Pathfinding: A* vs Greedy Best-first search

How does theory (big O space and time complexity) for A* and Greedy Best-first search compare with actual runtime and memory use?

Subject: Computer Science
Word Count: 3997
IB Candidate Code: 000443-0175

Contents

Introduction 2

1 Theory **3**
1.1 Time and Space Complexity 4
1.2 Basics of Pathfinding 5
1.2.1 Trees and Nodes 5
1.2.2 BFS 6
1.2.3 Grids 8
1.3 A* 11
1.3.1 Heuristic 11
1.3.2 Walkthough 11
1.3.3 Visualization 12
1.3.4 Time and Space complexity 14
1.4 Greedy Best-first Search 14
1.4.1 Heuristic 14
1.4.2 Walkthrough 15
1.4.3 Visualization 15
1.4.4 Time and Space complexity 16

2 Experimentation **17**
2.1 Methodology 18
2.2 Hypothesis 19
2.2.1 Time to completion 19
2.2.2 Memory use 20
2.3 Experimentation Data 21
2.3.1 Raw Data 21
2.3.2 Solution Paths 22
2.4 Analysis 25
2.4.1 Memory Consumption 25
2.4.2 Runtime 25
2.4.3 Path length 25

Conclusion 26

References **27**

Appendices **29**
A Further reading 29
B Maps used in EE 30
C Code used in EE (A* code, can be modified for Greedy) 31
C.1 Step by Step Algorithm program code 38
D Algorithm code for speed (No vis) 44
E Omitted Sections 49
E.1 Big O application: Bubble Sort (formerly 1.1.1) 49
E.2 Big Omega and Big Theta (formerly 1.1.2) 49
E.3 Basics of Pathfinding (formerly directly under 1.2) 50

Introduction

This Extended Essay focuses on the topic of pathfinding algorithms. Pathfinding algorithms are algorithms that find a path between two points. Contrary to its initial perceived simplicity, however, pathfinding is very diverse and intricate field, containing a large variety of potential applications: use in video games to move computer controlled adversaries (such as in Pacman)[16], real world uses such as controlling drone and robot movements. Even in simply helping people get to school/work as fast as possible everyday involves pathfinding[18].

Specifically, this essay focuses on two primary algorithms: A* and Greedy Best-first Search. The overarching goal of this essay will be to answer the following question:

- *How does theory (big O space and time complexity) for A* and Greedy Best-first search compare with actual runtime and memory use?*

This EE will be divided into two sections. The first, **Theory**, will detail the theory behind both the A* and Greedy Best-first search pathfinding algorithms. This section will be followed by the section **Experimentation**, where an experiment that answers the research question will be conducted and analyzed.

All code found in this essay can be found on **Github** as well as in **Appendix C**. **Appendix A** will also include a further read section that may clarify some of the more complex topics included in this essay.

1 Theory

This section will analyze the theory behind the A* and Best-first search algorithms.

When ranking a group of algorithms at their capability of completing a specific task, usually, the differentiating factor is in the efficiency of the algorithm. The way that computer scientists measure efficiency is through analyzing the time and space complexity of an algorithm. Therefore, before delving into the A* and Best-first search, we must gain a comprehensive understanding of time and space complexity.

Furthermore, a basic understanding of pathfinding must be attained to allow for a more in-depth discussion on the more complex aspects of pathfinding. Following the section on Time and Space Complexity will be a very brief overview of pathfinding basics to prepare for the discussion on A* and Greedy-Best first search.

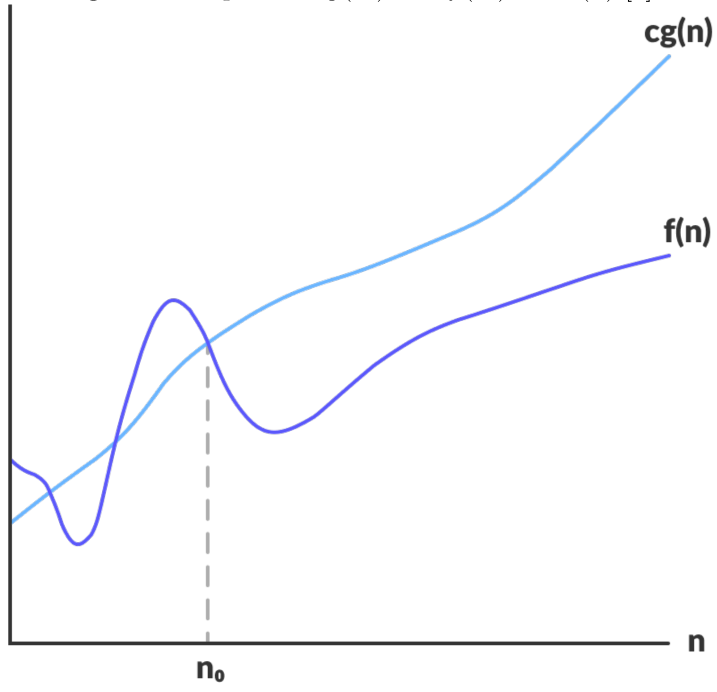
1.1 Time and Space Complexity

Time complexity provides a generalized approximation to the amount of times a statement/operation is executed in an algorithm with respect to its input size[1][10]. Space complexity describes the amount of memory an algorithm uses in respect to its input[2]. Both time and space complexity are generally represented in Big O Notation, which seeks to describe the *order*–highest power/degree–of a function (thereby describing a function's asymptotic behavior)[11]. A formal mathematical definition is:

$$f(N) = \mathcal{O}(g(N)) \Leftrightarrow \exists c, N_0 \ni f(N) \leq c * g(N) \forall N \geq N_0 : c, N_0 \in \mathbb{R}^+$$

This reads as: A function $f(N)$ is equal to $\mathcal{O}(g(N))$ if and only if there exists positive real constants c and N_0 such that f(N) is less than or equal to c times $g(N)$ for all N that is greater than or equal to N_0 [26]. Basically this is excessive jargon to simply state that $f(N)$ grows no faster than $g(N)$ as N becomes large[17], implying the function $f(N)$ can be bounded (upper bound) by a positive constant c times the function $g(N)$. The following is a visualization:

Figure 1: Graph of $c * g(N)$ and $f(N)$ for $\mathcal{O}(n)$ [3]



Let us consider an example where for every input n, $n^3 + 3x^2 + 3x + 1$ statements would be executed by an algorithm ($f(n) = n^3 + 3x^2 + 3x + 1$). Thus, we can choose the function $g(n) = n^3$ as there exists a positive constant c that we can multiply g by to get an upper bound of f (basically any $c > 1$ works–i.e. $2n^3$ grows faster than $n^3 + 3x^2 + 3x + 1$ as n becomes large)[3]. In mathematical notation that is: $f(n) = \mathcal{O}(n^3)$. Computing notation conventions differ slightly, using $T(n)$ instead

[1]It is important to note this is *not* the "actual time" it takes for the program to run, as many factors can influence this (programming language, compiler efficiency, hardware, quality of algorithm implementation)[5]. These factors will be expanded upon in **Methodology.**

[2]In many cases space complexity is insignificant and therefore unconsidered, however, for the sake of considering all factors, it will be included in this essay.

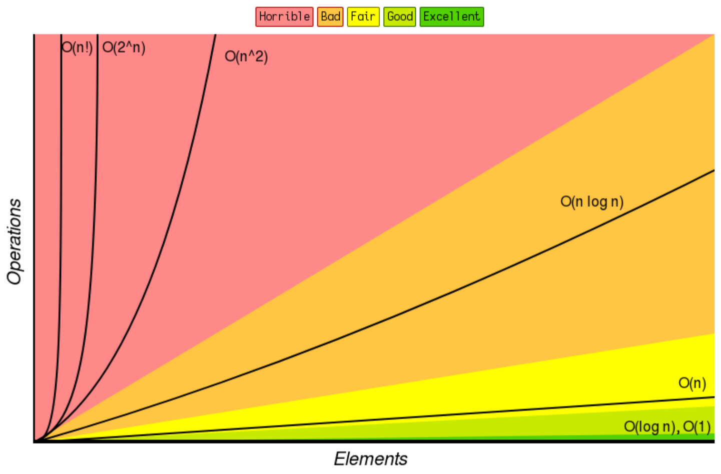
[3]We can also say that $g(n) = n^4$ or $n!^{n!}$, but these are obviously bad upper bounds, we want the upper bound to be as "tight" as possible for accuracy

of $f(n)$ as follows[4]:

$$T(n) = \mathcal{O}(n^3)$$

We say $T(n)$ grows at the order of n^3 (known as polynomial time)[4] as each input requires n^3 executions of a statement. A graph of common complexities is shown below:

Figure 2: Common Big O complexities[22]



While in some cases, we can find an exact function $T(n)$ to represent an algorithm, most of the time, the actual processing steps that are run are murkier to define. This is where it is important to remember that $O(n)$ is simply an *approximation*. Conceptually thinking about how an algorithm processes its input is a more common practice to define its big O complexity[5].

1.2 Basics of Pathfinding

1.2.1 Trees and Nodes

The "space" that pathfinding algorithms are run on is a mathematical graph (see **graph theory**), this can also sometimes be represented by a a grid or tree, with the former two (grid and graph) commonly being used for the larger and more complex visualizations and the latter (tree) for smaller and simpler ones.

First, we will briefly explore the tree structure as well as typically the simpler pathfinding algorithms (BFS and DFS) are explained using this concept. A tree is very similar to a graph. The key differences are in the fact that there exists a root "node" that all "nodes" stem from, "nodes" are arranged in a parent-child structure[6], and there are no loops[27]. Here is a figure to show the distinction:

[4]$T(n)$ is used to represent time complexity in programming, while $T_m(n)$ would be space complexity (as per MIT[4] and University of Texas notation[5]). However, notation can vary depending on the author/programmer.

[5]You will see how this works soon with BFS

[6]Meaning that one parent node "splits" into some n number of child nodes. If the number of child nodes is ≤ 2 for all nodes on the graph, then the tree is said to be a *binary tree*. The example above is a binary tree.

Figure 3: A tree [7]

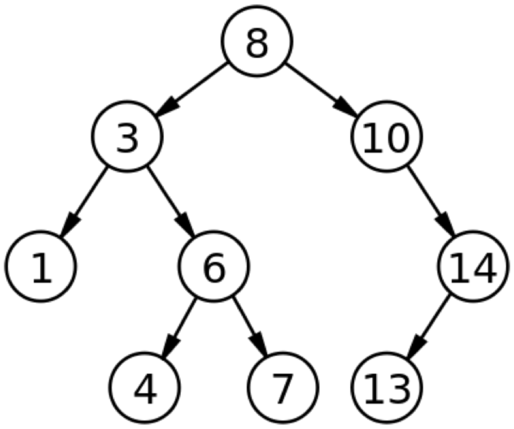
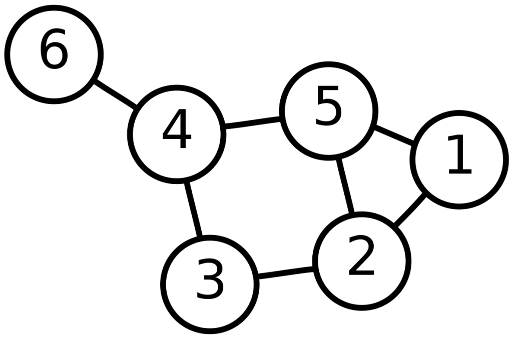


Figure 4: A graph [8]



Each circle in the tree and graph can be referred to as a *node* or *vertex.* Each node is connected to all adjacent nodes by what is called an *edge*[7]. The edges (which represent distance between nodes) can be either weighted (distances vary) or unweighted (edge distance is the same), some algorithms (such as BFS and DFS, which will be discussed in the following paragraph[8]) require unweighted nodes. Furthermore, edges can point in a single direction (as they do in the tree above) to represent that you can only travel from say some node A to some node B, but not B to A [27].

1.2.2 BFS

Understanding BFS is central to understanding the algorithms which make up the core of this essay: A* and Best-first search. Both are based on BFS. BFS is simple in principle as it follows common sense. If you were looking for something in a room, and you didn't know where it could be located, you would start with looking around you and gradually expand your scope of search. BFS follows this principle exactly. It uses a queue[9] to store nodes, and after exploring all nodes on a certain level (of a binary tree), it moves down a level, only stopping once the target node is found[9][28]. Lets take for example the following binary tree:

[7]These terms are from graph theory. Computer scientists don't use the term vertex or edge as much as "node" unless they are discussing time/space complexity.

