between 11 and 20 and so on. The bucket size keeps growing based on the number of values present within the chosen range for each bucket. Like the Min-Heap data structure, the four operations(insert, delete, contains and update) of the open list are also performed using buckets. In the following, we discuss in detail about the four operations.

Array Index	0	1	2	3	4	5	6	7
Input	29	25	3	49	9	37	21	43

0
1
2
3
4

Fig. 18: Example of Bucket data structure

Insertion:

As we know, the operation of insertion is to insert a new node which has the possibility to be expanded, into the open list. We created different buckets for storing the nodes, with each bucket holding a certain range of values it allows the nodes to be inserted. Arraylists were created for storing the nodes in the buckets as it does not require a fixed size to be declared when creating an object. Every time a node is to be inserted into a bucket, we calculated the position of the bucket where the node should be inserted based on the bucket interval.

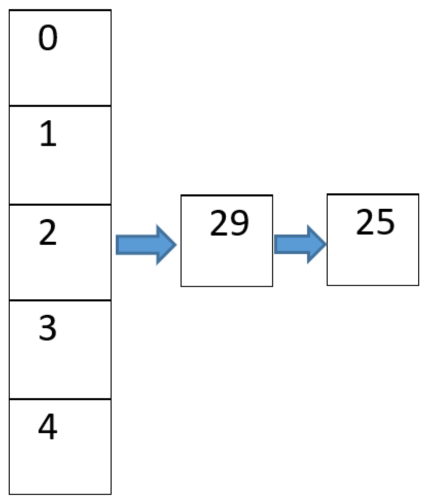

Fig. 19: Example of Bucket data structure

The insertion of the node in the bucket is quite straightforward compared to Min-Heap, as in the Min-heap we have to perform the bubble up(swapping) by comparing the value of the nodes with its child node. So, it takes O(log n) operations for inserting a node into a heap. The first element of the 29 is added to the bucket 3 as shown in the Fig 19. Followed by the second element 25 in bucket 3, since each bucket holds the range of 10.

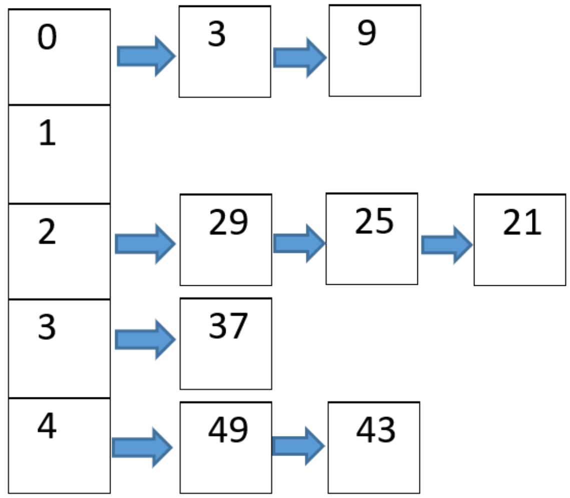

Fig. 20: Example of Bucket data structure

As shown in Fig 18, we have an unsorted array of 8 elements. We created 5 buckets for storing these 8 elements. The first bucket stores the values between 0 and 10, the second bucket stores values between 11 and 20 and so on. The first element of the array, 29, is added to the third bucket as it falls between the range 21-30, as shown in Fig 20. Likewise, the second element of the bucket, 25, is added to the third bucket which holds the range 21-30. Since the insertion of the node into the open list is straight forward(no swapping is required) by buckets, it only takes O(1) operations for inserting a new node into the bucket.

The nodes can also be stored separately in buckets with one node in each bucket. But storing the nodes this way would require a larger number of buckets when dealing with large sized maps. Instead, we can set a range of values based on the f values of the nodes to store the nodes falling under the range. Either way, the insertion in the bucket takes less operations compared to the Min-Heap.

Deletion:

When deleting the node from the bucket, the node with the lowest f value is selected and removed from the open list and added to the closed list. When we insert the node into the open list, it takes O(1) operations as we do not have to perform the bubble-up process like in Min-Heap for sorting the list. Whereas, when removing the node from the list, we need to scan through the array to find the node with the lowest f value if the bucket has more than one value stored in it. This operation takes O(k), where k is the size of the bucket. This k depends on the number of nodes added to the bucket and the number of buckets used.

However, the operations required for deletion of the node from the open list in buckets is still cheaper than the Min-Heap, where it requires O(log n). For example, in Fig 20, the nodes in the buckets are not sorted. So when deleting the least cost node from the bucket, we need to scan the array to find the node as shown in Fig 20. In the third bucket, there are 3 nodes: 29, 25 and 21. Among the 3 nodes, 21 will be removed as it is the least cost. For removing 21 from the list, it will take O(3) as the bucket size is 3.

Array Index	0	1	2	3	4	5	6	7
Input	3	9	21	25	29	37	43	49

Fig. 21: Example of Bucket data structure

Contains:

For the contains operation, before adding the node to the bucket, we need to check if the node is already existing in the open list or the closed list. For this, we need to search the list of all the buckets by comparing the name of the node with the existing nodes. When the node is located, we return the index of the node in the list. Sometimes, when we perform the Contains operation for finding the node, it might already exist in the list but with a different value than the current value. In this case, if the value is greater than the current value, it will be ignored. If not, the update operation will be performed for updating the value. This Contains operation takes O(n) to scan through the buckets to find the node. The number of operations required for performing this Contains operation is O(n) for both the data structures, Min-Heap and Buckets.

This bucket data structure was mainly introduced for solving the problems of Dijkstra's algorithm which does not perform the contains operations as it is performed in A* algorithm.

Update:

Update operation is performed for updating the f value of the already existing node in the bucket. Initially, we need to locate the node in the buckets for which we are updating the value. The existence of the node in the bucket is first checked by performing the contains operation and getting the position of the node in the bucket. After finding the node, we compare the f values of the current node to the already existing node(duplicate). If the value of the current node is lesser than the value of the existing node, its value will get replaced.

This shows that the updating of the node in the bucket requires performing the scanning(contains) first and then replacing the values of the node in the bucket. For the scanning of nodes in the bucket, it requires O(n) operations and then replacing the value of the node in the bucket takes O(log n), since after the value of the node is updated, the nodes in the bucket have to be sorted based on the new lowest value.

3.4 Summary

Based on our analysis, it is clear that implementing the open list for A* algorithm using the buckets produces better results comparing to the the Min-Heap data structure. Both the insertion and deletion operation takes only O(1) when performed using the buckets as it saves all the time that the Min-Heap takes in bubble-up and bubble-down process of sorting the tree every time a node is inserted or deleted from the tree. However, the contains and the update operations takes the same amount of time in both Min-Heap as well as the buckets. Since we have calculated the heuristics based on Octile distance, the agent was able to move in 8 directions, which made the algorithm perform better with finding a more optimal solution. With this, we can theoretically prove that the bucket data structure provides more efficient results compared to the Min-Heap and the unsorted array.

CHAPTER 4

Experiments and Results

4.1 Implementation Specifications

The implementation and testing of our project was done using Java and Eclipse IDE. The main goal of our thesis is to compare the different data structures for implementing the open list in the A* algorithm. We began by setting the start node and the goal node in a random map with obstacles, where the agent has to find the least cost path from the start node to the goal node. Among the several search algorithms for pathfinding, we chose to perform our research on the A* algorithm which is the most popular informed search algorithm. This algorithm maintains two lists, open list and the closed list. The operations performed in these two lists determines the performance of the algorithm.

Min-Heap and buckets were the two data structures we used for implementing the open list in the A* algorithm. For both the data structures, we used Arraylist for creating the open list. Although the bucket data structure was mainly introduced for Dijkstra's algorithm and it produced good results, not much research was done using the buckets for A* algorithm, this led us to implement A* algorithm using buckets. For producing fair results, we tested both the data structures on the same random game maps with the same number of obstacles, and we also varied the start and the goal node in the map based on the obstacles. We used the Eclipse IDE to display the map and the optimal path found through our experiment.

4.2 Experimental Setup

For our thesis, we performed our implementations on squared-grid maps as it produces better results. In our environment, the agent is allowed to move in 8 directions(horizontal, vertical and diagonal). The movement cost for horizontal and vertical movement is 10, and the diagonal movement cost is 14. The heuristics were calculated using the Octile distance.

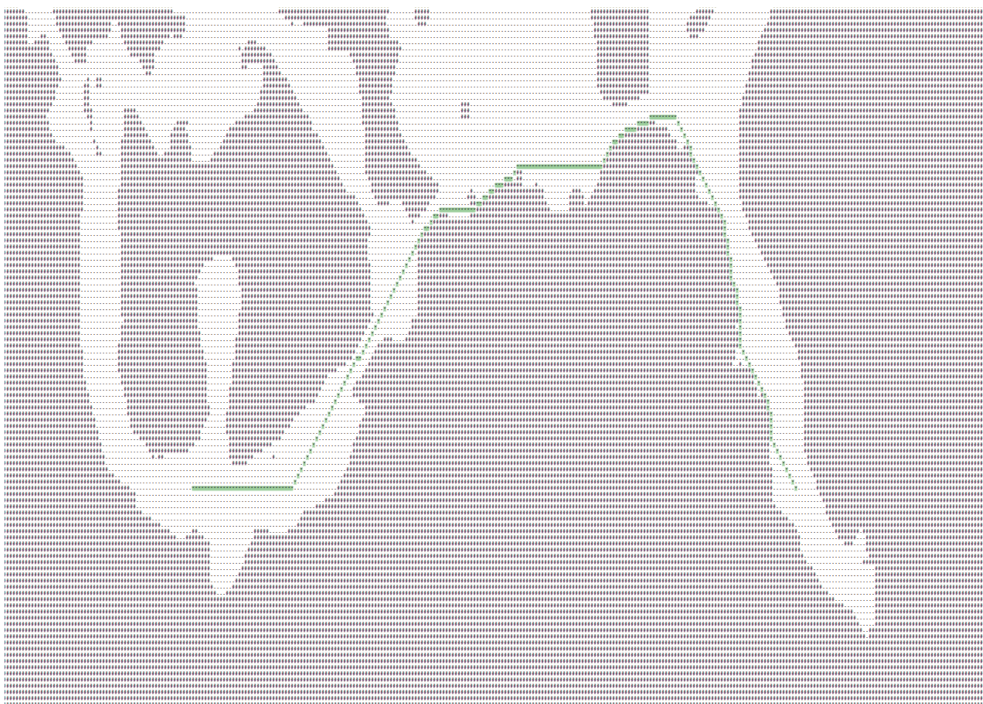

Fig. 22: Map Size 512x512

We ran our experiments on the maps of commercial games whose sizes ranged from 40x40, 80x80, 120x120 and 512x512. Some of the popular games were, Dragon Age: Origin, Baldur's Gate II and Warcraft III. The map sizes of Baldur's Gate II and Warcraft III are usually 512x512. Fig 22 is one of the results obtained by running our experiment in one of a real-time game of map size 512x512. We also set different percentages of obstacles in our maps for testing the performance of the data structures. In setting percentage of obstacles, we did not consider the distribution or clustering of obstacles. In hindsight, we think this might have been an interesting parameter to vary.

