103163537

My Report on Assignment 1 of Intro To AI

Assignment 1

Option B

103163537

Table of Contents

[Instructions on how to use Program. 2](#_Toc132964354)

[Introduction 2](#_Toc132964355)

[Search Algorithms 3](#_Toc132964356)

[Uninformed Search Algorithms 3](#_Toc132964357)

[Depth First Search (DFS) 3](#_Toc132964358)

[Breadth First Search (BFS) 4](#_Toc132964359)

[Uniform Cost Search algorithm (Custom 2) 4](#_Toc132964360)

[Informed 5](#_Toc132964361)

[Greedy Best first Search (GBFS) 5](#_Toc132964362)

[A\* Search Algorithm 6](#_Toc132964363)

[Custom Algorithm 1 (diagonal A\*) 7](#_Toc132964364)

[Implementation 7](#_Toc132964365)

[Uninformed 8](#_Toc132964366)

[Depth First Search 8](#_Toc132964367)

[Breadth First Search 9](#_Toc132964368)

[Uniform Cost Search 9](#_Toc132964369)

[Informed 10](#_Toc132964370)

[A\* Search algorithms 10](#_Toc132964371)

[Greedy Best First Search 11](#_Toc132964372)

[Features/Bugs/Missing 12](#_Toc132964373)

# Instructions on how to use Program.

1. Unzip the .zip file.
2. Navigate to the unzipped location in terminal.
3. In the terminal type *Search <Location of file> <algorithm>*



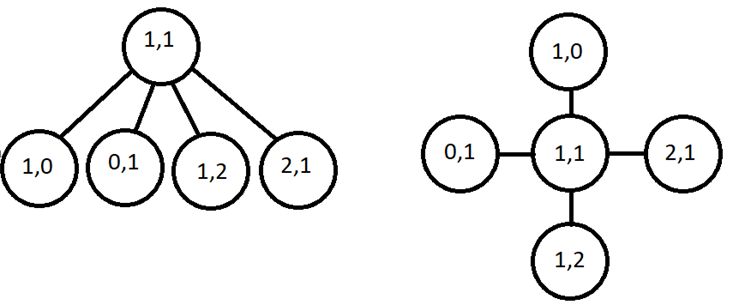
The Program will then output the map onto the console.

# Introduction

The Robot Navigation issue is a classic maze issue that puts several search algorithms to the test on search trees. A starting and ending point are usually supplied, and the search algorithms are compared to see how well they discover the solution.

A 2D array, a start points, a goal point, and several obstacles were provided for this project. The aim was to reassemble this data into a universe that search algorithms could traverse, and then to develop four separate search algorithms, as well as two optional bespoke search algorithms. Depth-First Search, Breadth-First Search, Greedy Best First Search, and A\* Search were the needed algorithms. The output required that I return the filename, method, number of nodes scanned, and the path taken.

To understand this problem, graph and tree concepts should be introduced. The basic principle is that a node can have several child nodes, and by expanding all potential nodes, a tree network representing the whole data set is formed. For example, the very top parent node may be (1,1) to indicate the initial position, with four child nodes surrounding it being "up" (1,0), "left" (0,1), "down" (1,2), and "right" (2,1). Each of them, in turn, may have its own child nodes. The full data set, as well as (ideally) the desired state, would be obtained by branching the entire tree out.