[8]We won't discuss DFS for a few reasons: it is exactly the same as BFS except it uses a stack instead of a queue, DFS is inefficient for pathfinding as it doesn't guarantee the shortest path and its speed is luck based, furthermore, neither A* or Best-first search are based on it.

[9]FIFO(first in first out) data structure. An analogy is like a line of people trying to order something, the first person to get there is the first person to order

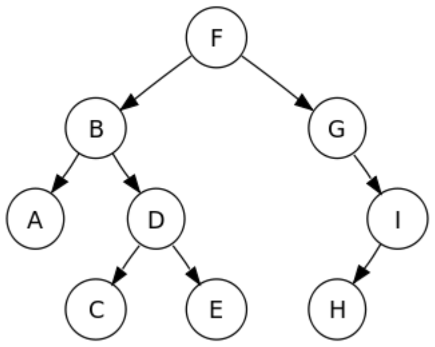


Figure 5: Example binary tree [28]

On the tree above, let us run BFS looking for the node lettered as E. Lets start with the queue q containing the root node F; $q = [F]$. We pop the queue (remove and return the first element) and check if the element is equal to the target node. $F \neq E$ so we continue by enqueuing the child nodes, meaning $q = [B, G]$. Now we repeat. Pop the queue, $B \neq E$, $q = [G, A, D]$. Pop, $G \neq E$, $q = [A, D, I]$. Pop, $A \neq E$, $q = [D, I]$. Pop, $D \neq E$, $q = [I, C, E]$. Pop, $I \neq E$, $q = [C, E, H]$. Pop, $C \neq E$, $q = [E, H]$. Pop $E = E$, and BFS is complete. A very simple implementation in Python is shown below:

```python
# This is a very simple classless implementation of a binary tree
# The tree dict is assigned in parent -> child pairs of the tree
tree = {
    'F':['B','G'],
    'B':['A','D'],
    'D':['C','E'],
    'G':['I'],
    'I':['H'],
    'A':[],
    'C':[],
    'E':[],
    'H':[]
}

# Begin with root node
queue = ['F']

# Pop queue and check if value is equal to target node
while (node := queue.pop(0)) != 'E':
    # Append children while not found
    queue.append(child for child in node)
    print("Checked and popped node letter "+node+" and appended \
    its children")

print("Found the node E")
```

Listing 1: Python implementation of BFS

The time and space complexity of BFS is $\mathcal{O}(n)$. The time complexity is $\mathcal{O}(n)$, because there are only $3n$ statements executed per n (specifically, this is popping the queue, checking node with target node, and appending children)[10]. Space complexity is $\mathcal{O}(n)$, as we assume that the tree dict is given (it is not part of the actual algorithm itself), and the only space we use is for the queue itself, which contains at worst $n - 1$ elements (in the case of a binary tree with only a root and two children)[21].

Commonly when analyzing time or space complexity of algorithms related to graph theory, instead of the variable n, the variables V for total vertices and E for total edges are used to give a clearer picture[14].

For time complexity, we realize that the while loop above runs E times (or proportional to the number of edges), as all operations within it are based on getting the child of a specific node (and the number of children of a node *is* E). The time complexity of BFS is often[11] said to be $\mathcal{O}(|V| + |E|)$. The V part comes from what is known as initialization or preprocessing. In the case of the example above, there isn't really any preprocessing, but there would be if you were to run BFS on a graph instead of a tree. In a graph, there is no parent child relationship, so you would have to make sure that you aren't travelling to nodes you have already travelled to. This means that an additional Boolean array of size V would have to be created and you would have to update this array V times (i.e. a value of 0 for unvisited nodes 1 for visited nodes). Thus, since this "preprocessing" step is $\mathcal{O}(|V|)$ and the rest is $\mathcal{O}(|E|)$ we get $\mathcal{O}(|V|) + \mathcal{O}(|E|) = \mathcal{O}(|V| + |E|)$ as the time complexity[23][24].

Space complexity is much easier to define. As stated before, the queue contains at most 1 less than the total number of nodes, thus, $\mathcal{O}$(V) is the space complexity[12][21].

1.2.3 Grids

Let us now discuss grids as they will represent the problems in this essay. Grids are essentially another way to visualize a graph. A sample grid is shown in the **figure** on the next page.

The use of grids is very common for pathfinding in many programming visualizations, especially for video games[16]. This is because in the majority of cases edges are unweighted and in a grid, much less emphasis is placed in the edges (we can, as you will see in **the next figure**, remove them altogether). Furthermore, it is also simpler with respect to programming, as we can think of the grid as simply a 2D array of nodes. Thus, we can refer to adjacent nodes with index notation. Here is example of implementing the grid above (with unweighted distances) in Python[13]:

[10]There are 3n operations as per this implementation, but the actual number doesn't really matter, as all we need to know is that the time complexity is linear (refer back to section **Time and Space Complexity.**)

[11]There is an argument to say that the time complexity is also $\mathcal{O}(|V|)$ as since BFS is a graph traversal algorithm you need to traverse through each node and thus the operation count must be proportional to node count [21]

[12]Even if you were to consider the preprocessing array which is of size V, you would simply get $\mathcal{O}(V) + \mathcal{O}(V) = \mathcal{O}(2V) = \mathcal{O}(V)$

[13]This is simply a representation of the nodes, for all the visualizations, I will be using the popular Python library **Pygame**. By allowing the Node class to inherit from the Pygame sprite class and adding a few extra lines of code, visualization is realtively easy. Once again all code can be found in the **Appendix**

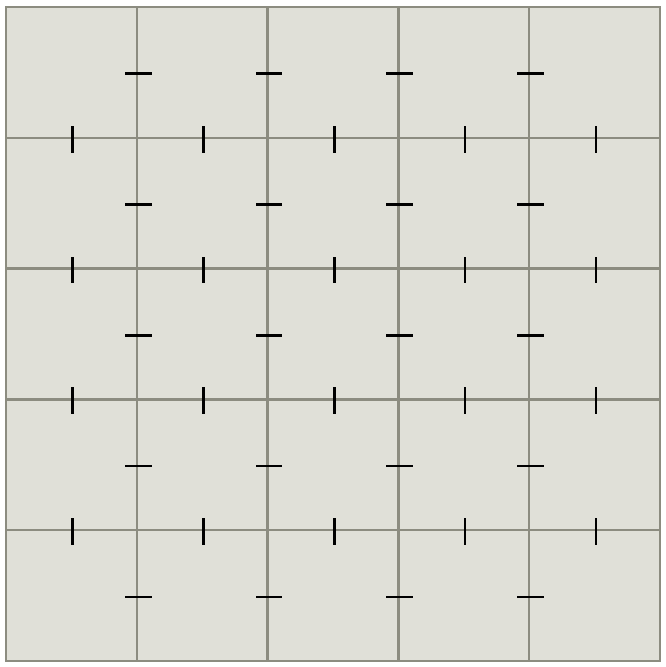


Figure 6: Sample pathfinding grid [20]

```
class Node:

    def __init__(self, pos):
        self.x, self.y = pos

    # Classmethod to help out with algorithms later
    @classmethod
    def exists(cls, pos):
        for node in cls.grid:
            if node.x == pos[0] and node.y == pos[1]:
                return True
        return False

# Grid list/array to hold all nodes
grid = []

# Create and arrange nodes as a 2D array
for i in range(4):
    row = []
    for j in range(4):
        row.append(Node((i,j)))
    grid.append(row)
```

Listing 2: Python grid of unweighted nodes example

As stated before the goal of pathfinding algorithms is to traverse through the grid of nodes to try and find a path from node A to node B[14]. "Obstacles" can be simply represented by removing nodes from the grid (del grid[y][x]), thus rendering the space previously occupied by the node as untraversable. The following figure shows a grid with starting node A and ending node B with a few removed nodes in around the center near node A:

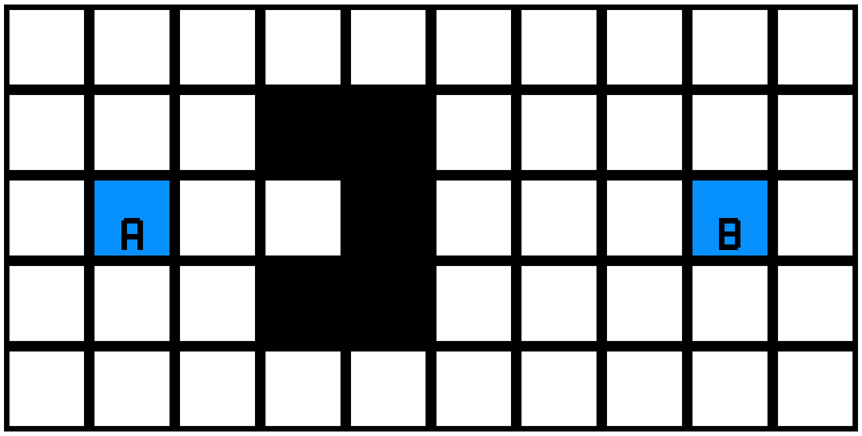


Figure 7: Example grid with obstacles and start and end nodes

An example path between the two nodes may look as follows:

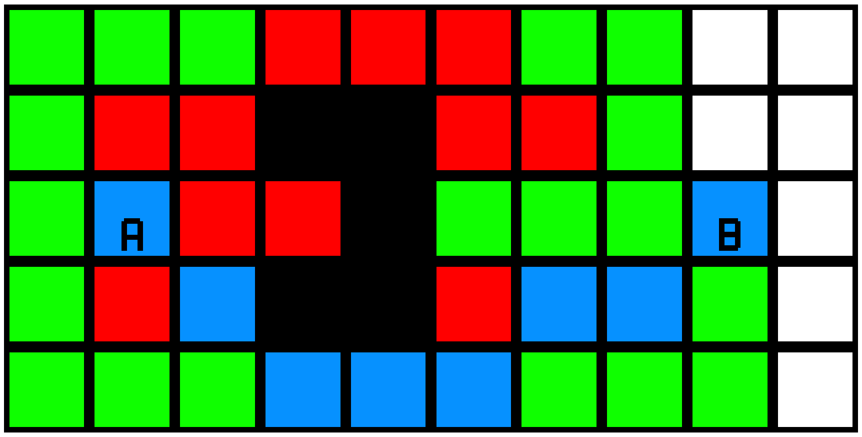


Figure 8: Path found (this is A*)

Here, nodes that are colored blue represent the path itself. Nodes that are colored red are nodes that the algorithm has selected which it believes–or had believed–to be part of the path[15]. Nodes that are colored green represent nodes that have been discovered or "seen" by the algorithm. This format will be used for the remainder of this essay[16].

[14]In this essay, the start and end nodes will be labeled as A and B respectively.

[15]while the algorithm is running, the blue nodes would been colored red as well, as at this point in time, the algorithm has not verified the correct path

[16]Colorscheme/Format based on [19] (all code is of my own creation, only the "theme" was inspired)

1.3 A*

1.3.1 Heuristic

The A* pathfinding algorithm is perhaps the best and most popular pathfinding techniques. It falls under the *informed search* category meaning that knowledge of the location of the end node is required to run the algorithm. A* can be seen as based on Dijkstra's algorithm, in the sense that it is dependent on a weighted version BFS. Furthermore, it is important to note that A* is optimal, meaning that it guarantees a shortest path is found.

The *heuristic*–essentially the "brains" of the algorithm[17]–that A* uses to evaluate what node it should travel to is as follows[20][12]:

$$f(n) = g(n) + h(n)$$

The function f represents the *cost* of a specific node n, which is equal to the sum of the functions g and h which are called the g cost and h cost respectively.

The function g represents the distance from the starting point (node A) to the current node n. Each node in A* is required to have a link to the parent node that "discovered" it, so what determines the distance function g is simply the distance from the start to the parent node plus the distance from the parent node to the current node n[18][20][12].

$$g(n) = distance(A.pos, n.pos) = n.distance = n.parent.distance + distance(n.pos, n.parent.pos)$$

The function h gives the value of the estimated distance between the node n and the ending node. In our case, $h(n)$ is simply the euclidean distance between nodes as follows (where B is, of course, the end node)[19][20]:

$$h(n) = distance(n.pos, B.pos) = \sqrt{n^2 + B^2}$$

Since A* is trying to minimize the total cost of travel, it attempts to minimize f.

$$min(f) = min(dist(A.pos, n.pos) + dist(n.pos, B.pos))$$

To summarize, A* will pick, out of a group of available nodes, a node n that–upon travelling to it–will minimize the overall distance from A to B (which is, of course, the main goal of a pathfinding algorithm)[20].