4.2.1 Search Parameters

Before running our experiments, we first need to determine the different map sizes we are going to be using to run our tests. Secondly, we need to determine the percentage of obstacles in the maps. We have used maps of sizes ranging between 40x40 and 560x560. Through our experiments, we found that when the percentage of obstacles goes above 40 percent, the chances of finding a path from the start node to the goal node reduces. Hence, we varied the obstacle percentages between 0% to 40% in our maps. In our system, we have made the start node and the goal to be manually changed and varied obstacle densities for every map. Since we are going to be comparing the performance of different data structures for implementing the open list of the A* search, there are certain factors we need to consider. Min-Heap and buckets are the two data structures we are going to be comparing and running our tests on.

The parameters we need to consider when comparing the performance of two data structures can be the different sizes of maps, the percentage of obstacles in the map, the number of operations each data structure takes for performing insert, delete, contains and update, the time taken to find the least cost path, the number of nodes expanded for finding the least cost path, the memory required for the open list and the closed list etc. Additionally, we have made it possible in our system to change the number of buckets used for storing the nodes in the open list. This is one of the important parameters to consider when calculating the operations required for insertion and deletion of nodes using bucket data structure. Based on these parameters, we ran our tests in different data structures separately.

After running the tests with one data structure, we used the same parameters for the other data structure to be getting a fair result. Based on the results, we can determine which data structure performs better under which conditions for performing the operations of the open list and the closed list of the A* search.

4.3 Performance Evaluation

For comparing the different data structures based on certain parameters, we gathered the results of the time taken to run the algorithm using different data structures, the number of nodes expanded and added to the closed list, the length of the path found from the start node to the goal node, the number of operations performed by the open list and the closed list and the memory required for storing the lists(maximum size of the open list and the closed list).

4.3.1 Time

Time is evaluated based on the duration taken to complete the process of finding the least cost path beginning from the start node to the end of reaching the final node and returning the optimal(least cost) path found. Determining the time taken to find the path shows the efficiency of the search algorithm. As we have implemented our algorithm in Java, the execution time may have some variations each time we run the program.

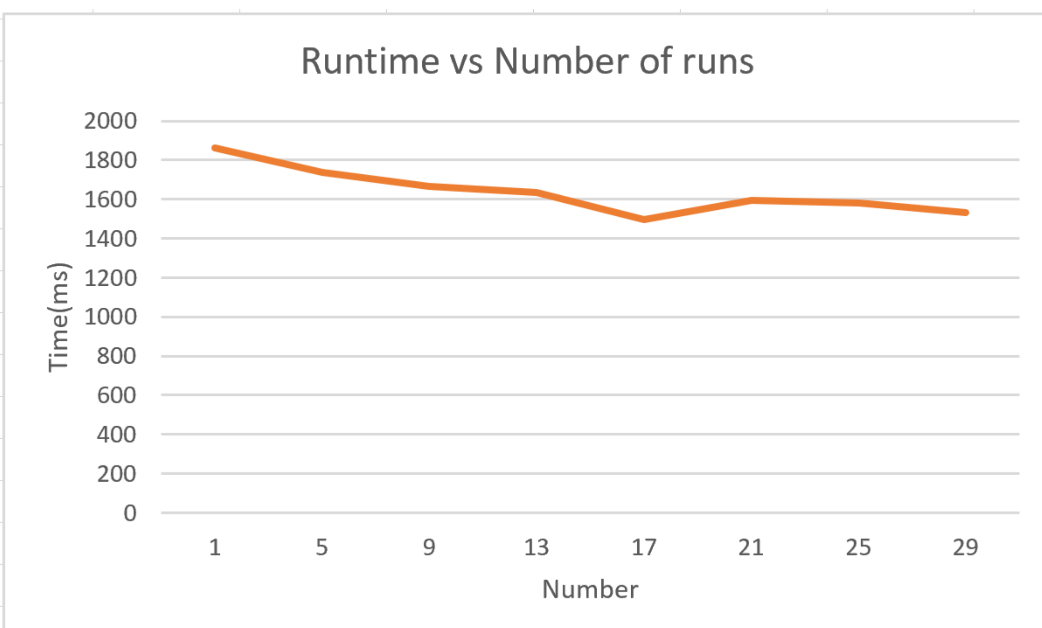

Fig. 23: Runtime for 512x512 size map

The java performance may get affected by the applications running in the background, as well as by the VM(Virtual Machine) driven by the JIT(Just In Time) optimizer, garbage collection, thread scheduler, etc[12]. To get a precise execution time we have to shut down all the other processes running in the background and run our program a couple of times. We ran our program with the same parameters for about 30 times to get an average execution time. As shown in Fig 23, the execution time reduced from 1800ms and maintained between 1500ms and 1600ms after running the program for more than 10 times. We tested this with the same parameters for different data structures, and for both the data structures the execution time was higher for the first 5 times we ran the program and then maintained the average runtime for the remaining times.

4.3.2 Number of Nodes Expanded

As we have implemented the A* algorithm using different data structures, we have considered the different parameters to look at when determining the performance of the data structures. Keeping track of the number of nodes expanded is one of the factors to be considered, as the A* algorithm will always choose the best node with the lowest f value in the open list. Based on the efficiency of the data structures, the number of nodes expanded may vary and it may affect the number of operations it takes. However, since we use the A* algorithm, we do not require to check the number of nodes in the final path, as the number of nodes in the final path will always be the same. We have used the same parameters for both the data structures and kept track of the number of nodes in the open list and the closed list.

4.3.3 Path Length

The path length refers to the number of nodes we have expanded along the final path we found from the start node to the goal node. This path does not change for different data structures we use for implementing the open list in the A* algorithm as along as we perform it on the same map, as the heuristics we use satisfies the properties of consistency and admissibility. Although this parameter seem to be not quite required to evaluate the data structures, it will still be useful for verifying the results produced. As the nodes in the final path will be the same for both the data structures.

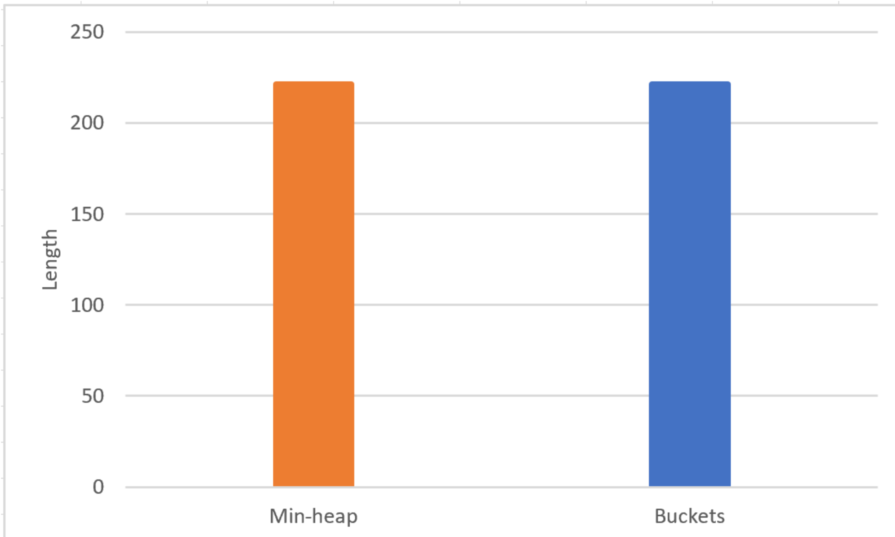

Fig. 24: Path length of the data structures

Since the path does not change for different data structures, we can verify after running the program if both the data structures have produced the same path. This way the path length can be considered as a necessary parameter for performance evaluation.

4.3.4 Number of Operations

For evaluating the performance of the A* algorithm using different data structures, we keep track of different parameters like the time, number of nodes expanded, memory required and the path length. However, we also need to keep track of the number of times the operations(insert, delete, contains and update) of the open list are called during the process of finding the least cost path. Insert, delete, contains and update are the operations carried out by the open list. Each operation varies in the number of times it is being called based on the f values of the nodes.

In the Min-Heap data structure, the insert operation of the open list takes O(log n) times as the node is inserted at the last position of the bottom-most level of the tree and it has to perform bubble-up operation to satisfy the condition of the binary tree: the parent node should always be smaller than the child nodes. Similarly, the delete operation takes O(log n) for deleting the node with the lowest f value in the open list as it performs the bubble down process. As for the contains, we compare the current node with the nodes in the open list to avoid duplicate node to be inserted into the open list. This takes O(n) operations for checking the entire tree to locate the node we are searching for. The update operation is called O(n) for checking the entire tree for the node and O(log n) for updating the f value of the node if found.