# Search Algorithms

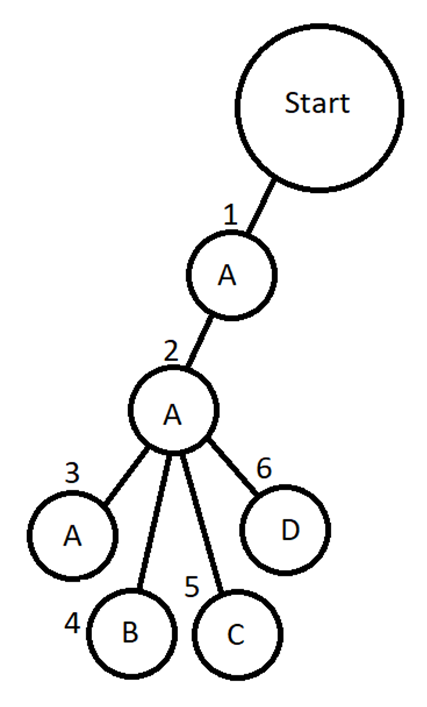
## Uninformed Search Algorithms

Uninformed Search algorithms, or blind search algorithms, are a type of search algorithm that has no additional information on the goal node other than what is provided in the problem definition. In the context of the Assignment the uninformed search algorithms know the start location and the goal location. These algorithms differentiate only between goal and non-goal states and can’t inspect the inner structure of a state to estimate how close it is to the goal. There are 5 kinds of Uninformed Search algorithms, Breadth First Search (BFS), Depth First Search (DFS), Uniform Cost Search (UCS), Iterative Deepening, and Bidirectional Search. In this report I will discuss DFS, BFS, and UCS, as these are the algorithms implemented in the program.

## Depth First Search (DFS)

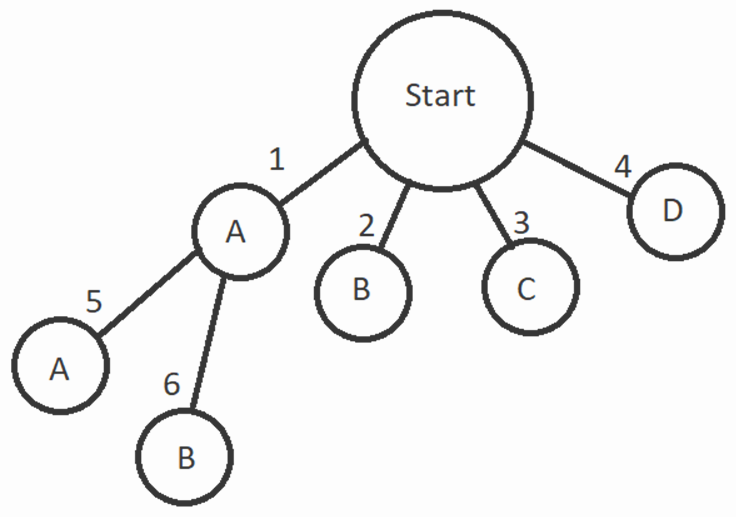
DFS is an algorithm for searching tree or graph data structures. It starts at the root node and explores as far as possible along each branch before back tracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph. The basic idea is to start from the root node and mark the node, then move to an adjacent unopened node, and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

Given a tree diagram with three stages of nodes, each with the letters A, B, C, and D, its search path would be 'A, AA, AAA, AAB, AAC, AAD, ABA, ABB, ABC, ABD...' and would continue in this pattern until it reached the end. This is depicted visually by a tree diagram in which one side of the tree is searched first before shifting to the other side.



## Breadth First Search (BFS)

BFS is a very similar to DFS in that it is another algorithm used for searching trees or graphs. It differs from DFS in the fact that it starts at the root node and explores all nodes at the current depth before moving on to the next layer. A queue is used to keep track of the child nodes that were encountered but not explored. With the same naming structure as DFS, the search path would be 'A, B, C, D, AA, AB...' and so on to the end.



## Uniform Cost Search algorithm (Custom 2)

Uniform Cost Search (UCS) is a version of Dijkstra’s algorithm. It is used to determine the path with the lowest cumulative cost in a weighted network where nodes are extended based on their traversal cost from the root node. This is accomplished using a priority queue, where the smaller the cost, the higher the priority.

Uniform Cost Search (UCS) is a weighted network search technique that finds the lowest cost path from a given start node to a goal node. The algorithm grows the graph nodes in the order of their cumulative cost from the start node. It considers all paths with costs less than a certain threshold and selects the path with the lowest cost to grow next.