1.3.2 Walkthough

A* begins with adding all nodes surrounding A to an array, also simultaneously defining A to be the parent of these nodes (as A "discovered" them). These nodes are also all given a cost based on the function for f above[20]. A* will then choose a node n in the array that has the lowest[21] f cost, removing it from the array. Once this node is chosen, the process repeats. All nodes around the chosen node n are given a cost (if they already have one, it is updated if using n as a parent yields a

[17] This is essentially what differentiates Dijkstra's and A*, also causing A* to be categorized as an informed search

[18] The distances generated from this method may initially not represent the shortest distance from the starting node A to n, as the parent node may not be the optimal way to get to the current node n, however, over time, this cost will eventually be updated as more nodes are discovered and a better parent is found, this will be shown later on

[19] However, there are a wide variety of different functions that h be, from

[20] All nodes are assumed to initially have an infinite or undefined distance from the starting node A as the overall cost f is dependent on the g cost, which requires the parent node to be defined, meaning the current node n much first be "discovered".

[21] if there are multiple nodes with the lowest cost, it will pick the first one it encounters

lower total cost[22]). Then A* again looks at the array, choosing a new node n, discovering all nodes surrounding the new n, etc. This continues until B is finally discovered. Looking at **code**[23] and a visualization of A* will provide more clarity on this explanation.

1.3.3 Visualization

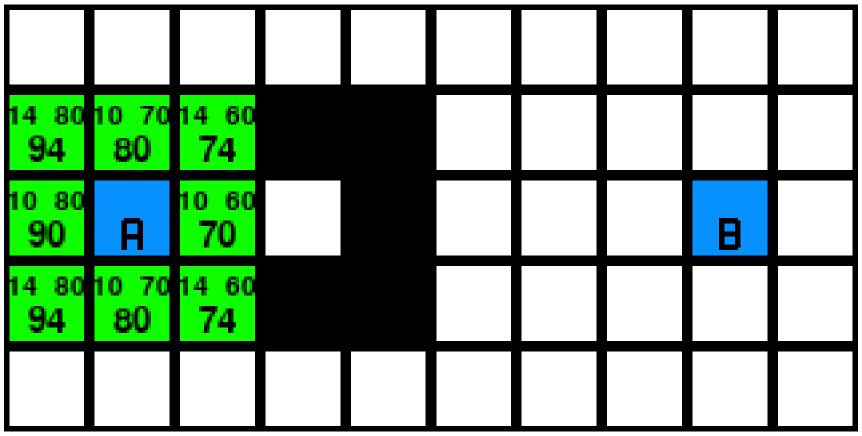


Figure 9: A* example first step

In the figure above, the g and h costs are shown in the left and right respectively, and the f cost/total cost is in the center[24]. We have just discovered and updated all nodes around the starting node, giving them their initial values. The g cost for all of these nodes is accurate as the distance is simply the distance from A (the distance from the parent node to A is 0 as the parent node *is* A).

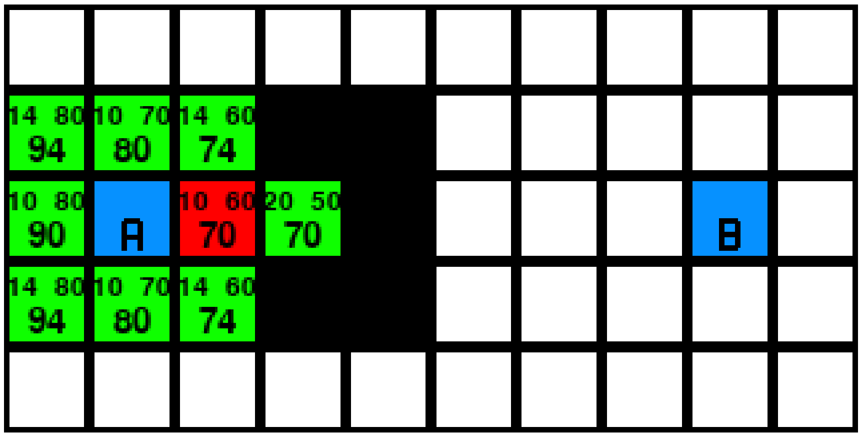


Figure 10: A* example second step

[22]Sometimes a node can have a "wrong" inital g cost. That is to say there is a more efficient way of travelling to that node. Thus, every time A* chooses a new node, it updates all nodes that surround it with a new parent node if it has discovered a faster way to get to that node

[23]Code is linked in the appendix to keep word count low, you should take a look before continuing reading to get general understanding

[24]If you haven't already assumed so, each node is of area 10 x 10, and the euclidean distance is floored (i.e. $\sqrt{200} \approx 14$)

This is what the grid looks like on the second step of A*. The algorithm picked the node with the cost of 70 as it was the node with the lowest cost out of all discovered nodes. It then discovered all nodes around that node, of which, there was only one.

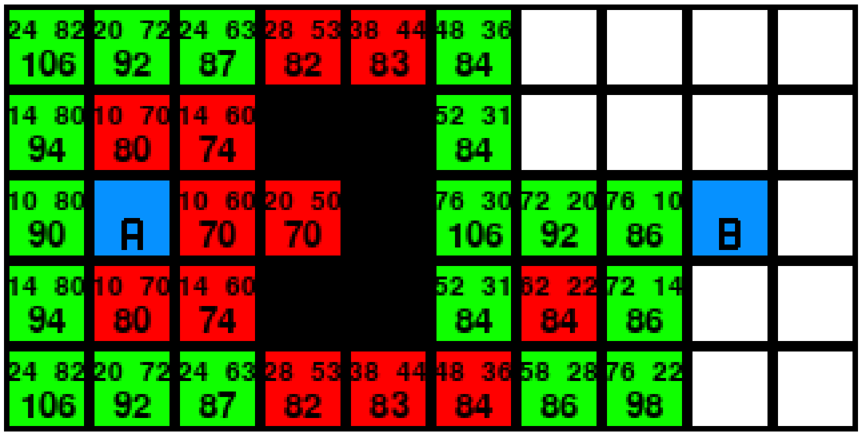


Figure 11: A* example step 11

A few steps down the line, our grid now looks like this. There are a few interesting things to watch for here. First and foremost, there are multiple nodes with a low cost of 84. Furthermore, watch what happens to the node 3 spaces left of *B*.

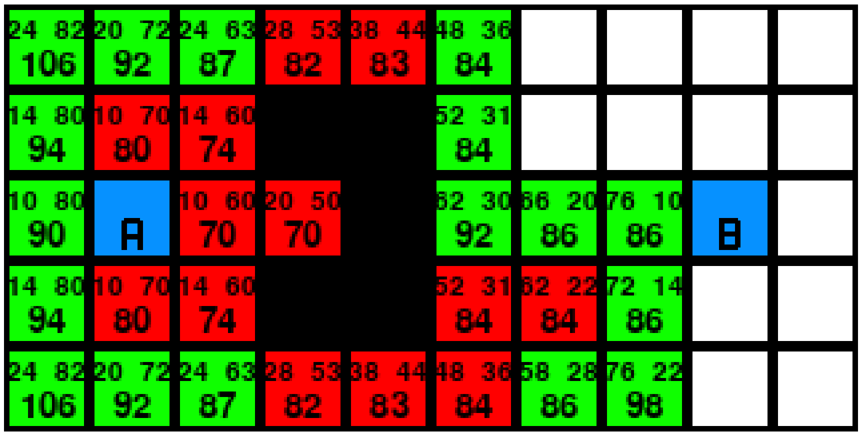


Figure 12: A* example step 12

The node 3 spaces to the left of B updated. Its original cost was 106, but it changed to 92. This is because the algorithm found a better path to that node. Instead of getting to that node travelling through the 84 on the rightmost side, it found out that it was more efficient to get to that node through the left 84. With regard as to why it picked this 84 over the rest, it turns out that it actually doesn't matter which node it picks if many nodes are the same cost. Eventually, the other nodes will be selected, or a equally short path will be found.

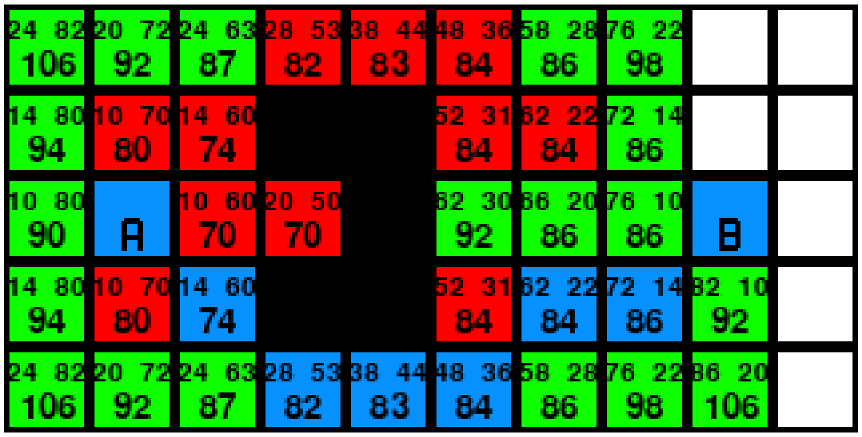


Figure 13: A* with costs shown

This is the completed algorithm that was seen in **Figure 10**.

1.3.4 Time and Space complexity

We can say that the time complexity of A* is $\mathcal{O}(|E|)$[25]. Since A* is simply a "improved" version of BFS, at worst, we can bound it by the worst case complexity of BFS. However, we realize that we don't need a step to cover for returning to a discovered node, as the heuristic continually guides the algorithm towards B. This means that we do not need the Boolean array/preprocessing step. Thus, the algorithm can be seen as $\mathcal{O}(|E|)$ as it only processes nodes that branch from the current node[26][12].

The space complexity of A* is the same as BFS, since the only storage we are using is proportional to the number of nodes V, giving $\mathcal{O}(|V|)$[12][15].

1.4 Greedy Best-first Search

1.4.1 Heuristic

Greedy Best-first search is very closely related to A*. It, like A*, falls under the *informed search* category and is also based on Dijkstra's algorithm. The heuristic for Greedy is as follows[6]:

$$f(n) = h(n)$$

This is essentially the heuristic for A* minus the g cost. So, we can think of Greedy to be attempting to do the following:

$$min(f) = min(dist(n.pos, B.pos))$$

To summarize, Greedy will pick a node n that minimizes the cost of travelling to B, disregarding how far it has travelled to get to the node n.

[25] A* doesn't have an "agreed upon" time complexity, as "it depends on the efficiency of the heuristic". This is evident if you consider the scenario when you know in advance h(n) represents the euclidean distance and is exact. This means that the distance would be a straight line, and thus, you could essentially compute the path instantaneously, suggesting a best case and worst case time complexity of $\mathcal{O}(1)$, $\theta(1)$, $\Omega(1)$

[26] Revisiting nodes does not change the time complexity as it would revisit nodes in a linear proportion to E

1.4.2 Walkthrough

The method in which Greedy works is exactly the same as A*, as thier only differences lie in the heurisitc that guides their search. So, like A*, Greedy begins with discovering all nodes around A, picking the node n with the lowest cost, discovering all nodes around n and repeat until B is found.

1.4.3 Visualization

The following is a visualization of Greedy using the same grid that the A* algorithm was run on.

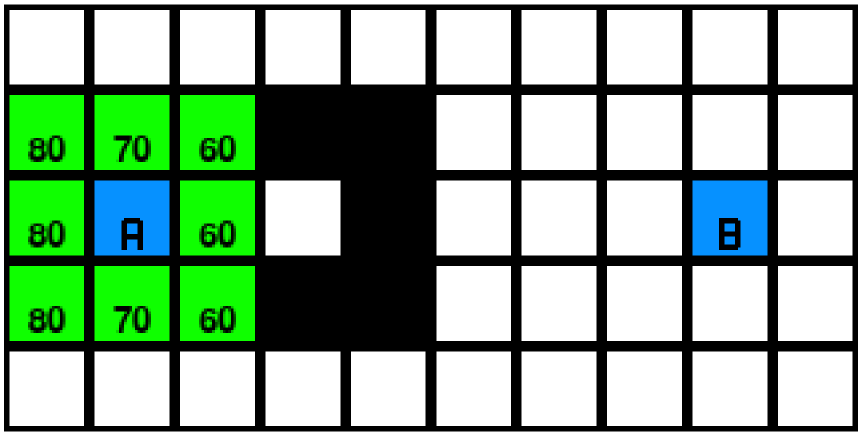


Figure 14: Step 1 of Greedy Best-first search

As we can see, the first step is the same, but the values are vastly different. Since Greedy best-first uses only the h cost, that is the only cost shown in the image above.