For the bucket data structure, the insertion of node into the open list is quite straight-forward as it does not have to perform bubble-up as in Min-Heap. Therefore it takes O(1) for performing insertion in the buckets. However, deletion of the node from the open list takes O(k) operations, where k is the size of the bucket . The size of the buckets are not fixed as it depends on the number of buckets used and the f values of the nodes. For insertion, we insert the nodes into an unsorted arraylist, whereas, for deleting in order to remove the node with the lowest f value, we scan through the list to remove the node with the lowest f value at the head of the list since the A* algorithm performs the deletion of node from the open list by first in first out(FIFO) method. However, the number of operations required for deleting the node from the open list by buckets is still lesser than Min-Heap where it takes O(log n).

4.3.5 Number of Buckets

Another factor to be considered for the performance evaluation is the number of buckets used for performing the operations of the open list. In our system, we made it possible to manually change the number of buckets we used for storing the nodes. By running the program with different number of buckets, we found that the number of operations does not change based on the number of buckets used. When we look at the insertion of the nodes into the open list, the nodes are inserted into the bucket in unsorted manner. Even if we increase or reduce the number of buckets used, it will still take only O(1) operations for inserting the node. On the other hand, for deletion, the number of operations may vary based on the number of buckets used as the size of the bucket increases based on the total number of nodes added to the bucket.

The performance of the bucket can be measured by varying the size of the bucket(k). Based on the map sizes and the obstacle density, varying the size of the bucket might be useful in finding the optimal size of the bucket, which might improve its performance.

4.4 Results and Analysis

For measuring the performance of the data structures, we ran our tests in maps of different sizes and varied the percentage of obstacles in every map. The sizes of the maps that we generated are ranging from 40x40 to 560x560. The start node and the goal node was chosen randomly in each map and we set the start and the goal node as far as possible, and maintained the same start node and the goal node for every test we ran in different maps with different obstacle percentages to be fair. Since the distance between the start and goal node has major impact on the time taken and the number of operations. We compared the results of the two data structures, Min-heap and Bucket, with the same map specifications.

While running our tests, we noted that the time taken for finding the path varied every time we ran the test with the same specification due to the applications running on the background and the Virtual Machine operations[12]. However, when the size of the map grew bigger, the differences between the time were more clear so we took the average time after running the tests for 5 times. For our experiments, we varied the percentage of obstacles in the maps between 0 percent and 40 percent. When the obstacle percentage increased above 40 percent, the chances of finding a shortest path became harder. The results of the tests were compared based on the number of operations, runtime, memory usage and the obstacle density.

4.4.1 Runtime

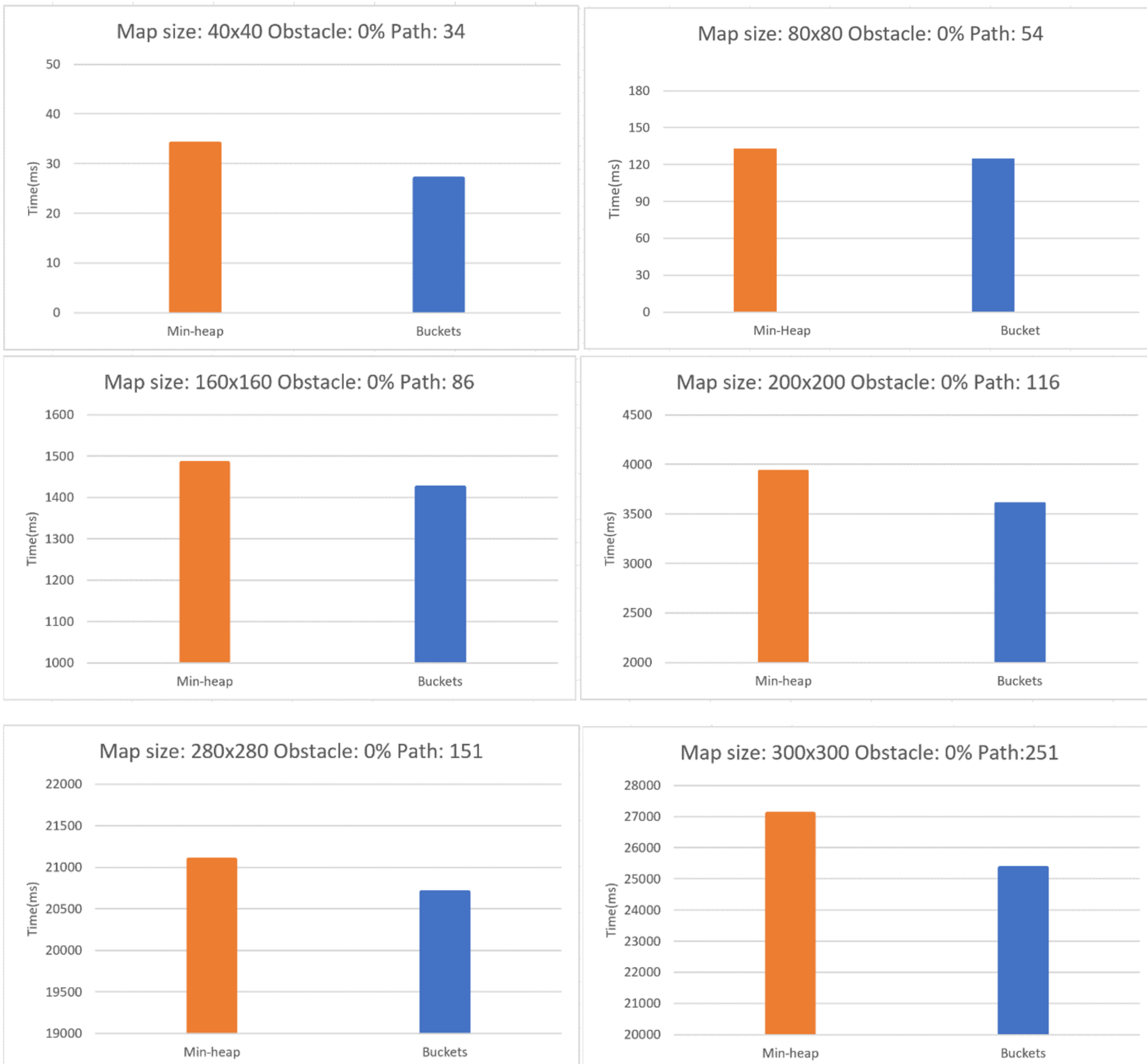

Fig. 25: Runtime of Map Sizes from 40x40 to 300x300

The performance of the data structures can be measured based on several methods such as, runtime, number of operations and the memory required for the open list and the closed list.

For calculating the runtime, we ran the tests on the maps that we generated by varying the percentage of obstacles in it. The runtime was calculated based on the duration taken to find the least cost path from the beginning of the start node to the end of reaching the final node and returning the least cost path that was found. The tests were performed on both the data structures with the same specifications(start node, goal node, obstacle percentage and map size).

Fig. 26: Runtime of Map Sizes from 360x360 to 560x560

The time taken for finding the least cost path was lesser majority of times when using Bucket data structure compared to the Min-Heap as shown in Fig 25 and Fig 26. However, the bucket data structure consumed more time than min-heap when we ran the test on a 80x80 map. This might be due to the applications running on the background which might have affected the java performance[12]. Since in every other map the bucket data structure took lesser time than the min-heap. We found that, this was the case when the size of the map was smaller and the difference of runtime between the data structures will be shorter. So we shutdown all the applications, changed the start node and the goal node with 0% obstacles, and ran our tests a couple of times again to verify if this was the reason.

As we increased the size of the map in our test environment, the time taken for finding the least cost path also increased. We added the length of the final path which was found since it was useful in determining the correctness of the results produced by the two data structures, as the A* search algorithm finds the least cost path irrespective of the data structure used for creating the open and closed lists. For example, the path length of the least cost path for the map 520x520 in Fig 25 is 381 for both the data structures.

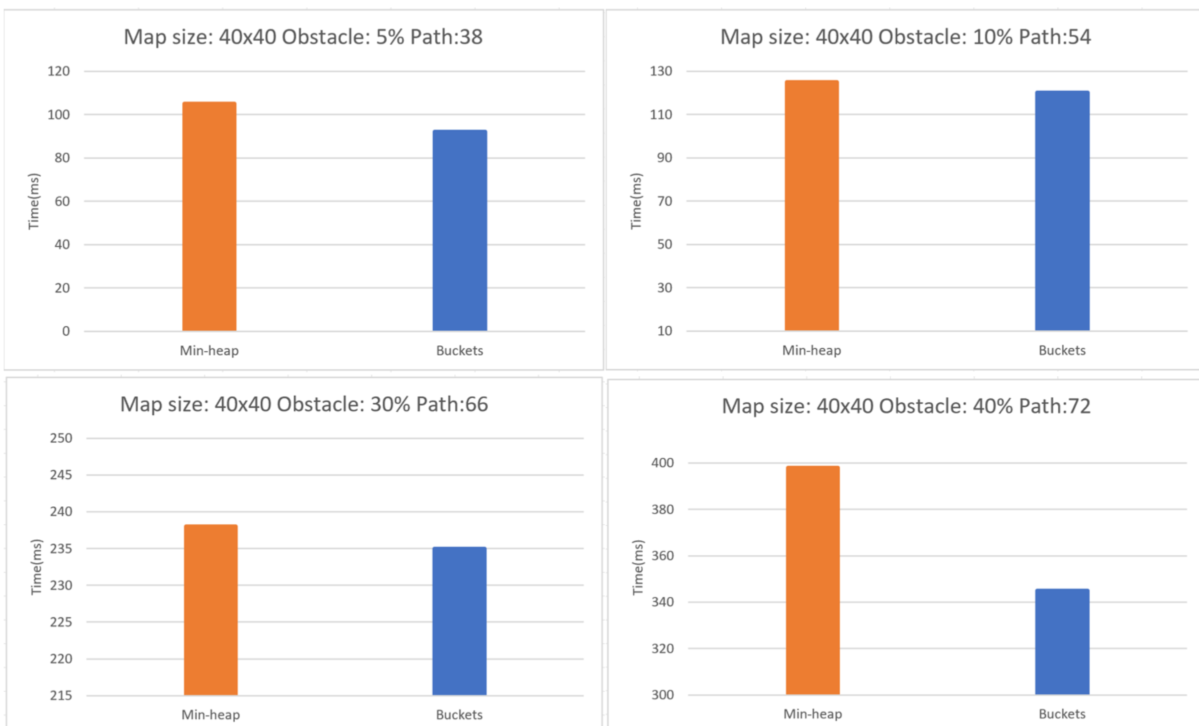

Fig. 27: Runtime of Map Size: 40x40 with Obstacles: 5% to 40%

After running the tests on the maps we generated without the obstacles, we chose maps: 40x40 and 200x200 to run tests with varying obstacle percentages of 5% to 40%. When we increased the percentage of obstacles in the map, the length of the final path also increased and it became difficult finding a least cost path when the obstacle percentage went above 50%. When we increased the percentage of the obstacle to 40%, the min-heap data structure took the maximum time for the 40x40 map. As the path length increased, the runtime also got longer.