The algorithm keeps a priority queue of nodes that need to be extended. Initially, only the start node with a cost of zero is in the queue. The algorithm eliminates the node with the lowest cost from the queue and grows it by examining all its neighbours at each iteration. If a neighbour has never been visited, it is added to the queue at a cost equal to the sum of its own and the current node's costs. If the neighbour has previously been visited, the cost is updated if the new cost is less than the previous cost.

When the goal node is extended or the queue is empty, the algorithm quits, indicating that there is no path from the start node to the goal node. Backtracking from the destination node to the start node using the parent pointers saved throughout the search yields the optimum path. UCS has the advantage of always finding the optimal path, i.e., the path with the lowest cost, if one exists. However, if the cost of the best path is high or there are many nodes in the graph, the technique may be slow. Furthermore, UCS may not be ideal for graphs with negative edge weights because it may become trapped in a cycle of diminishing costs.

# Informed

Because they are more useful, informed search algorithms are the most used search algorithms. This is because when searching through nodes, the search algorithms take heuristic data into account. The types of data required are determined by the search requirements of the data set. A 2D maze structure would, at a bare minimum, require basic distance estimates between a node and the goal in this assignment, allowing the search algorithm to make 'smarter' decisions.

This estimate is usually obtained using one of three methods: Manhattan, Diagonal, or Euclidean. Manhattan is specifically designed for four-directional movement and provides heuristic values based on horizontal and vertical distances added together. Diagonal is intended for eight directions and returns a heuristic value based solely on the greatest difference between horizontal and vertical distances. Euclidean is designed to provide a heuristic value based on Pythagoras' theorem using horizontal and vertical distances for every angle of movement on a 2D plane.

The Manhattan method seemed most useful in this implementation as the movements were limited to horizontal and vertical.

## Greedy Best first Search (GBFS)

GBFS is a search algorithm that uses heuristics to estimate which node to visit next. We do not guarantee an optimal solution, but we often provide a good solution within a reasonable time. The heuristic function h(n) estimates the cost of the cheapest path from node n to a destination node. The algorithm maintains a priority queue of nodes to visit and selects the node with the lowest heuristic function value for the next expansion.

One of the great advantages of GBFS is speed, especially in large or complex search spaces. A well-designed heuristic function allows the algorithm to quickly converge to a solution even if it is not optimal. GBFS can also work well for online searches where the search space changes dynamically, and the goal is to find a solution as quickly as possible.

However, GBFS also has some limitations. First, GBFS algorithms tend to get stuck in local minimum regions or plateaus that cannot be explored beyond a certain point.

Second, GBFS can be sensitive to the quality of the heuristic function. If the heuristic function is too optimistic or too pessimistic, the algorithm may not find an optimal solution or come close to a solution at all. Finally, GBFS cannot handle negative edge weights because it can get stuck in an infinite loop. In summary, GBFS is a useful search algorithm that can provide fast problem solving in many cases, but its limitations should be considered when choosing the right algorithm for a given problem.

The algorithm works as follows.

1. Initialize the priority queue with the estimated distance from the start node to the destination.
2. While the queue is not empty:
3. Dequeue the node with the lowest estimated cost to the goal.
4. If the dequeued node is the goal node, return the solution.
5. Otherwise, expand the dequeued node by generating its successors and adding them to the priority queue with their estimated distance to the goal.