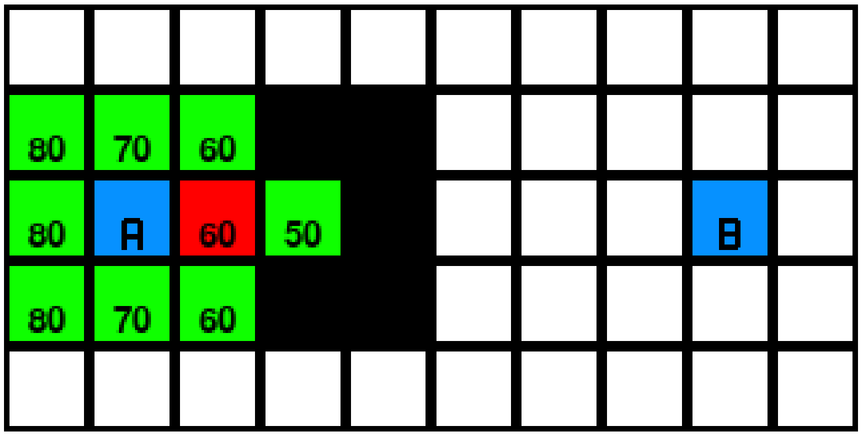


Figure 15: Step 2 of Greedy Best-first search

The second step is basically the exact same; Greedy follows essentially the same steps as A*, the only difference between the two comes in which node to select (due to the differences in heuristics).

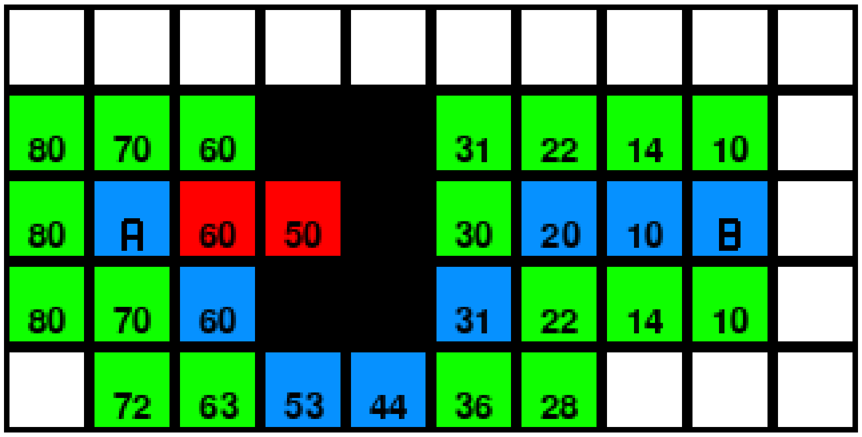


Figure 16: Greedy Best-first completed

Above you can see the completed Greedy best-first search. It found the end node before even going to 11 steps. Furthermore, the amount of area it traversed through was significantly less than A*. While Greedy doesn't always guarantee the best solution, in this case, it did in fact find it.

1.4.4 Time and Space complexity

Similar to A*, we can bound the time complexity of Greedy by BFS, and we can also remove the preprocessing step required to get a time complexity of $\mathcal{O}(|E|)$. This is because like A*, in the worst case scenario, the algorithm will traverse through the nodes that surround each node, meaning that the number of edges in the graph give an approximation of the time complexity[6].

For space complexity, it is once again the same as A*, as in the worst case scenario, all nodes have to be included in the array, meaning that the space complexity is $\mathcal{O}(|V|)$.

2 Experimentation

This section will provide an experimentation of the A* and Greedy Best-first algorithms.

The experimentation will involve a hypothesis of predicted outcomes for the A* and Greedy Best-first algorithms with respect to each specific environment they are run in, in addition to an analysis of findings. Both of these subsections will use the accumulated knowledge garnered from the preceding **Theory** section.

This is the final section of the EE, and thus following this section will be a conclusion that finalizes the answer the guiding question presented in the introductory paragraph.

2.1 Methodology

There will be 3 grids that the algorithms[27] will be run on. A maze, a relatively plain grid, and a mix of the two. Here are the grids[28]:

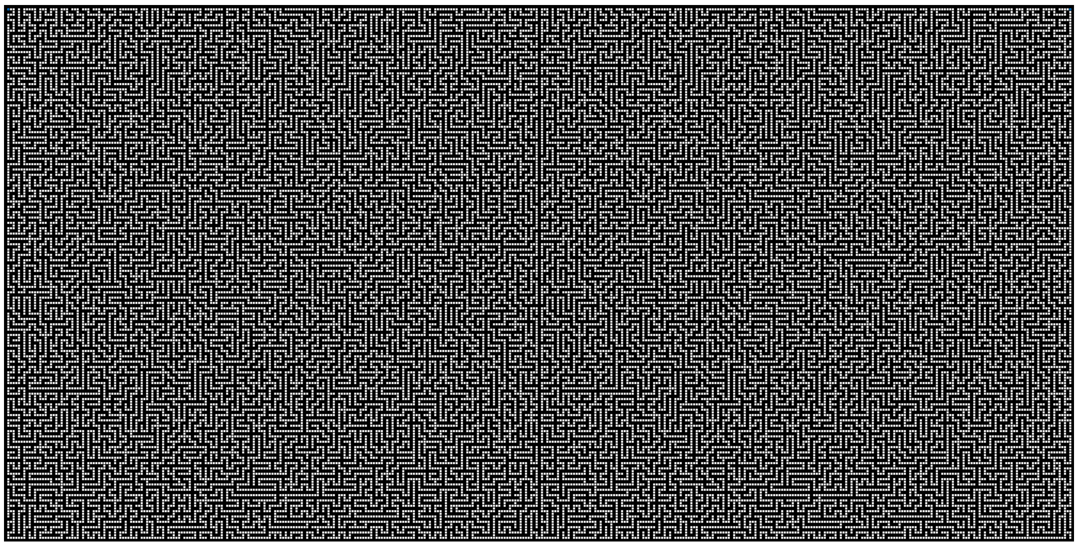


Figure 17: Grid 1: A maze(A and B are in the top right and left corners respectively)

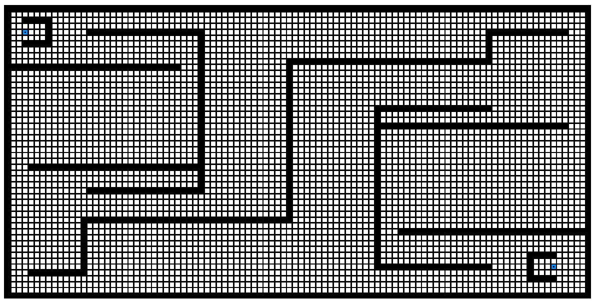


Figure 18: Grid 2: A semi-sparse grid

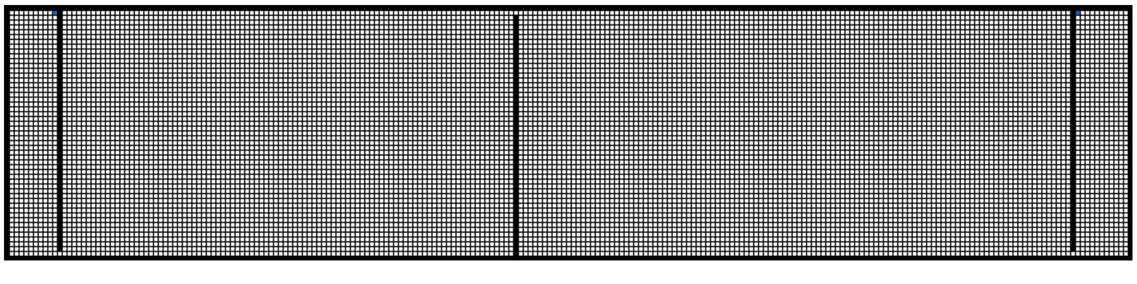


Figure 19: Grid 3: A sparse grid

[27]I will be using a modified version for these tests where moving diagonally is disallowed

[28]It is difficult to see grid 1 since the image is small. Here is a link to the full size of the image: **Google Drive Link**

I will be measuring runtime using the builtin time module.

```python
import time
start = time.now()
# program code goes here
print("Total time: "+(time.now()-start))
```

Listing 3: Python program runtime

I will be measuring memory usage using the os and psutil modules.

```python
import os, psutil
process = psutil.Process(os.getpid())
# program code goes here
print("Memory use: "+psutil.bytes2human(process.memory_info().rss))
```

Listing 4: Measuring memory use of a python program

There will be three trials for each map, and I will average the results[29].

2.2 Hypothesis

Looking at Big O space and time complexity, we would expect that Greedy best-first search and A* should perform exactly the same. However, we must keep in mind that Big O is simply an approximation, and it is most optimal for analyzing an algorithms efficiency as the size of the input becomes extremely large. The input in our case are not "extremely large"[30] but perhaps a scale of magnitude lower, and thus there should be at least a slight noticeable difference between them.

2.2.1 Time to completion

In terms of time to completion, we should expect that Greedy Best-First wins this race. This is because Greedy has less to compare; A* must compare distance to the end node *in addition* to distance from the starting node, while Greedy only has to compare end node distance. However, on the same side of the coin, while Greedy compares less, hinting at less comparisons, it also has a less efficient path, so it may have to traverse through more nodes (thus meaning it has to compute more data) in certain situations.

In the case where the map is a maze, and there isn't much room for movement, Greedy should perform much better than A* since due to the fewer options, it would be harder to pick an inefficient path. In a more sparsely populated and open map, either could be faster. This would depend on the positioning of start and end nodes in respect to the map. This becomes apparent looking at the figures[31] below which is a slight variation of the walkthrough examples:

[29] I realize that I should also factor in the inefficiency that Greedy creates, probably by creating a single number representing an overall weighted sum of runtime, memory use, and time complexity. However, this would add a lot of subjectivity as I would need to decide which matter more (I would rate it as efficiency followed by runtime followed by memory). I don't have the word space to add it either–I have already passed the EE word count and now need to cut out sections

[30] I basically used the largest I could find/create

[31] The idea for this grid was taken from [20]

Figure 20: A*

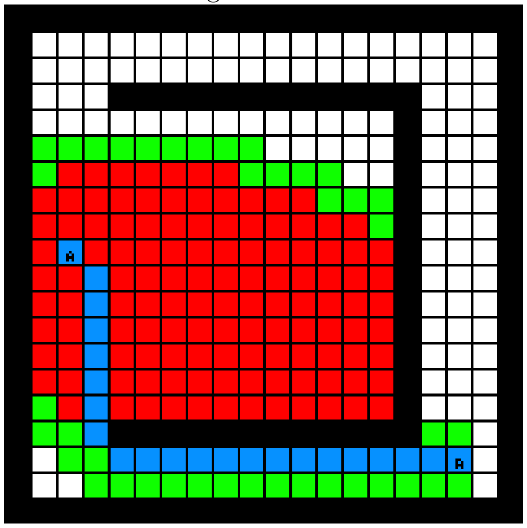
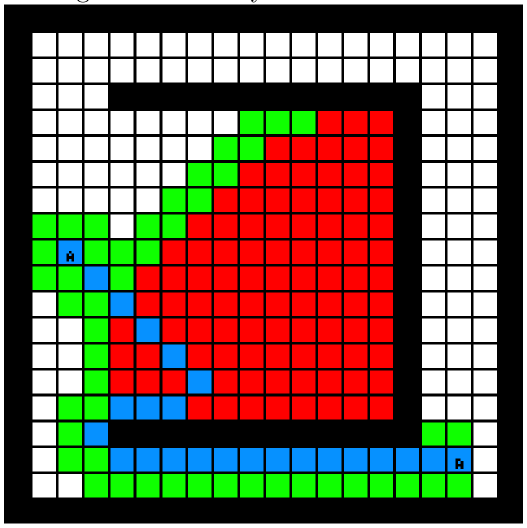


Figure 21: Greedy Best-First Search



As you can see because of the inefficiency, Greedy evaluates almost as many nodes as A*. Thus, I predict that Greedy will be faster for the first two, but much slower on the last grid.[32]

2.2.2 Memory use

Greedy should again win with respect to overall memory use. The same line of reasoning follows as thinking about runtime, Greedy should, in most cases, have less work to do, and thus, it has less it needs to store. Overall, looking at Greedy's solutions, it generally looks through less nodes than A*, and combined with the fact that it doesn't even have to store distance from A (or compute it–computation requires memory use as well), as part of its cost calculation, it should have a lower memory consumption.