We noticed that in almost all the cases, the bucket data structure performed better than the min-heap. It was also noted that the differences in runtime between the data structures were smaller in the 40x40 map with 5% to 30% obstacles, but it increased only when the obstacle percentage increased to 40%.

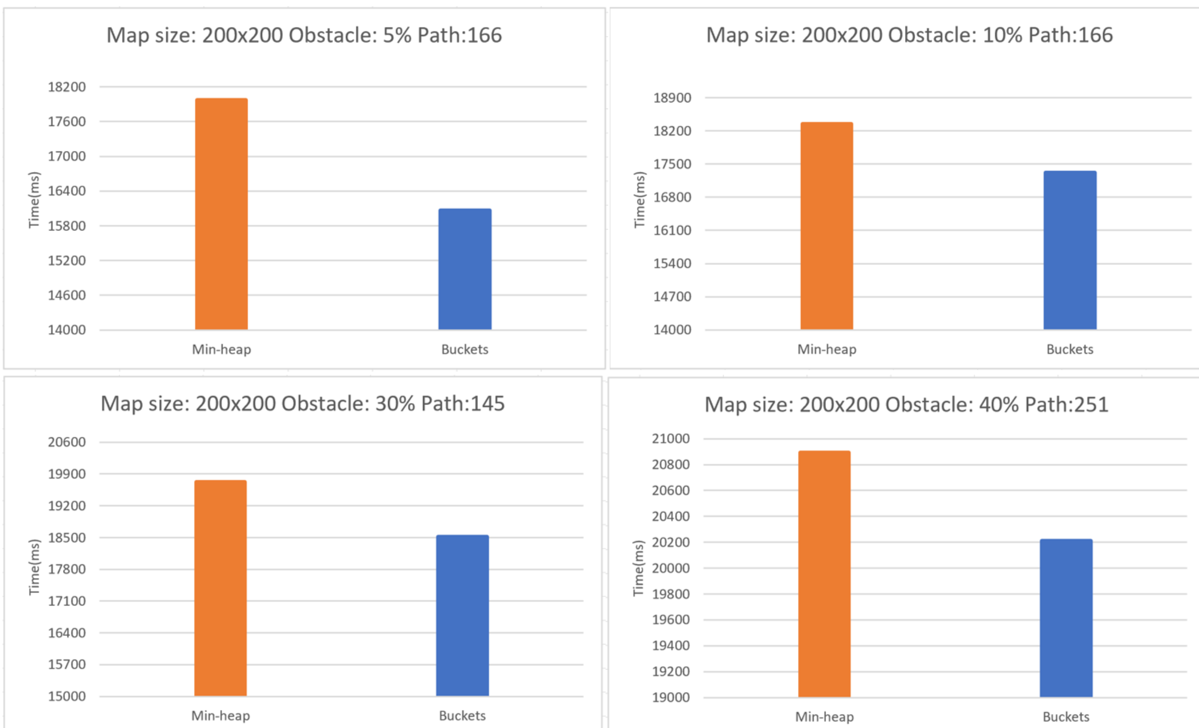

Fig. 28: Runtime of Map Size: 200x200 with Obstacles: 5% to 40%

The length of the path was the same for the 200x200 map with 5% obstacle and 10% obstacle. In the above Fig 28, the runtime went upto 20,800 when the obstacle percentage was over 40% in 200x200 map size. Since the map size was bigger and we set the start node as the goal node as far apart as possible, larger number of nodes would have been expanded to find the least cost path.

As shown in Fig 28, the number of operations taken to find the least cost path in a 200x200 map with 0% obstacles is approximately 85000. This number was increased when the obstacle percentage was 40% and hence took longer runtime. When running the tests on larger maps, we found that the difference in the runtime between the two data structures was huge as the map size was bigger, and the bucket data structure was performin much better compared to min-heap.

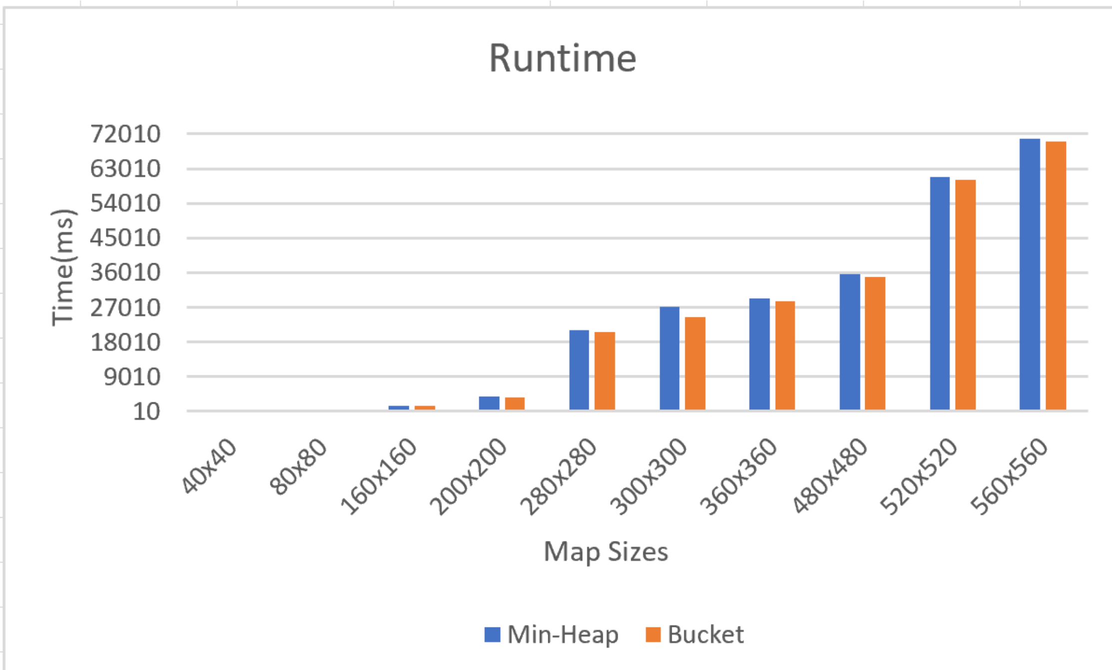

Fig. 29: Runtime of Map Sizes: 40x40 to 560x560

As we can see in the above graph, Fig 29, the runtime of the algorithm increases along with the increase in the map size. This is due to the greater distance between the start node and the goal node, so the agent has to travel through larger number of nodes to find the optimal(least cost) path in the map. It is clear from the graph that the difference in the runtime between the two data structures were between 1000 and 2000. After running the tests on maps with 0% obstacle, we increased the obstacle density for 40x40 and 200x200 size maps and compared the time taken by the Min-Heap and Buckets.

Similar tests were run for 512x512 map with obstacle percentage ranging between 5% to 40%. The time taken for finding the least cost path was varying based on the map structure and the obstacle density. When we look at Fig 29, the runtime for map with 15% and 20% obstacle density was higher than when there was 25% obstacle density, this is due to the selection of the start node and the goal node, and the obstacles in between them.

Obstacle Percentage	Heap Runtime(ms)	Bucket Runtime(ms)	Path Length
5%	105	92	38
10%	125	120	54
30%	238	235	66
40%	398	345	72

Table 2: Runtime Comparison of 40x40 Map With Different Obstacle Density

When the obstacle density was 15%, there might have been much obstacles in the path towards the goal node but not the surrounding places, so the algorithm might have to look for traversal nodes which might have taken longer time for calculating and finding the least cost path.

Obstacle Percentage	Min-Heap Runtime(ms)	Bucket Runtime(ms)	Path Length
5%	17970	16063	166
10%	18348	17320	166
30%	19728	18521	145
40%	20891	20211	251

Table 3: Runtime Comparison of 200x200 Map With Different Obstacle Density

In conclusion of the runtime parameter for measuring the performance of the data structures, we can say that the time taken for finding the optimal(least cost) path was much higher when the obstacle density was increased to 40% and the path length was also higher compared to the other maps. However, in every test, it was clear that the Min-Heap was taking more time in finding the least cost path compared to the Buckets. This might be the result of the number of operations being performed by the Min-Heap, which are lesser when using Buckets.