3. If the queue is empty and the goal node has not been found, return failure.

## A\* Search Algorithm

A\* (pronounced "A-star") is a popular and widely used search algorithm that combines the best features of equal-cost search and greedy, best-first search. A\* can find an optimal solution if the heuristic function used to estimate the residual cost for an object is acceptable (never overestimates the true cost) and consistent (satisfying the triangular inequality). The algorithm maintains a scaled priority queue of nodes, where the priority of each node is determined by a combination of the cost (g-value) of reaching that node from the source node and the expected cost of reaching the target from that node. (h value). The combination of these two values ​​is called the f-value. The heuristic function h(n) is used to estimate the remaining cost from node n to the destination. The cost g(n) is the cost of moving from the start node to node n. The value of f is the sum of the value of g and the value of h, or f(n) = g(n) h(n). Algorithm A\* selects the node with the smallest value f and then expands it. The A\* algorithm has several advantages over other search algorithms. First, the heuristic function is guaranteed to find the optimal solution if it is acceptable and consistent. Second, it is efficient in terms of time and space complexity, especially in large or complex search spaces. Third, it can handle problems with multiple objectives or constraints. However, A\* also has some limitations. First, A\* can be slow if the heuristic function is poorly designed or if the search space is too large. Second, the negative side of A\* may not be suitable for the weight problem because it may end up in an infinite loop. Third, A\* may be sensitive to the quality of the heuristic function. If the heuristic function is too optimistic or too pessimistic, the algorithm may not find an optimal solution or come close to a solution at all.

There are several versions of A\* that can be used to improve performance or solve certain types of problems. For example, the weight A\* can be used to balance optimality and speed by adjusting the weights of the heuristic function. Use real-time A\* targets or obstacles to manage dynamic environments that can change over time. Hierarchical A\* can be used to narrow the search space by dividing a problem into smaller subproblems.

In summary, A\* is a popular search algorithm that can efficiently find an optimal solution to a problem, provided the heuristic function is well designed and the search space is not too large. There are several variations of this algorithm that can be used to improve performance or solve certain types of problems.

The A\* algorithm works as follows:

1. Initialize the priority queue with the start node and its f-value.

2. While the queue is not empty:

a. Dequeue the node with the lowest f-value.

b. If the dequeued node is the goal node, return the solution.

c. Otherwise, expand the dequeued node by generating its successors and adding them to the priority queue with their f-values.

3. If the queue is empty and the goal node has not been found, return failure.

## Custom Algorithm 1 (diagonal A\*)

Custom 2 is essentially A\* with the addition of diagonal movement. Throughout my A\* research, it became clear to me that most implementations on a 2D plane showing A\* incorporates diagonal movement, so I thought it would be a good experiment to see how much of a difference it made. In essence, Custom 2 is an identical implementation of A\*, with the exception that it also checks diagonal movement and calculates its heuristics using Diagonal Distance rather than Manhattan Distance. This will be covered in the research section of this report.

# Implementation

My program is implemented in c#. The first step in my program is to parse in the given text file that contains the data set. The “world” is generated using the grid size, start position, goal position(s), and wall positions. The start position is represented by a 2d string Array of multi variable size, in which the start position is represented as “S”, Goal position(s) is/are represented by “G” and Walls are represented by “W”. Empty spaces are denoted by a white space.

The search algorithms are all encapsulated in its own Class namely, “astar”, “bfs”, “dfs’, “diag", “gbfs”, and “ucs”. Each search algorithm starts by calling the “Search” method. This is the main method in every algorithm. It contains all the function calls that implements the search. The “Search” algorithm is given the world map as a string array, so that it can search. It creates a list of outcomes that it returns as a string at the end of the “Search” method.

The start position and goal position(s) are checked to see if they exist. If either is not present, then the algorithm skips to the conclusion that no solution can be found.

# Uninformed

## Depth First Search

The DFS program starts by defining a few variables,

* Map: This is a 2d array that represents the map the algorithm is searching
* Nodes: This is a counter that keeps track of the number of nodes that have been visited
* StartGoalExists: This is a bool variable that indicates whether both start and goal exists.
* goalReached: This is a bool that indicates whether the goal has been reached.
* Directions: This is a list that stores the directions that the algorithm has taken to reach the current position.

The next few lines of code find the start and goal positions of the map.