However, these algorithms aren't really storing that much data overall; the algorithms only really have to store node locations and nodes data for a few hundred nodes.

I predict Greedy will have a lower memory use in all cases but by an insignificant margin.

[32]The B node seems to be a bit distorted in these images. It is the node on the bottom right, and the A node is on the left.

2.3 Experimentation Data

2.3.1 Raw Data

	A*		Greedy	
	Memory (MB)	**Runtime (sec)**	**Memory (MB)**	**Runtime (sec)**
Trial #1	113.7	1614.88	113.7	1025.40
Trial #2	115.6	1577.76	113.7	1064.91
Trial #3	115.9	1572.00	112.0	1025.47
Average	115.1	1588.21	113.3	1038.59

Table 1: Grid 1 Raw Data

	A*		Greedy	
	Memory (MB)	**Runtime (sec)**	**Memory (MB)**	**Runtime (sec)**
Trial #1	38.6	88.05	38.3	20.31
Trial #2	38.6	92.78	38.3	20.42
Trial #3	38.8	84.83	38.5	20.18
Average	38.7	88.55	38.4	20.30

Table 2: Grid 2 Raw Data

	A*		Greedy	
	Memory (MB)	**Runtime (sec)**	**Memory (MB)**	**Runtime (sec)**
Trial #1	47.0	294.61	46.1	50.49
Trial #2	46.7	279.19	46.2	42.78
Trial #3	47.1	279.90	45.6	39.29
Average	46.9	284.57	45.97	44.19

Table 3: Grid 3 Raw Data

2.3.2 Solution Paths

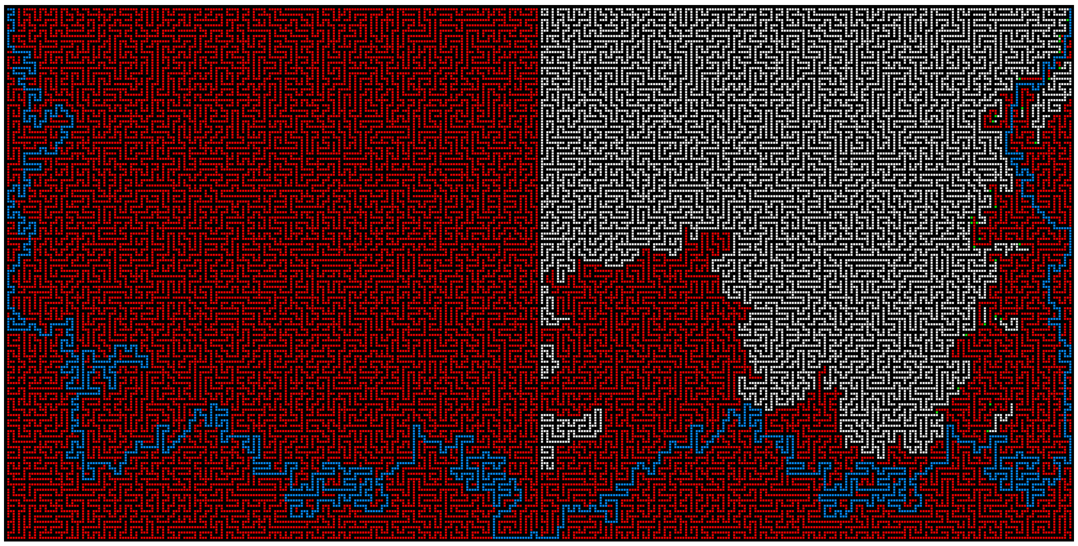


Figure 22: A* solution for grid 1

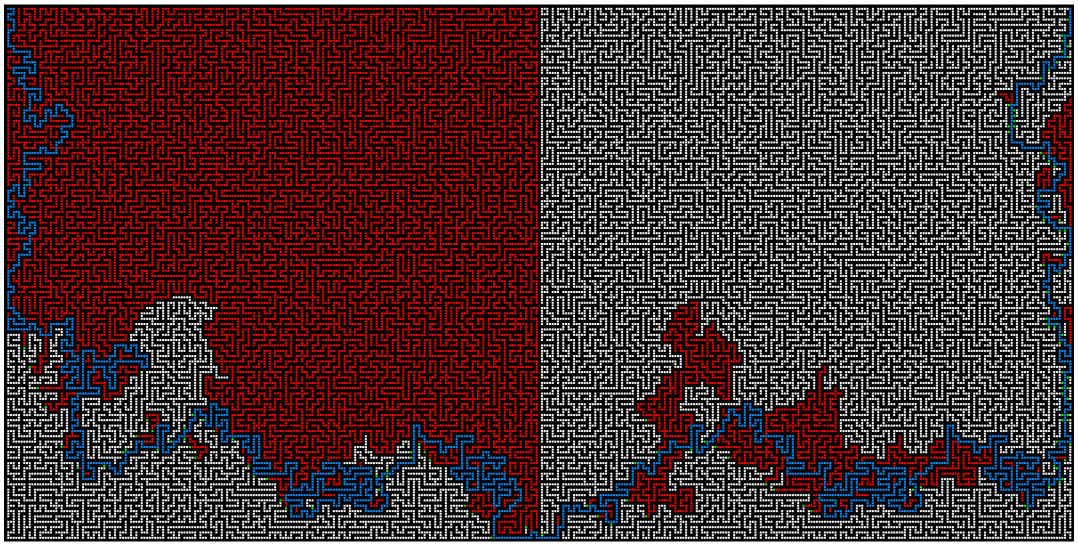


Figure 23: Greedy solution for grid 1

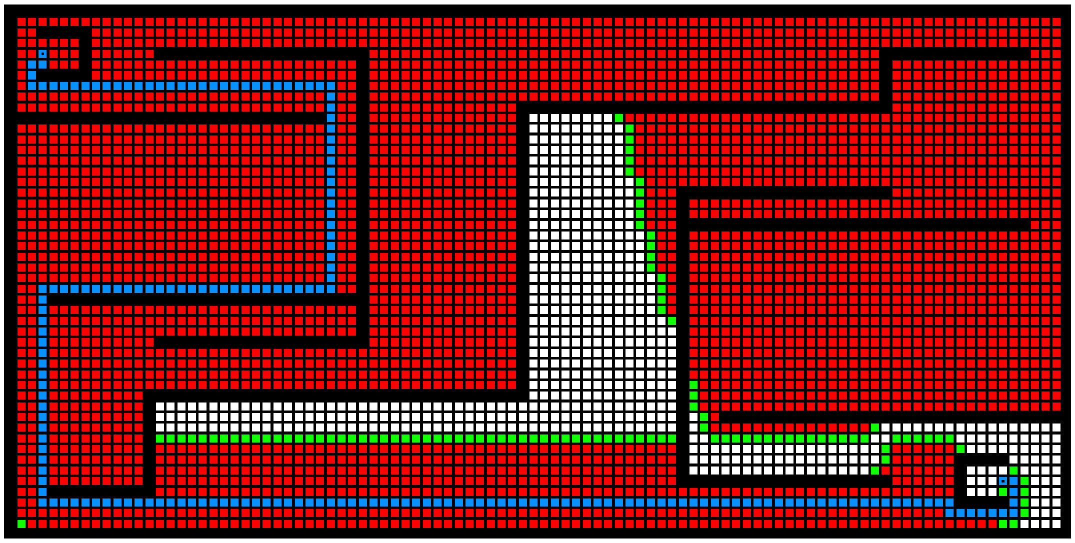


Figure 24: A* solution for grid 2

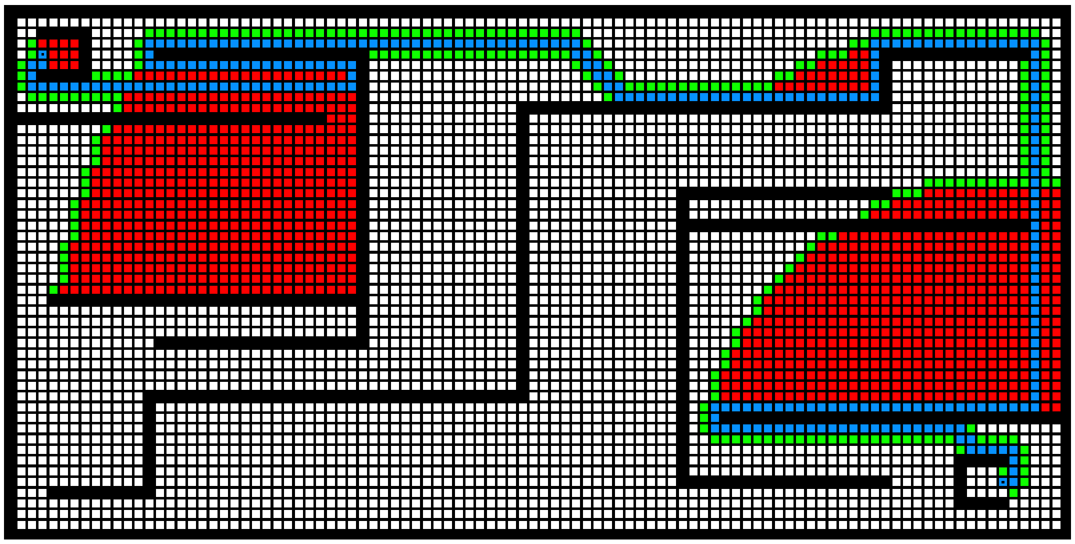


Figure 25: Greedy solution for grid 2

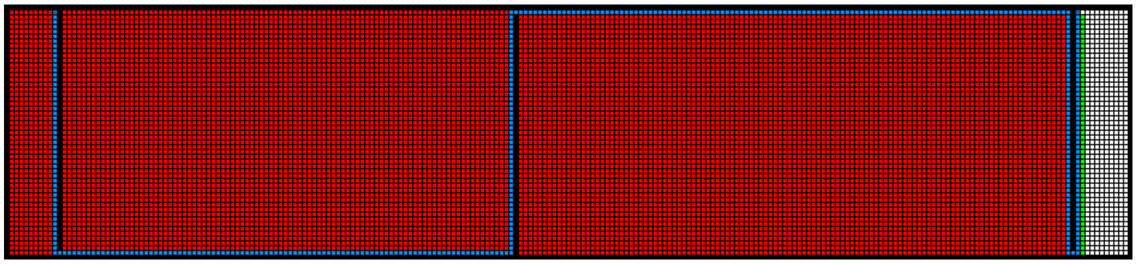


Figure 26: A* solution for grid 3

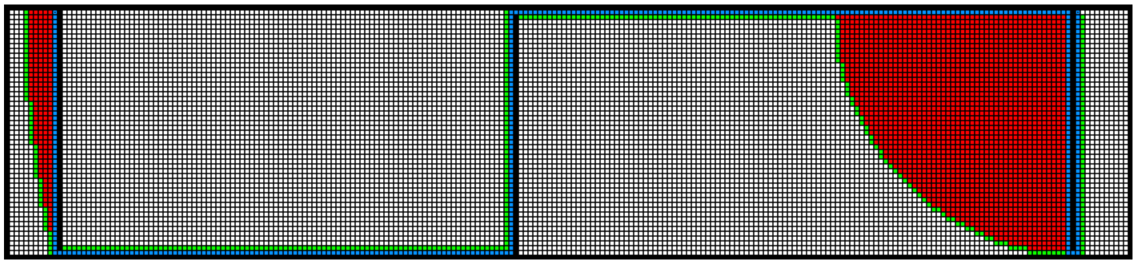


Figure 27: Greedy solution for grid 3

2.4 Analysis

2.4.1 Memory Consumption

One thing that is important to note is that the majority of the memory use was simply generating the grid (and all other Pygame processes) itself. The algorithms used $\leq 5\%$ of the reported memory use in all cases. It was expected the the difference between the algorithms would be relatively insignificant as there is not much data storage occurring in these algorithms.

2.4.2 Runtime

Looking back I'm not really sure why I thought A* would be better on grid 3, as it does make sense that simply rushing towards the solution in this situation would be lead to an efficient solution as well as a decreased runtime. However, even with this line of reasoning the average runtime of 44.19 seconds for Greedy to the nearly 5 minutes for A* is much higher (nearly a factor of six times better) than I would've anticipated[33].