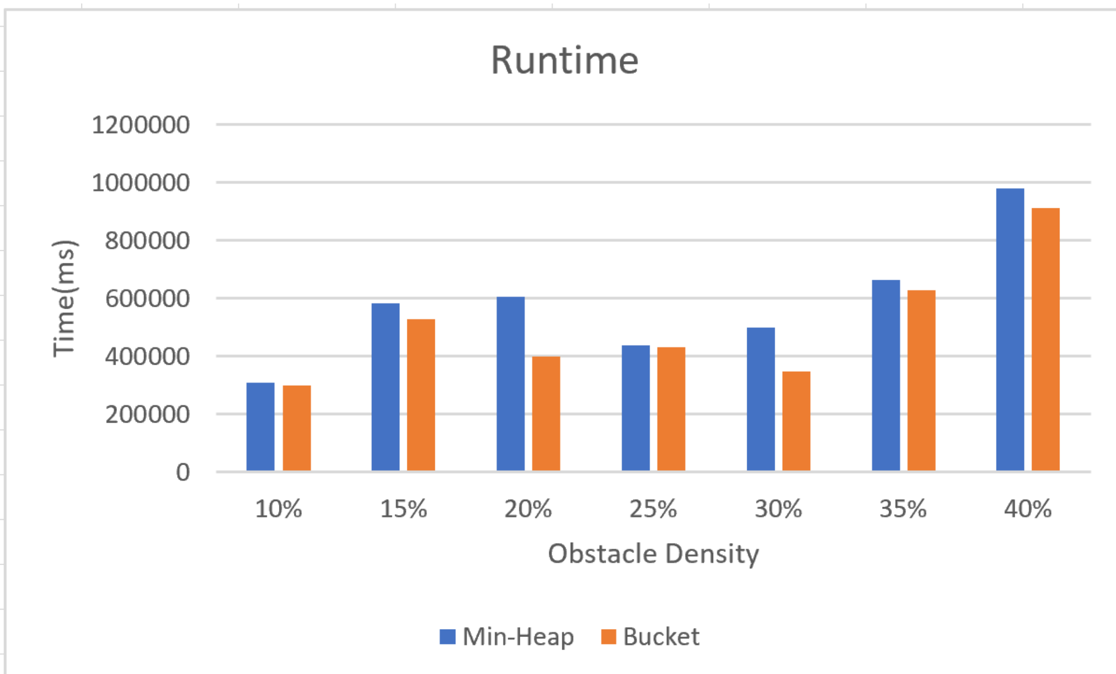

Fig. 30: Runtime of Map Size: 512x512 with different Obstacle Density

Table 2 and Table 3 shows the runtime comparison of the 40x40 and 200x200 maps with different obstacle density. The path length count was helpful in verifying the correctness of the data structures. Table 4 shows the result of running the 512x512 map with varying obstacle density between 5% to 40%. Even though in certain maps the runtime difference between the two data structures were not huge, Buckets still took lesser time compared to Min-Heap.

Obstacle Percentage	Min-Heap Runtime(ms)	Bucket Runtime(ms)	Path Length
10%	309566	297595	234
15%	583756	529474	243
20%	606026	400064	256
25%	439189	431343	238
30%	499060	347583	240
35%	662399	628840	260
40%	980071	912960	300

Table 4: Runtime Comparison of 512x512 Map With Different Obstacle Density

4.4.2 Number of Operations

Since calculating the runtime alone was not sufficient to come to a conclusion of which data structure was performing better, it is also required to measure the performance of the data structures based on the various factors which could affect the execution of the runtime[12]. We calculated the number of operations taken by the data structures for performing the operations of the open list. We counted the number of operations of the open list: insert, delete, contains and update, and totalled it to get the total number of operations performed by each data structure.

For the Min-Heap, each time a node is inserted into the open list, it has to go through up-heap process, and for removing the node from the open list, it has to go through down-heap process, where the value of the parent node is constantly compared with the value of the child node to satisfy the conditions of the Min-Heap data structure as discussed in Chapter 3. Due to this, the number of operations will be more in the Min-Heap.

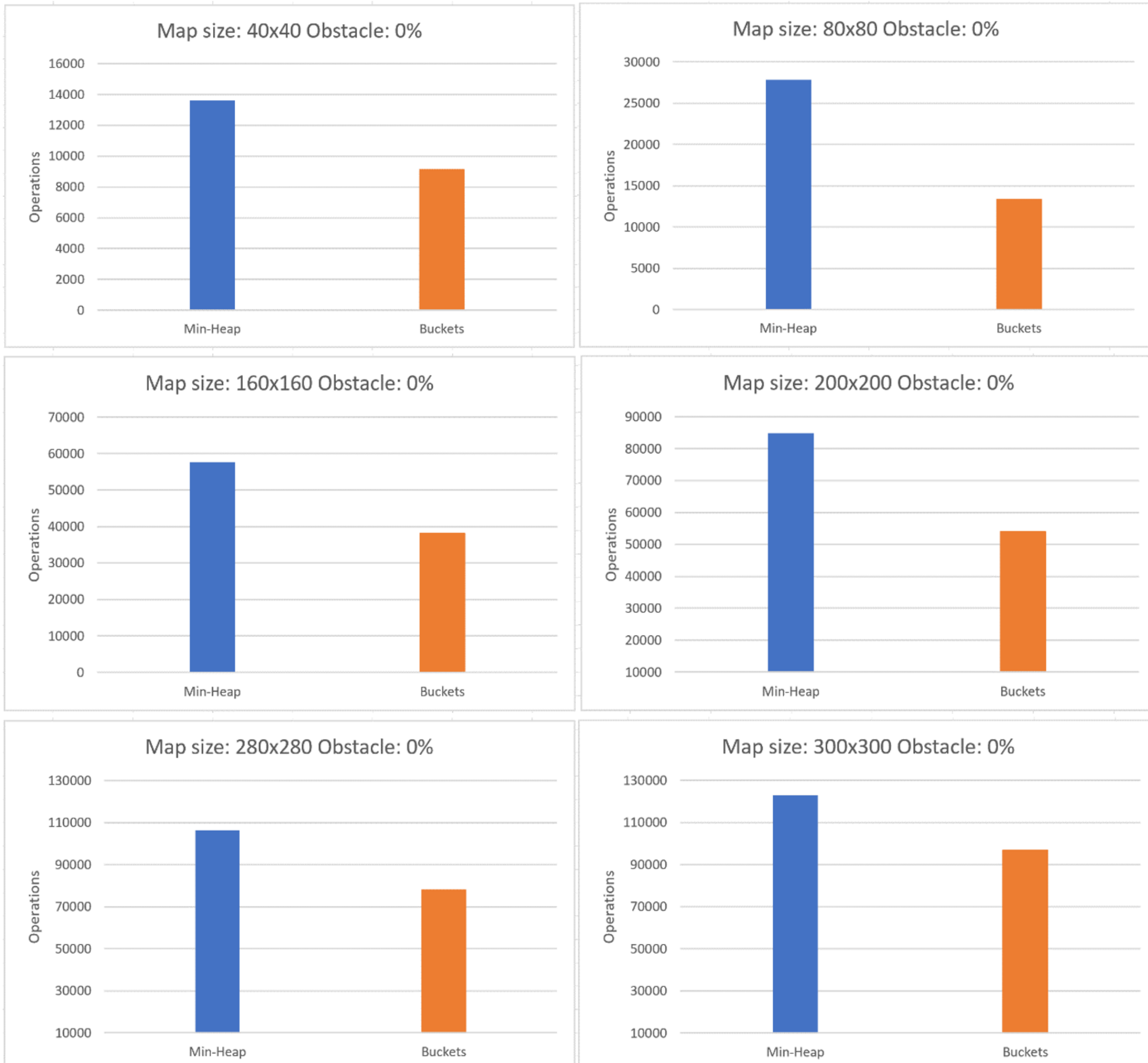

Fig. 31: Number of Operations of Map Sizes: 40x40 to 300x300

Each swap operation performed in the up-heap process is counted as one operation. Same as the up-heap, each swap operation performed in the down-heap is considered as one operation. As for the contains, every time a node is compared with other nodes in the open list to check for the existence of the current node is counted as one operation. Similarly, in the bucket data structure, for insertion, every time a node is inserted into the open list is considered to be one operation and every time a node is deleted from a bucket is counted as one operation. However, in Min-Heap, every time a node is inserted into the list, the value of the node is compared to its parent node to check if the parent node is smaller than the child node. This process involves more than one number of swapping. Therefore, for each insertion or deletion, there might be multiple number of swapping performed.

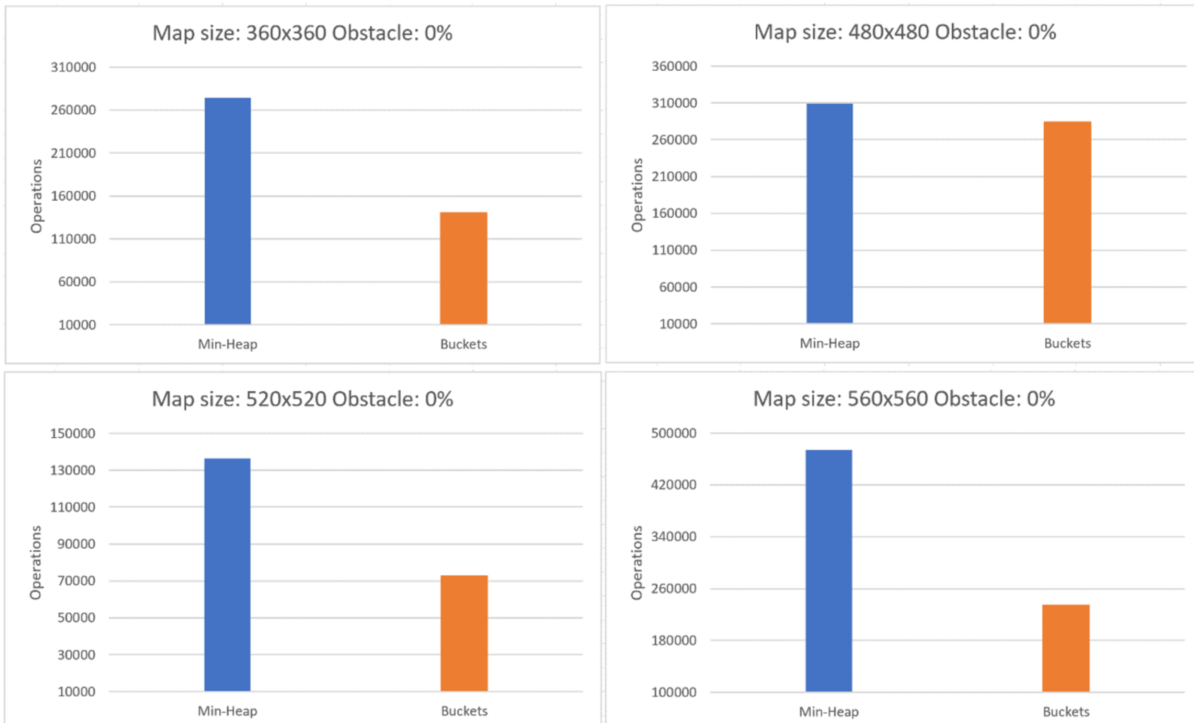

Fig. 32: Number of Operations of Map Sizes: 360x360 to 560x560

Based on the Fig 31 and Fig 32, we found that the difference between the data structures in terms of the number of operations performed is quite large. The bucket data structure has taken a considerably lesser number of operations for insert, delete, contains and update compared to the min-heap. This might be due to the lesser swapping needed in the bucket when compared to min-heap where every time a node is inserted or deleted, it has to compare its values with the parent node to satisfy the conditions of the min-heap where the parent node should always be smaller than the child nodes.

Obstacle Percentage	Min-Heap Operations	Bucket Operations	Path Length
5%	377591	205683	163
10%	376155	239907	156
30%	401502	325608	175
40%	402834	347826	178

Table 5: Number of Operations of 280x280 Map With Different Obstacle Density

When conducting these tests, we found that the number of operations for Contains was almost the same in both the data structures, buckets and min-heap. This is because, in the Contains operation, the existence of the current node in the open list or the closed is checked by comparing it to each node present in the open list and the closed list. Table 5 shows the number of operations taken by the Min-Heap and the Buckets in a 200x200 map with different obstacle density ranging between 5% to 40%.

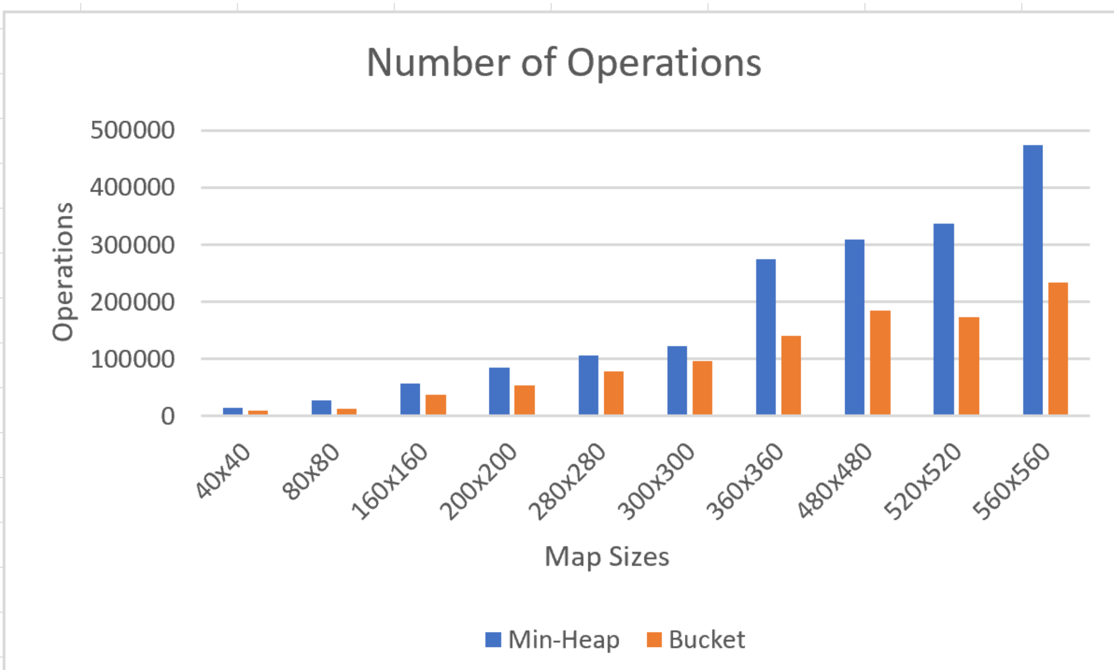

Fig. 33: Number of Operations of Map Sizes: 40x40 to 560x560

The same operations(insert, delete, contains and update) are performed using both Min-Heap as well as Buckets. Therefore, the major difference in the total number of operations taken by the data structures is due to the number of insertion and the number of deletions performed. The insertion in the buckets is straight forward, whereas in the min-heap it has to perform the swapping(upheap for insertion and down heap for deletion). However, for the deletion, the buckets are scanned in order to find the node with the lowest f cost in the open list. Therefore, the total number of operations are also higher in the min-heap compared to the buckets.

In Fig 33, it is clear that the number of operations taken by the Min-Heap grew larger as the size of the map grew larger. Whereas, with the Buckets, the number of operations did not take such drastic changes. The total number of operations(insert, delete, contains and update) taken by the two data structures are shown in Table 6 along with the length of the final path for different density of obstacles in the map.

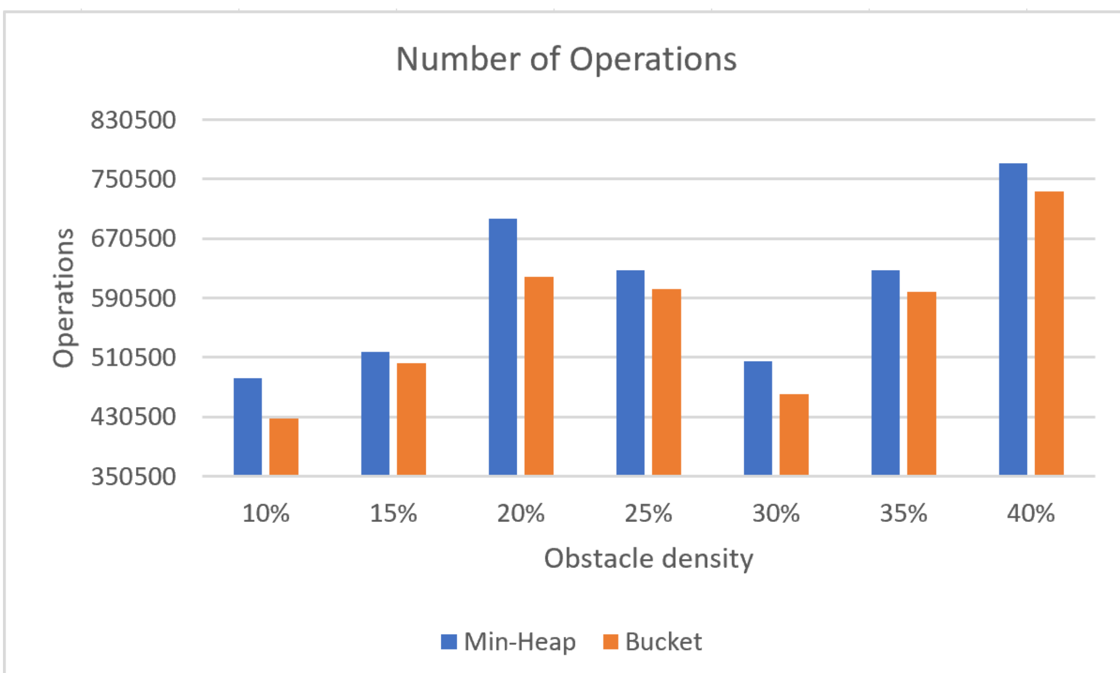

Fig. 34: Number of Operations of Map Size: 512x512 with Obstacles

After adding the obstacles to the 512x512 map, the difference in the number of operations taken between the data structures were smaller compared to the Fig 34, where when the map size was 520x520 and 560x560, the difference was huge. The detailed number of operations taken by each data structures is shown in Table 6. From the table and the graphs, we can see that the Bucket data structure was using lesser number of operations for performing the operations of the open list and the closed list compared to the Min-Heap. We verified this by running our test in different map sizes with different obstacle densities. We set the same start node and goal node for both the data structures.

Obstacle Percentage	Min-Heap Operations	Bucket Operations	Path Length
10%	482204	427879	234
15%	518670	502438	243
20%	696802	618670	256
25%	627879	602246	238
30%	505968	460863	240
35%	627879	599030	260
40%	771274	734268	300

Table 6: Number of Operations of 512x512 Map With Different Obstacle Density

4.4.3 Memory

The performance of the data structures can also be determined by the maximum size of the open list and the closed list required by the Min-Heap and the Buckets. To calculate the memory requirement, we ran our tests in different map sizes: 40x40 to 560x560 without any obstacles and with varying the percentage of obstacle density between 5% and 40%. From the tests, we found that the maximum size of the open list and the closed list were almost the same for Min-Heap and the Buckets when there were no obstacles in the map.

As shown in Fig 35, the size of the open list is the same for Min-Heap and Buckets, though it is lesser than the size of the closed list. There was a difference of one or two in the open list and the closed list between both the data structures, but it can be ignored as it was very small. This is due to the fact that both the data structures followed the Last In First Out(LIFO) rule where the redundant nodes were not traversed[4]. In majority of the cases, the size of the open list was smaller than the size of the closed list. However, there were times when the size of the open list was almost the same as the size of the closed list.