Greedy was also significantly better in grid 2 as well, performing almost five times as fast as A*. I was expecting this, as while I attempted to design this grid to be somewhat similar to the example I showed in the **hypothesis**, the C shaped untraverseable sections of the map took up were not as massive with respect to the overall maze size. Thus, I doubted that they would cause much of a hindrance[34].

As expected, Greedy did perform better with respect to grid 1 as well, but it was again much better than I anticipated, performing almost a whole 10 minutes faster than A*. I had initally predicted a much closer runtime between the two, as the grid is split into two sections with the only point of navigation between the two being a single node at the bottom[35]. Since Greedy would simply try to get to B as fast as possible, I assumed it would quickly try to arrive there vertically, taking a long time to finally find the node at the bottom of the map. I predicted this correctly, but what gave Greedy most of its advantage was the second half of the maze. This section had a more simple and clear cut route to B. A*, as usual, tried to uniformly navigate though this section, while Greedy quick dash was more efficient.

2.4.3 Path length

Greedy actually found the most optimal path for grids 1 and 3. Furthermore, the path for grid 2 was not that much longer than what A* found–it was only a few nodes longer.

[33] I would hence predict that if we slightly change the map by adding a single "pipe" like structure on the left most side that is the sole connection to the end node, A* would outperform Greedy. This is because in this altered scenario, as Greedy rushes to the end node, the leftmost portion of the map would be the last section it considers. A* on the other hand, would search all sections of the map more evenly, meaning that it would find this "pipe" much faster than Greedy.

[34] And even considering the hypothesis, the overall node analysis count (# of colored nodes) was still a bit less for Greedy.

[35] It is difficult to see this since the image is small. Here is a link to the full size of the image: **Google Drive Link**

Conclusion

The guiding research question of this EE–"How does theory (big O space and time complexity) for A* and Greedy Best-first search compare with actual runtime and memory use?"–can be considered answered. Big O was correct in predicting that the memory use for A* and Greedy would be identical, but it was vastly off with respect to algorithm runtime, as time and time again, Greedy was faster by a considerable margin. General theory of each algorithms respective behaviors helped predict runtime and memory use to a much greater degree.

References

[1] Rafi Akhtar. Search algorithms in ai, Aug 2021. URL: https://www.geeksforgeeks.org/search-algorithms-in-ai/.

[2] Unknown Author. URL: https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm.

[3] Unknown Author. Asymptotic analysis: Big-o notation and more. URL: https://www.programiz.com/dsa/asymptotic-notations.

[4] Unknown Author. Big o notation. URL: https://web.mit.edu/16.070/www/lecture/big_o.pdf.

[5] Unknown Author. Complexity analysis. URL: https://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture2.html.

[6] Unknown Author. Informed search algorithms in ai - javatpoint. URL: https://www.javatpoint.com/ai-informed-search-algorithms.

[7] Unknown Author, Nov 2021. URL: https://en.wikipedia.org/wiki/Binary_search_tree.

[8] Unknown Author, Nov 2021. URL: https://en.wikipedia.org/wiki/Graph_theory.

[9] Unknown Author. Breadth first search or bfs for a graph, Oct 2021. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[10] Unknown Author. Time complexity: What is time complexity & algorithms of it?, Nov 2021. URL: https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

[11] Baeldung. Space complexity, Aug 2021. URL: https://www.baeldung.com/cs/space-complexity.

[12] Rachit Belwariar. A* search algorithm, Oct 2021. URL: https://www.geeksforgeeks.org/a-search-algorithm/.

[13] Subham Datta. Computing bubble sort time complexity, Oct 2020. URL: https://www.baeldung.com/cs/bubble-sort-time-complexity.

[14] Tuan Nhu Dinh. Graph data structure cheat sheet for coding interviews., Jun 2020. URL: https://towardsdatascience.com/graph-data-structure-cheat-sheet-for-coding-interviews-a38aadf8aa87#:~:text=foreveryvertex.-,TimecomplexityisO(VE)whereVis,ofedgesinthegraph.

[15] DW. A* graph search time-complexity, Jun 2016. URL: https://cs.stackexchange.com/questions/56176/a-graph-search-time-complexity.

[16] Ross Graham, Hugh McCabe, and Stephen Sheridan. Difference between graph and tree, 2003. URL: https://arrow.tudublin.ie/cgi/viewcontent.cgi?article=1063&context=itbj.

[17] Shen Huang. What is big o notation explained:space and time complexity, Jun 2021. URL: https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/.

[18] Hybesis H.urna. Pathfinding algorithms:the four pillars., Jan 2020. URL: https://medium.com/@urna.hybesis/pathfinding-algorithms-the-four-pillars-1ebad85d4c6b.

[19] Sebastian Lague, Dec 2014. URL: https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW.

[20] Amit Patel. Introduction to a*. URL: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm.

[21] Frank Q, amitamit, Luke Heytens, and sumayla khan. "what is the time and space complexity of a breadth first and depth first tree traversal?", Jun 2012. URL: https://stackoverflow.com/questions/9844193/what-is-the-time-and-space-complexity-of-a-breadth-first-and-depth-first-tree-tree.

[22] Eric Rowell and many others. Know thy complexities! URL: https://www.bigocheatsheet.com/.

[23] Toshad Salwekar. What is the time complexity of breadth first search traversal of a graph? URL: https://www.quora.com/What-is-the-time-complexity-of-Breadth-First-Search-Traversal-of-a-graph.

[24] Gurasis Singh. Algorithms: Breadth-first search vs. depth-first search, Feb 2020. URL: https://betterprogramming.pub/algorithms-searching-through-a-tree-33610e4577bd.

[25] Said Sryheni. Difference between bfs and dijkstra's algorithms, Aug 2021. URL: https://www.baeldung.com/cs/graph-algorithms-bfs-dijkstra.

[26] Marty Stepp. University of washington - cse 373 algorithm analysis and runtime complexity slides. URL: https://slideplayer.com/slide/9739625/.

[27] Romin Vaghani. Difference between graph and tree, Jan 2019. URL: https://www.geeksforgeeks.org/difference-between-graph-and-tree/.

[28] Viva, Apr 2021. URL: https://vivadifferences.com/12-difference-between-tree-and-graph/.

Appendices

A Further reading

OMITTED

B Maps used in EE

To run the code in the following appendices, you will need a folder called "mazes" located in the current working directory (cwd–the same folder that the program file (.py) is in). In this "mazes" folder, you can add mazes that you want the algorithms to be run on. These mazes are in the format of a .txt file. Each character in the file represents something in the maze. That is to say that each row of characters corresponds to each row of the maze displayed by pygame. The conventions that I used to create the maze are the following. This list is in the format character ⇒ maze object.

- * = wall
- . = empty space (or anything else, can literally be whitespace, i.e. " ")
- a = start location
- b = end location

For example, if we look at **Hypothesis**, the example inefficiency map would look like the following following the above convention (in a file ending with .txt).

```
********************
*..................*
*..................*
*...************...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*.a............*...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*...************...*
*................b.*
*..................*
********************
```

Using this convention, you can create your own maps for the algorithm to run. Once you have done so, you can add it to the map_sizes dictonary on line 171 in the code below. Simply add the name of the .txt file along with the dimensions and the recommended size each character should be as a node.

You can also change this convention by changing the code on lines 253-257 by changing the "char ==" lines. The other maps are too large to add here, since they are up to 200 times as large as the example above, but, as always, can be found on **Github**.

C Code used in EE (A* code, can be modified for Greedy)

This is the code that I used in this Extended Essay for the latter part of the essay that had the benchmarking with respect to the runtime and memory consumption for A* and Greedy Best-first search. Since the two are very closely related, I did not put two separate programs in the appendix, as there would be a lot of repeated code. To easily run Greedy from this code, simply copy paste then CTRL-F the term "!DELETE", this will show you all the lines you need to delete and/or modify to run greedy (i.e. on line 78 delete "Node.start_node.gcost + \" as the comment specifies).

There is a large, unmissable comment in the code that allows you to change the map/grid as well (line 203).

If you are unfamiliar with the escape character \, I am using it to make sure that the code fits within the vertical margins, as without the character, some lines would be too long. For example, take line 78 again. With the character it reads as:

```
Node.start_node.fCost = Node.start_node.gCost + \
                        Node.start_node.hCost
```

Which is identical to

```
Node.start_node.fCost = Node.start_node.gCost + Node.start_node.hCost
```

in the eyes of the python interpreter, but beneficial for the reader. If you get errors with the code due to these (as the copy paste may not paste characters exactly, so the \may escape a space instead of an enter character), simply delete them as the compiler ignores them anyway.

```
1   import pygame as pg
2   from math import hypot
3
4   # Benchmarking libraries
5   from time import sleep, time
6
7   import os, psutil
8   process = psutil.Process(os.getpid())
9
10
11  # Occupies space where nodes do not exist
12  class Wall(pg.sprite.Sprite):
13
14      def __init__(self, pos, game):
15          self.groups = game.all_sprites
16          pg.sprite.Sprite.__init__(self, self.groups)
17          self.size = game.NODE_SIZE
18          self.image = game.BLACK_NODE
19          self.rect = self.image.get_rect()
20          self.x, self.y = pos
21          self.rect.x, self.rect.y = [i*game.NODE_SIZE for i in pos]
22
23  class Node(pg.sprite.Sprite):
24      all_nodes = {}
25      init_counter = 0
26      start_node = None
27      end_node = None
```

```
    seen_end = False

    def __init__(self, pos, game):
        self.game = game
        self.groups = self.game.all_sprites
        pg.sprite.Sprite.__init__(self, self.groups)
        self.size = game.NODE_SIZE
        self.image = game.WHITE_NODE
        self.rect = self.image.get_rect()
        self.x, self.y = pos
        self.rect.x, self.rect.y = [i*game.NODE_SIZE for i in pos]
        self.parent = None
        self.start = False
        self.end = False
        self.hCost = None
        # !DELETE line below for Greedy
        self.gCost = None
        self.fCost = None
        self.seen = False
        self.discovered = False
        self.end_node = []
        Node.all_nodes[str(pos)] = self

    @classmethod
    def exists(cls, pos):
        return cls.all_nodes[str(pos)] if str(pos) in cls.all_nodes else False

    def kill_node(self):
        Wall([self.x, self.y], self.game)
        Node.all_nodes[f"[{self.x}, {self.y}]"] = None
        self.kill()

    # Place start node
    def set_start(self):
        self.image = self.game.START_NODE
        self.start = True
        self.gCost = 0
        self.hCost = None
        Node.start_node = self
        sleep(.2)

    # Place end node
    def set_end(self):
        self.image = self.game.END_NODE
        self.end = True
        self.hCost = 0
        Node.end_node = self
        Node.start_node.hCost = hypot(Node.start_node.x-self.x, \
                                      Node.start_node.y-self.y)
        # !DELETE gCost for Greedy
        Node.start_node.fCost = Node.start_node.gCost + \
                                    Node.start_node.hCost

    # "Scans" all nodes surrounding a node
    def scan(self):

        if not self.discovered and not self.start and \
           not (self.end and Node.seen_end):
            return

        if self.end:
            return self.gen_final_path()

        # Change color
        if not self.start and not self.end:
            self.image = self.game.RED_NODE
            self.seen = True

        # Generate a list of surrounding nodes to update
        check = []
        for x in range(-1,2):
            for y in range(-1,2):
                check.append([self.x+x, self.y+y])

        # The following 3 lines of code disallow diagonal movement
        # Comment or !DELETE to allow it
        #rem_list = [[-1, -1], [-1, 1], [1,1], [1,-1]]
        #for i in rem_list:
        #    check.remove([self.x+i[0], self.y+i[1]])

        check.remove([self.x,self.y])

        # Update them if they exist
        for node_pos in check:
            node = Node.exists(node_pos)
            if node:
                node.update_cost(self)


    def update_cost(self, parent):

        if self.seen or self.start:
            return

        if not self.end:
            self.image = self.game.GREEN_NODE
            self.discovered = True
        else:
            Node.seen_end = True

        # !DELETE all code that follows for this function for Greedy
        dist = hypot(self.x-parent.x, self.y-parent.y)
        if not self.gCost or self.gCost > parent.gCost + dist:
            self.parent = parent
            self.gCost = self.parent.gCost + dist

        if not self.hCost:
            self.hCost = hypot(self.x-Node.end_node.x, self.y-Node.end_node.y)

        self.fCost = self.gCost + self.hCost

        # After !DELETEing above, add the following:
        '''
        if not self.hCost or self.hCost < parent.hCost:
            self.parent = parent
            self.hCost = hypot(self.x-Node.end_node.x, self.y-Node.end_node.y)

        self.fCost = self.hCost
        '''


    def gen_final_path(self):

        node_list = []
        parent_node = self.parent

        while not parent_node.start:
            node_list.append(parent_node)
            parent_node = parent_node.parent

        for node in node_list:
            node.image = self.game.BLUE_NODE

        # For benchmarking
        # !DELETE to see completed graph
        #global g
        #g.running = False


class Game:
    start_node = None
    end_node = None
    running = True
    map_sizes = {
                    'map1':{
                        'size': [401, 201],
                        'recommended_node_size': 4,
                        },
                    'map2':{
                        'size':[100,50],
                        'recommended_node_size': 4,
                        },
                    'map3':{
                        'size':[235,53],
                        'recommended_node_size': 4,
                        },
                    2:{
                        'size': [18, 18],
                        'recommended_node_size': 10,
                        },
                    'example':{
                        'size': [10, 5],
                        'recommended_node_size': 10,
                    },
                    'inefficiencyExample':{
                        'size': [20,20],
                        'recommended_node_size': 10,
                    }
                }

    def __init__(self):

        ###############################################################
        # !CHANGE THIS VARIABLE TO CHANGE MAP, LOOK ABOVE FOR OPTIONS #
        ###############################################################
        self.map = 'inefficiencyExample'

        self.WIDTH, self.HEIGHT = Game.map_sizes[self.map]['size']
        self.NODE_SIZE = Game.map_sizes[self.map]['recommended_node_size']

        self.screen = pg.display.set_mode((self.WIDTH*self.NODE_SIZE, \
                                   self.HEIGHT*self.NODE_SIZE), pg.NOFRAME)
        self.running = True
        self.clock = pg.time.Clock()
        self.WHITE_NODE = pg.transform.scale(\
            pg.image.load('img\\White.png').convert(), \
            (self.NODE_SIZE, self.NODE_SIZE))
        self.BLACK_NODE = pg.transform.scale(\
            pg.image.load('img\\Black.png').convert(), \
            (self.NODE_SIZE, self.NODE_SIZE))
        self.RED_NODE = pg.transform.scale(\
            pg.image.load('img\\Red.png').convert(), \
            (self.NODE_SIZE, self.NODE_SIZE))
        self.GREEN_NODE = pg.transform.scale(\
            pg.image.load('img\\Green.png').convert(), \
            (self.NODE_SIZE, self.NODE_SIZE))
        self.START_NODE = pg.transform.scale(\
            pg.image.load('img\\Start.png').convert(), \
            (self.NODE_SIZE, self.NODE_SIZE))
        self.END_NODE = pg.transform.scale(\
            pg.image.load('img\\End.png').convert(), \
            (self.NODE_SIZE, self.NODE_SIZE))
        self.BLUE_NODE = pg.transform.scale(\
            pg.image.load('img\\Blue.png').convert(), \
                (self.NODE_SIZE, self.NODE_SIZE))


    def new(self):
        self.all_sprites = pg.sprite.Group()

        # Place all nodes on map
        for row in range(self.WIDTH):
            for col in range(self.HEIGHT):
                Node([row,col], self)

        # Generate node map from txt file
        # This changes the already existing nodes
        # You can change this to add a char == "."
        # And use that to add a Node to the xy pos
        with open(f'mazes\{self.map}.txt', 'r') as f:
            x=0
            for line in f:
                y=0
                for char in line:
                    if char == '*':
                        Node.all_nodes[f"[{y}, {x}]"].kill_node()
                    elif char == 'a':
                        Node.all_nodes[f"[{y}, {x}]"].set_start()
                    elif char == 'b':
                        end_node_loc = [y, x]
                    y+=1
                x+=1
            Node.all_nodes[str(end_node_loc)].set_end()

        # All code other than self.run() is for benchmarking
        start = time()
        self.run()
        print(process.memory_info().rss)
        print(time()-start)

    # Visualizer run function, contains all tasks
    def run(self):
        while self.running:
            self.clock.tick(60)
            self.update()
            self.draw()

    # Update function that actually "runs" the algorithms
    def update(self):

        found_nodes = {}
        for node in Node.all_nodes.values():
            if node and node.discovered:
                found_nodes[node.fCost] = node
        if Node.seen_end:
                Node.end_node.scan()
                self.run_vis = False
            elif len(found_nodes) == 0:
                Node.start_node.scan()
            else:
                found_nodes[min(found_nodes)].scan()
                found_nodes[min(found_nodes)].discovered = False

    # Draw everything on the screen
    def draw(self):
        self.all_sprites.draw(self.screen)
        pg.display.flip()


# Run the visualizer
g = Game()
g.new()
pg.quit()
```

C.1 Step by Step Algorithm program code

This is where you will find the program that I used in the beginning of the paper for the visualizations to demonstrate the individual steps of A* and Greedy Best-first search. Instructions for how to use the program are given in the beginning of the program in a long comment. The steps are the same to convert this algorithm into Greedy as mentioned above. There is only 3 additional lines that need to be deleted (lines 264-269). This program allows you to "draw" your own maps. If you want the maps to be automatically generated by the program, follow the instructions in the comments directly below the import statements.

```
1   import pygame as pg
2   from math import hypot
3   import time
4   
5   '''
6   A program to visualize how A* pathfinding works
7   You want to click on the block with the smallest number
8   until you eventually reach the end block
9   
10  To place the starting node and ending node, left click
11  To place walls/obstacles, right click
12  To restart press the r key
13  '''
14  
15  class Wall(pg.sprite.Sprite):
16  
17      def __init__(self, pos, game):
18          self.groups = game.all_sprites
19          pg.sprite.Sprite.__init__(self, self.groups)
20          self.size = 32
21          self.image = pg.image.load('img\\Black.png').convert()
22          self.rect = self.image.get_rect()
23          self.x, self.y = pos
24          self.rect.x, self.rect.y = [i*32 for i in pos]
25  
26  class Node(pg.sprite.Sprite):
27      all_nodes = {}
28      init_counter = 0
29      start_node = None
30      end_node = None
31      seen_end = False
32  
33      def __init__(self, pos, game):
34          self.game = game
35          self.groups = self.game.all_sprites
36          pg.sprite.Sprite.__init__(self, self.groups)
37          self.size = 32
38          self.image = pg.image.load('img\\White.png').convert()
39          self.rect = self.image.get_rect()
40          self.x, self.y = pos
41          self.rect.x, self.rect.y = [i*32 for i in pos]
42          self.parent = None
```

```
        self.start = False
        self.end = False
        self.hCost = None
        self.gCost = None
        self.fCost = None
        self.seen = False
        self.discovered = False
        self.end_node = []
        Node.all_nodes[str(pos)] = self

    @classmethod
    def exists(cls, pos):
        return cls.all_nodes[str(pos)] if str(pos) in cls.all_nodes else False

    def clicked(self, setup=False):
        if pg.mouse.get_pressed()[0] and \
           self.rect.collidepoint(pg.mouse.get_pos()):
            if Node.click_counter == 0:
                self.set_start()
            elif Node.click_counter == 1:
                self.set_end()
            else:
                self.scan()
            Node.click_counter+=1
        elif pg.mouse.get_pressed()[2] and \
             self.rect.collidepoint(pg.mouse.get_pos()):
            self.kill_node()

    def kill_node(self):
        Wall([self.x, self.y], self.game)
        Node.all_nodes[f"[{self.x}, {self.y}]"] = None
        self.kill()

    def set_start(self):
        self.image = pg.image.load('img\\Start.png').convert()
        self.start = True
        self.gCost = 0
        self.hCost = None
        Node.start_node = self
        time.sleep(.2)

    def set_end(self):
        self.image = pg.image.load('img\\End.png').convert()
        self.end = True
        self.hCost = 0
        Node.end_node = self
        Node.start_node.hCost = hypot(Node.start_node.x-self.x, \
                                      Node.start_node.y-self.y)
        Node.start_node.fCost = Node.start_node.gCost + Node.start_node.hCost
        time.sleep(.2)

```

```python
    def scan(self):

        if not self.discovered and not self.start and not\
           (self.end and Node.seen_end):
            return

        if self.end:
            return self.gen_final_path()

        # Change color
        if not self.start and not self.end:
            self.image = pg.image.load('img\\Red.png').convert()
            self.seen = True

        # Generate a list of surrounding nodes to update
        check = []
        for x in range(-1,2):
            for y in range(-1,2):
                check.append([self.x+x, self.y+y])
        check.remove([self.x,self.y])

        # Update them if they exist
        for node_pos in check:
            node = Node.exists(node_pos)
            if node:
                node.update_cost(self)


    def update_cost(self, parent):

        if self.seen or self.start:
            return

        if not self.end:
            self.image = pg.image.load('img\\Green.png').convert()
            self.discovered = True
        else:
            Node.seen_end = True

        dist = hypot(self.x-parent.x, self.y-parent.y)
        if not self.gCost or self.gCost > parent.gCost + dist:
            self.parent = parent
            self.gCost = self.parent.gCost + dist

        if not self.hCost:
            self.hCost = hypot(self.x-Node.end_node.x, self.y-Node.end_node.y)

        self.fCost = self.gCost + self.hCost


    def gen_final_path(self):

        node_list = []
        parent_node = self.parent

        while not parent_node.start:
            node_list.append(parent_node)
            parent_node = parent_node.parent

        for node in node_list:
            node.image = pg.image.load('img\\Blue.png').convert()


class Game:
    start_node = None
    end_node = None
    running = True
    map_sizes = {
                    'map1':{
                        'size': [401, 201],
                        'recommended_node_size': 4,
                        },
                    'map2':{
                        'size':[100,50],
                        'recommended_node_size': 4,
                        },
                    'map3':{
                        'size':[235,53],
                        'recommended_node_size': 4,
                        },
                    2:{
                        'size': [18, 18],
                        'recommended_node_size': 10,
                        },
                    'example':{
                        'size': [10, 5],
                        'recommended_node_size': 10,
                    },
                    'inefficiencyExample':{
                        'size': [20,20],
                        'recommended_node_size': 10,
                    }
                }

    def __init__(self):
        pg.init()
        self.WIDTH = int(input('W: '))
        self.HEIGHT = int(input('H: '))

        # Uncomment the 3 following lines for map generation
        #self.map = 3

        #self.WIDTH, self.HEIGHT = Game.map_sizes[self.map]['size']
        self.NODE_SIZE = 10#Game.map_sizes[self.map]['recommended_node_size']

        # You don't need this line for Greedy
        self.disp_costs = True if input('Show g and h costs(y/n)?').lower() == \
                          'y' else False
        self.screen = pg.display.set_mode((self.WIDTH*32, self.HEIGHT*32))
        self.running = True
        self.clock = pg.time.Clock()

    def new(self):

        # Restart stuff here
        self.setup = True
        self.all_sprites = pg.sprite.Group()
        for row in range(self.WIDTH):
            for col in range(self.HEIGHT):
                Node([row,col], self)

        # Uncomment for map generation
        '''
        with open(f'mazes\{self.map}.txt', 'r') as f:
            x=0
            for line in f:
                y=0
                for char in line:
                    if char == '*':
                        Node.all_nodes[f"[{y}, {x}]"].kill_node()
                    elif char == 'a':
                        Node.all_nodes[f"[{y}, {x}]"].set_start()
                    elif char == 'b':
                        end_node_loc = [y, y]
                    y+=1
                x+=1
            Node.all_nodes[str(end_node_loc)].set_end()
        '''

        self.run()

    def run(self):
        while self.running:
            self.clock.tick(60)
            self.events()
            self.draw()

    def events(self):

        if self.setup:
            Node.click_counter = 0
            self.setup = False

        for event in pg.event.get():
            if event.type == pg.QUIT:
                self.running = False
            elif event.type == pg.KEYDOWN:
                key = pg.key.get_pressed()
                if event.key == pg.K_r:
                    g.new()

        for node in Node.all_nodes.values():
            if node:
                node.clicked()

    def draw(self):
        self.all_sprites.draw(self.screen)
        for node in Node.all_nodes.values():
            if node and node.discovered:
                # !DELETE all 3 lines below for Greedy
                if self.disp_costs:
                    self.draw_text(str(int(node.gCost*10)), \
                                   15, (0, 0, 0), node.rect.x+7, node.rect.y+5)
                    self.draw_text(str(int(node.hCost*10)), \
                                   15, (0, 0, 0), node.rect.x+node.size-7, \
                                   node.rect.y+5)
                self.draw_text(str(int(node.fCost*10)), \
                               20, (0, 0, 0), node.rect.x+node.size/2, \
                               node.rect.y+node.size/2)
        pg.display.flip()

    def draw_text(self, text, size, color , x, y, font_name = ''):
        font = pg.font.Font(pg.font.match_font(font_name), size)
        text_surface = font.render(text, True, color)
        text_rect = text_surface.get_rect()
        text_rect.midtop = (x, y)
        self.screen.blit(text_surface, text_rect)

g = Game()
g.new()
pg.quit()
```

D Algorithm code for speed (No vis)

The previous appendices showed the code that was used in this Extended Essay, which was made specifically for visualization purposes. The following code shows a more “optimized” version of the two algorithms since Pygame is not included in them. I found that the runtime for A* for grid 1 (the maze) was around 2 and a half minutes, which was about a factor of 10 times faster than it was with the code in **Appendix C**. To go from the code there to this “optimized” code, you would simply need to remove all references of Pygame and refactor the last section. Even though there is a lot of repeated code between these two versions, I am including this in the appendix since there is a lot you must delete/change to be able to covert the program to this version, and it would be difficult to comment out *everything* that would need to be deleted and/or modified. Once again, this is A* code, and to change it to Greedy, simply delete all mentions of G cost as explained in the comments of the code located in **Appendix C**.