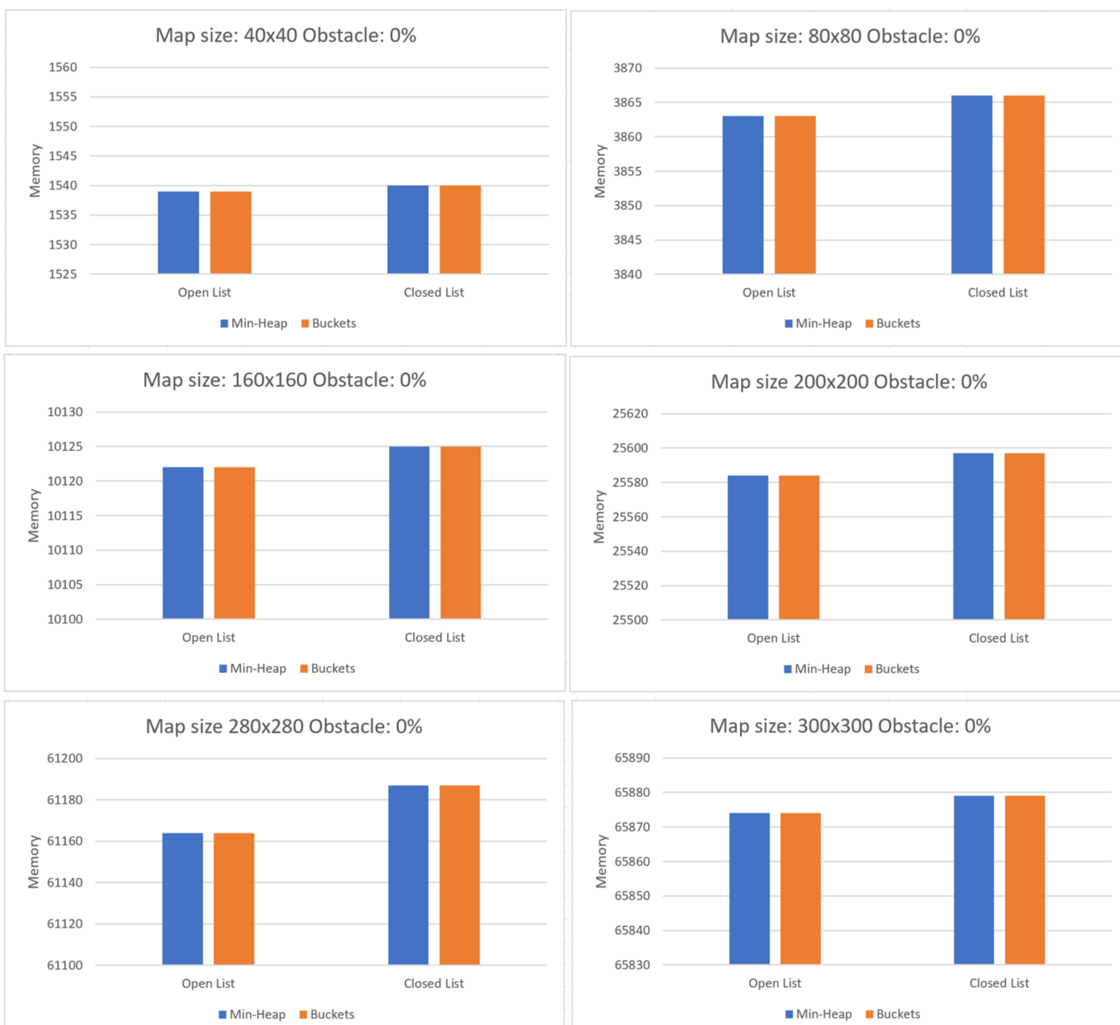

Fig. 35: Memory required for Map Size: 40x40 to 300x300

As the size of the map grew bigger, the maximum size of the open list and the closed list also got bigger as the distance between the start node and the goal node would be larger so it has to expand more nodes, which is, adding more nodes to the open list and the closed list.We verified the results by checking the length of the final path found by the Min-Heap and the Buckets.

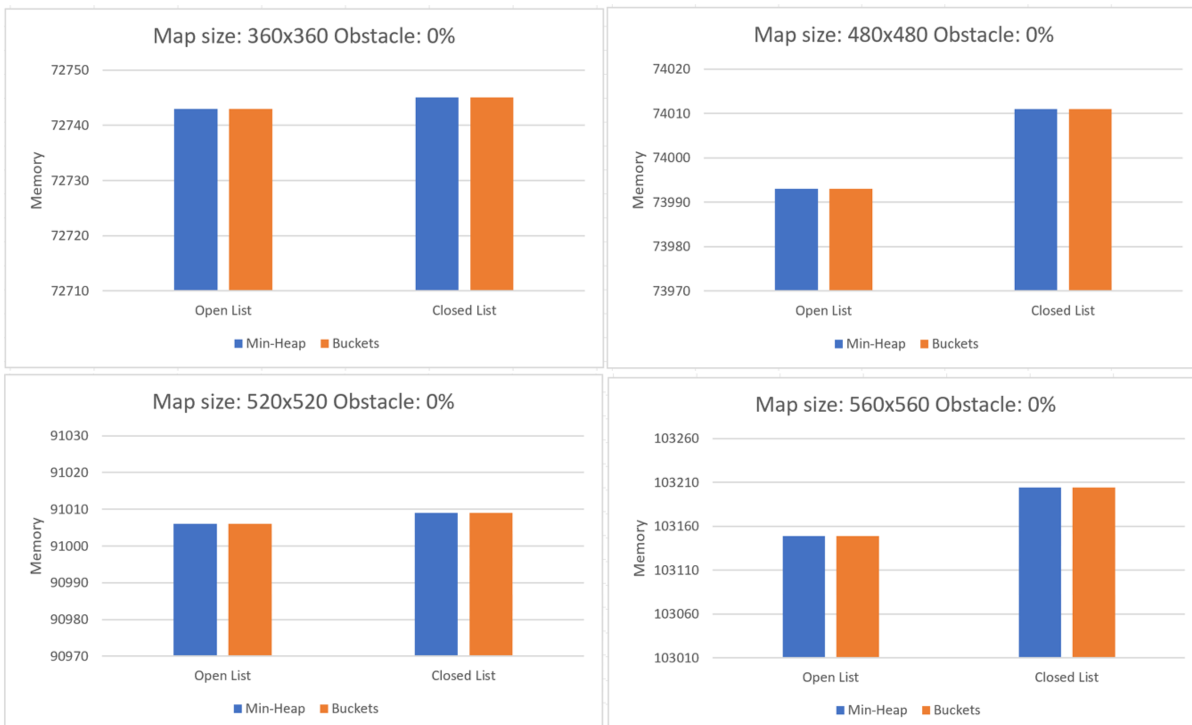

Fig. 36: Memory required for Map Sizes: 360x360 to 560x560

When we refer the Fig 35 and Fig 36, we can clearly see the difference in sizes of the open list and the closed list. Depending on the length of the path found, the difference between the lists became larger. Since we calculated the maximum size of the open list and the closed list without obstacles, there was not any difference in the lists between the two data structures, that is, the size of the list was the same in the Min-Heap and the Bucket. However, when we ran the tests by adding obstacles in the maps, we noticed that the Min-Heap data structure required more memory compared to the Buckets, which means that the Min-Heap expanded more nodes compared to the Buckets. This was already proved when we ran the tests for the number of operations performed by each data structure.

The differences in the memory requirement of the data structures were not high when we ran our tests on smaller sized maps, as the number of nodes expanded would be less. So, we ran the tests on larger maps like 480x480 and 560x560, as shown in Fig 36, in which the differences between the memory requirement of the open list and the closed list was larger. Sometimes the differences in the memory requirement for the open list and the closed list were almost the same or close enough depending on the optimal(least cost) path which was chosen.

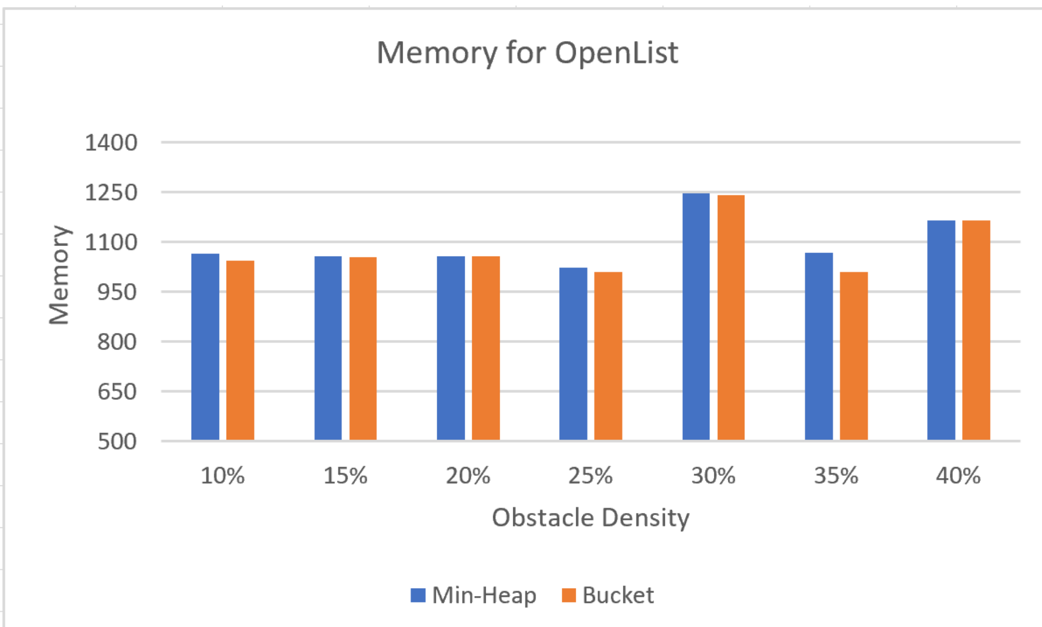

Fig. 37: Memory required for Map Size: 512x512 with different Obstacle Density

Since the difference between the memory requirements of the data structures: Min-Heap and Buckets was almost the same in the maps without obstacles, we chose 512x512 map with varying range of obstacles to check for the difference in requirement of memory. In the above graph, Fig 37, the memory requirement for both the data structures are almost the same for the open list. The difference was not huge between the data structures irrespective of the percentage of the obstacle density in each map.

However, in the closed list, when the obstacle percentage was 10%, 20% and 35%, the difference in the memory requirement between the data structures were larger.

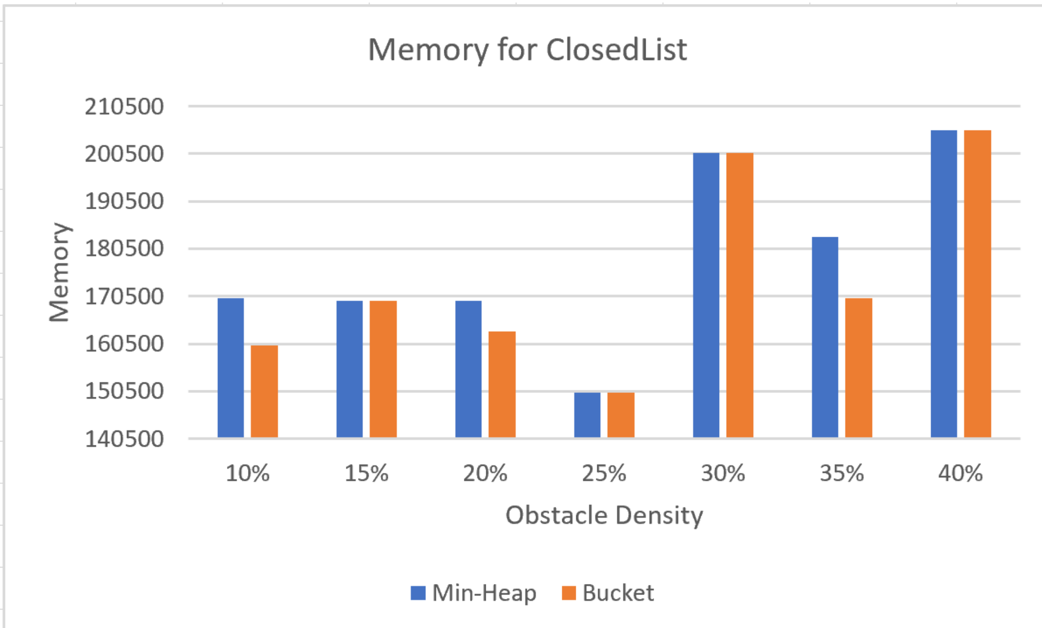

Fig. 38: Memory required for Map Size: 512x512 with different Obstacle Density

The memory requirement of the closed list was considerably lesser when the obstacle percentage was 25%. Whereas, when we increased the obstacle density to 30% and 40%, the memory requirement of the closed list became much higher. This can be due to the different paths to traverse when finding the optimal(least cost) path. That is, since the percentage of obstacles became higher, it got more difficult to find the optimal(least cost) path, so the agent has to traverse through redundant nodes to find the final path.

4.5 Summary

Based on our analysis of using different maps with different sizes and obstacle percentages, we came to the conclusion that the performance of the A* algorithm can be improved by using the Buckets for creating the open list and the closed list, compared to the other data structures such as the Min-Heap. We measured the performance of the data structures: Min-Heap and Buckets based on the runtime taken to find the optimal(lowest cost) path, the number of operations(insert, delete, contains and update) performed by each data structure and the memory requirement(maximum size of the open list and the closed list). In almost all the cases, the Bucket data structure produced better results than the Min-Heap. The differences were more prominent when we tested them on larger maps with obstacles. There were times when the Buckets performed larger number of operations but still it was still not larger than the Min-Heap.

CHAPTER 5

Conclusion

In this thesis, among the several search algorithms introduced for pathfinding, we focused on improving the performance of the A* search algorithm. While we explored the different ways for improving the performance of the A* algorithm, we found that the A* algorithm for pathfinding was affected by the data structures used for implementing the open lists. The commonly used data structure for implementing the open list is the Min-Heap. The insertion and deletion operations in Min-Heap takes O(log n) times to insert nodes into the open list and O(log n) times to delete the nodes from the open list as it has to perform bubble-up and bubble-down process for satisfying the conditions of the binary tree.

To improve the performance of the A* algorithm by reducing the number of operations performed by the open list, we proposed the use of buckets for implementing the open list. Firstly, we calculated the heuristics using Octile distance which allows the agent to move in 8 directions. This produced better heuristics compared to the Manhattan distance where the movement is restricted to 4 directions. The bucket data structure was mainly introduced for improving the performance of the Dijkstra's algorithm. Although buckets have been considered for implementing the A* algorithm, we could not find any research papers regarding it. Therefore, we implemented the A* algorithm using the bucket data structure to improve the performance of the algorithm. Buckets takes O(1) for inserting the nodes into the bucket in an unsorted array, and it takes O(n) for deleting the node with lowest f value from the bucket since it has to search the bucket by comparing the nodes values to find the node with lowest value. However, the contains operation takes O(n) for both the data structures. We considered the various factors for evaluating the performance of the data structures such as time, number of nodes expanded, number of buckets used, path length and the number of operations. Based on these factors we compared the results to find which data structure was performing better with A*.

In our experiments, we differed the sizes of maps and the percentage of obstacles in the map and found that the possibility of finding a path from the start node to the goal node becomes harder when we increase the percentage of obstacles above 40 percent. The path found using the Min-Heap and buckets were also the same under the same parameters used, but the number of nodes expanded by the buckets was one lesser than the Min-Heap. We ran our tests a number of times to find the average execution time of finding the path from the beginning of the search till finding the final node and returning the path.

In conclusion, our approach of using buckets for implementing the open list in A* algorithm performs better in terms of operations and runtime. Although the branching factor of the tree may change the overall performance of the data structures, the buckets still performs a little better compared to the Min-Heap.

CHAPTER 6

Future Work

For our thesis, we found better heuristics by allowing the agent to move in 8 directions by setting the horizontal and vetical cost as 10 and the diagonal cost as 14. Additionally, for improving the performance of the A* algorithm in terms of the operations performed in the open list, we used the bucket data structure. Although the buckets produced better results compared to the Min-Heap, the number of operations taken for deleting the node from the open list can grow significantly high for larger maps. In this case, the overall performance of the data structure may get affected and may lead to Min-Heap outperforming the buckets.

As for the future work, more research can be done in bringing up a solution to reduce the number of operations for deletion of nodes from the open list using buckets.

REFERENCES

[1] Yngvi Björnsson and Kári Halldórsson. Improved heuristics for optimal path-finding on game maps. *AIIDE*, 6:9–14, 2006.

[2] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage, and Dana Nau. Pathfinding in games. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[3] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.

[4] Qing Cao. Exploiting problem structure in pathfinding. 2018.

[5] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *IJCSNS*, 11(1):125, 2011.

[6] Xiao Cui and Hao Shi. An overview of pathfinding in navigation mesh. *IJCSNS*, 12(12):48–51, 2012.

[7] Rina Dechter and Judea Pearl. The optimality of a* revisited. In *AAAI*, pages 95–99, 1983.

[8] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a. *Journal of the ACM (JACM)*, 32(3):505–536, 1985.

[9] Samuel Erdtman and Johan Fylling. Pathfinding with hard constraints: Mobile systems and real time strategy games combined, 2008.

[10] Ariel Felner, Roni Stern, Asaph Ben-Yair, Sarit Kraus, and Nathan Netanyahu.

Pha*: finding the shortest path with a* in an unknown physical environment. *Journal of Artificial Intelligence Research*, 21:631–670, 2004.

[11] George HL Fletcher, Hardik A Sheth, and Katy Börner. Unstructured peer-to-peer networks: Topological properties and search performance. In *International Workshop on Agents and P2P Computing*, pages 14–27. Springer, 2004.

[12] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *ACM Sigplan Notices*, volume 43, pages 367–384. ACM, 2008.

[13] Andrew V Goldberg and Craig Silverstein. Implementations of dijkstras algorithm based on multi-level buckets. In *Network optimization*, pages 292–327. Springer, 1997.

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

[15] Patrik Haslum, Blai Bonet, Héctor Geffner, et al. New admissible heuristics for domain-independent planning. In *AAAI*, volume 5, pages 9–13, 2005.

[16] Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[17] Colin JH McDiarmid and Bruce A. Reed. Building heaps fast. *Journal of algorithms*, 10(3):352–365, 1989.

[18] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, 2(4):251–256, 2012.

[19] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[20] Li-Yen Shue and Reza Zamani. An admissible heuristic search algorithm. In *International Symposium on Methodologies for Intelligent Systems*, pages 69–75. Springer, 1993.

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

[22] Nathan R Sturtevant. Choosing a search space representation. *Game AI Pro: Collected Wisdom of Game AI Professionals*, 1:253–258, 2013.

[23] Anders Strand-Holm Vinther, Magnus Strand-Holm Vinther, and Peyman Afshani. Pathfinding in two-dimensional worlds. *no. June*, 2015.

[24] Ko-Hsin Cindy Wang, Adi Botea, et al. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.

[25] F Benjamin Zhan. Three fastest shortest path algorithms on real road networks: Data structures and procedures. *Journal of geographic information and decision analysis*, 1(1):69–82, 1997.

[26] Zhifu Zhang, Nathan R Sturtevant, Robert Holte, Jonathan Schaeffer, and Ariel Felner. A* search with inconsistent heuristics. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

VITA AUCTORIS

NAME:	Sheeba Mohanraj
PLACE OF BIRTH:	Chennai, India
YEAR OF BIRTH:	1994
EDUCATION:	Anna University, B.E Computer Science and Engineering, Chennai, India, 2015
	University of Windsor, M.Sc in Computer Science, Windsor, Ontario, 2019