The solution will be output as a .txt in the CWD. This will follow the same conventions as the map, with the exception that the final path is plotted with the # character. For example, to solve the inefficiency example, the solution would be the following:

```
********************
*..................*
*..................*
*...************...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*..............*...*
*.A#...........*...*
*..#...........*...*
*..#...........*...*
*..#...........*...*
*..#...........*...*
*..#...........*...*
*..#...........*...*
*..#************...*
*..##############B.*
*..................*
********************
```

Keep in mind that I removed diagonal movement, which is why it is different than the solution that was previously presented (this is because it looked odd in textfiles), you can re-enable it by commenting out/deleting lines 63-65 and uncommenting the line that follows.

```
1  from math import hypot
2  from time import time
3
4  class Node:
5      all_nodes = {}
6      init_counter = 0
7      start_node = None
8      end_node = None
9      seen_end = False
10
11     def __init__(self, pos):
        self.x, self.y = pos
        self.parent = None
        self.start = False
        self.end = False
        self.hCost = None
        self.gCost = None
        self.fCost = None
        self.seen = False
        self.discovered = False
        self.end_node = []
        self.final = False
        Node.all_nodes[str(pos)] = self

    @classmethod
    def exists(cls, pos):
        return cls.all_nodes[str(pos)] if str(pos) in cls.all_nodes else False

    def kill_node(self):
        Node.all_nodes[f"[{self.x}, {self.y}]"] = None

    def set_start(self):
        self.start = True
        self.gCost = 0
        self.hCost = None
        Node.start_node = self

    def set_end(self):
        self.end = True
        self.hCost = 0
        Node.end_node = self
        Node.start_node.hCost = hypot(Node.start_node.x-self.x, \
                                      Node.start_node.y-self.y)
        Node.start_node.fCost = Node.start_node.gCost + Node.start_node.hCost

    def scan(self):

        if not self.discovered and not self.start and not \
           (self.end and Node.seen_end):
            return

        if self.end:
            return self.gen_final_path()

        # Change color
        if not self.start and not self.end:
            self.seen = True

        # Generate a list of surrounding nodes to update
        check = []
        for x in range(-1,2):
            for y in range(-1,2):
                check.append([self.x+x, self.y+y])
        rem_list = [[-1, -1], [-1, 1], [0,0], [1,1], [1,-1]]
        for i in rem_list:
            check.remove([self.x+i[0], self.y+i[1]])
        #check.remove([self.x,self.y])

        # Update them if they exist
        for node_pos in check:
            node = Node.exists(node_pos)
            if node:
                node.update_cost(self)


    def update_cost(self, parent):

        if self.seen or self.start:
            return

        if not self.end:
            self.discovered = True
        else:
            Node.seen_end = True

        dist = hypot(self.x-parent.x, self.y-parent.y)
        if not self.gCost or self.gCost > parent.gCost + dist:
            self.parent = parent
            self.gCost = self.parent.gCost + dist

        if not self.hCost:
            self.hCost = hypot(self.x-Node.end_node.x, self.y-Node.end_node.y)

        self.fCost = self.gCost + self.hCost


    def gen_final_path(self):

        node_list = []
        parent_node = self.parent

        while not parent_node.start:
            node_list.append(parent_node)
            parent_node = parent_node.parent

        for node in node_list:
            node.final = True

        global running
        running = False


start_node = None
```

```
end_node = None
running = True
max_x = 0
max_y = 0
map_sizes = {
    'map1':[401, 201],
    'map2':[100,50],
    'map3':[235,53],
    2:[18, 18],
    'example':[10, 5],
    'inefficiencyExample':[20,20],
}
mapOption = 'map1'

def main():

    for row in range(map_sizes[mapOption][0]):
        for col in range(map_sizes[mapOption][1]):
            Node([row,col])

    # Restart stuff here
    with open(f'mazes\\{mapOption}.txt', 'r') as f:
        x=0
        for line in f:
            y=0
            for char in line:
                if char == '*':
                    Node.all_nodes[f"[{y}, {x}]"].kill_node()
                elif char == 'a':
                    Node.all_nodes[f"[{y}, {x}]"].set_start()
                elif char == 'b':
                    end_node_loc = [y, x]
                y+=1
                global max_y
                max_y = max(max_y, y)
            x+=1
            global max_x
            max_x = max(max_x, x)
        Node.all_nodes[str(end_node_loc)].set_end()

def update():

    found_nodes = {}
    for node in Node.all_nodes.values():
        if node and node.discovered:
            found_nodes[node.fCost] = node
    if Node.seen_end:
        Node.end_node.scan()
    elif len(found_nodes) == 0:
        Node.start_node.scan()
    else:
```

```python
        found_nodes[min(found_nodes)].scan()
        found_nodes[min(found_nodes)].discovered = False

def show():
    s = ''
    with open('out.txt', 'w') as f:
        for x in range(max_x):
            for y in range(max_y-1):
                if Node.exists([y,x]) and Node.all_nodes[f"[{y}, {x}]"].final:
                    s+='#'
                elif Node.exists([y,x]) and Node.all_nodes[f"[{y}, {x}]"].start:
                    s+='A'
                elif Node.exists([y,x]) and Node.all_nodes[f"[{y}, {x}]"].end:
                    s+='B'
                elif Node.exists([y,x]):
                    s+='.'
                else:
                    s+='*'
            s+='\n'
        f.write(s)


start = time()
main()
while running:
    update()
show()
print('Solved in: ' + str(round(time()-start,3)) + ' seconds')
```

E Omitted Sections

E.1 Big O application: Bubble Sort (formerly 1.1.1)

To demonstrate an application of big O notation, let us use the algorithm Bubble Sort. Bubble sort is a relatively common sorting algorithm that is one of the first many programmers study due to its simplicity. Bubble sort iterates through a array of n elements from index $i = 0$ to $i = n-1$. For each element at index index i we compare if the element at index $i+1$ is greater than it and swap them if that condition is true. After each "pass" of the algorithm (a pass refers through one full iteration through the array), the element that was last swapped is considered fully sorted and exempt from the sorting process[2]. This means that each pass causes the array to have $n = n-1$ elements in need of sorting. To summarize, we start with $n-1$ comparisons and decrease comparisons by 1 each pass. Mathematically we can represent this as follows:

$$(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

The sum of the comparisons is equal to the series $\sum_{k=0}^{n-1} k$ (i.e. $T(n) = \sum_{k=0}^{n-1} k$), and by the definition of the sum of an arithmetic series ($S_n = \frac{1}{2}(a_1 + a_n)$), we can evaluate the series to be equal to $\frac{(n-1)((n-1)+1)}{2} = \frac{n(n-1)}{2}$. We can expand this to get $\frac{1}{2}n^2 - \frac{1}{2}n$ as the number of operations. This means the time complexity of Bubble sort is simply $\mathcal{O}(n^2)$ or *quadratic time*[13]. While it may be easy to calculate the exact number of calculations for bubble sort, analyzing the time complexity for other algorithms is much more conceptual.

As for the space complexity of Bubble Sort, it is $\mathcal{O}(1)$ or *constant space*. This is because the Bubble Sort algorithm sorts in place, meaning it does not occupy any additional memory to store the array itself. It only requires a few extra variables (for swapping and iterating) which are independent of the input size n[13].

E.2 Big Omega and Big Theta (formerly 1.1.2)

A caveat to consider here is that Big O notation represents the *worst case complexity* of an algorithm as mathematically, it is simply **upper bound** of the number of computations. This is the most used notation, as often we care more about the most problematic situations that could arise. Other symbols that may be used are Ω and Θ, which stand for the *best case complexity* and *average complexity*, and are mathematically the **lower** and **exact**(meaning both upper *and* lower) bounds respectively[4]. In the case of the bubble sort algorithm, the worst, best and average case complexities are the same[36]. Two figures[3] are shown below of the mathematical representations of both Ω and Θ:

[36]However, a common modification to bubble sort is to terminate the algorithm if no swaps were made, meaning in best case (where the array is already sorted), bubble sort is $\Omega(n)$

Figure 28: Representation of $\Omega(n)$

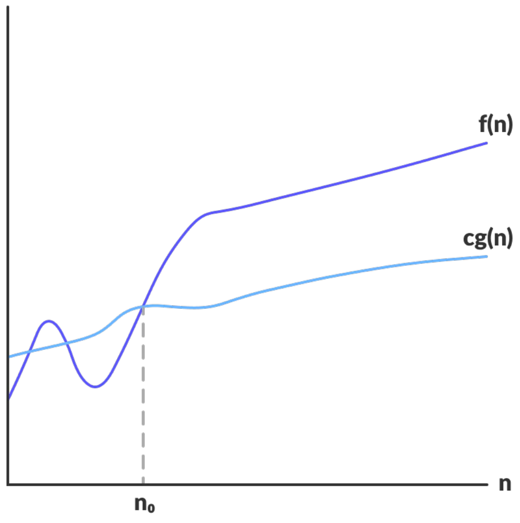
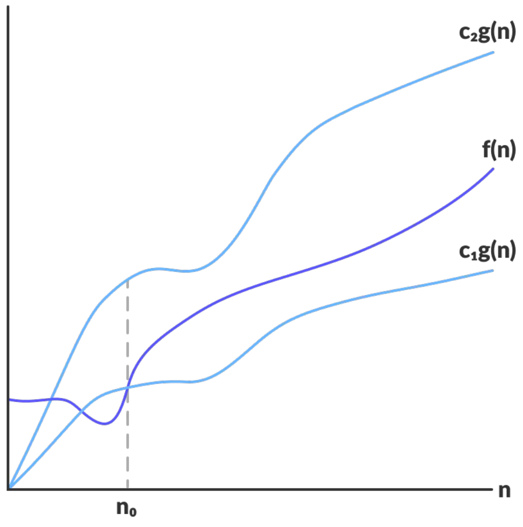


Figure 29: Representation of $\Theta(n)$



E.3 Basics of Pathfinding (formerly directly under 1.2)

The goal of pathfinding lies in the name itself; Pathfinding simply aims to find a *path* between a starting location and ending location. A more strenuous definition is as follows: Pathfinding algorithms address the problem of finding a path from a source to a destination avoiding obstacles and minimizing the costs (time, distance, risks, fuel, price, etc.)[18]. Pathfinding algorithms are primarily divided into two main groups: Informed and Uninformed searches [1]. Informed searches require knowledge of the location of the ending node while uninformed searches do not. An analogy of informed searches vs uninformed searches would be like trying to find the exit in a maze with vs without a map.

Uninformed searches mainly consist of Breadth-first search(BFS), Depth-first search(DFS), and Dijkstra's algorithm. BFS explores nodes nearest to it first, while DFS explores nodes furthest away. Dijkstra's is essentially just BFS, but with the possibility of weighted distance between nodes[37]. Out of these three, only DFS does not guarantee the shortest path[25]. To get more in-depth than this, we will first need to understand "where" pathfinding algorithms are run.

[37]However, the edges cannot have any negative weights