The “Search” function in DFS works as follows,

1. Select the first unvisited node in the notVisited list.
2. Mark the node as visited.
3. Add the node's neighbors to the adjacents list.
4. Remove the node from the notVisited list.
5. If the goal position has been reached, return the directions that were taken.
6. If there are any unvisited nodes in the adjacents list, go back to step 1.

## Breadth First Search

* The BFS algorithm starts by defining a few variables.
* map: This is a 2D array that represents the map that the algorithm is searching.
* nodes: This is a counter that keeps track of the number of nodes that have been visited.
* Initialize\_Valid: This is a boolean variable that indicates whether both the start and goal positions have been found.
* goalReached: This is a boolean variable that indicates whether the goal position has been reached.
* directions: This is a list that stores the directions that the algorithm has taken to reach the current position.

The Search () method is the main function of the algorithm. It takes a 2D array as input and returns a string that contains the directions that the algorithm took to reach the goal position. The algorithm works by repeatedly doing the following:

1. Add the start position to the notVisited list.
2. While the notVisited list is not empty:
   1. Remove the first position from the notVisited list and set it as the current position.
   2. Mark the current position as visited.
   3. Add the current position's neighbors to the adjacents list.
   4. If the current position is the goal position, return the directions that were taken.
   5. Add the adjacents to the notVisited list.

## Uniform Cost Search

The cost of a node is the sum of the distance from the start node to the current node and the distance from the current node to the goal node.

The code starts by defining a few variables:

* map: This is a 2D array that represents the map that the algorithm is searching.
* nodes: This is a counter that keeps track of the number of nodes that have been visited.
* Initialize\_Valid: This is a boolean variable that indicates whether both the start and goal positions have been found.
* goalReached: This is a boolean variable that indicates whether the goal position has been reached.
* directions: This is a list that stores the directions that the algorithm has taken to reach the current position.

The algorithm works by repeatedly doing the following:

1. Add the start position to the openSet list.
2. While the openSet list is not empty:
   * Remove the node with the lowest F value from the openSet list and set it as the current node.
   * Mark the current node as visited.
   * Add the current node's neighbors to the closedSet list.
   * If the current node is the goal node, return the directions that were taken.
   * For each neighbor of the current node that is not in the closedSet:
     + Calculate the F value for the neighbor.
     + Add the neighbor to the openSet list.

# Informed

## A\* Search algorithms

The code starts by defining a few variables:

* map: This is a 2D array that represents the map that the algorithm is searching.
* nodes: This is a counter that keeps track of the number of nodes that have been visited.
* Initialize\_Valid: This is a boolean variable that indicates whether both the start and goal positions have been found.
* goalReached: This is a boolean variable that indicates whether the goal position has been reached.
* directions: This is a list that stores the directions that the algorithm has taken to reach the current position.

The next few lines of code find the start and goal positions in the map. The start position is marked with an "S" and the goal position is marked with a "G".

The Search () method is the main function of the algorithm. It takes a 2D array as input and returns a string that contains the directions that the algorithm took to reach the goal position.

The algorithm works by repeatedly doing the following:

1. Add the start position to the openSet list.
2. While the openSet list is not empty:
   * Remove the node with the lowest F value from the openSet list and set it as the current node.
   * Mark the current node as visited.
   * Add the current node's neighbors to the closedSet list.
   * If the current node is the goal node, return the directions that were taken.
   * For each neighbor of the current node that is not in the closedSet:
     + Calculate the F value for the neighbor.
     + Add the neighbor to the openSet list.

## Greedy Best First Search

The estimated cost of a node is the sum of the distance from the start node to the current node and the estimated distance from the current node to the goal node.

The code starts by defining a few variables:

* map: This is a 2D array that represents the map that the algorithm is searching.
* nodes: This is a counter that keeps track of the number of nodes that have been visited.
* Initialize\_Valid: This is a boolean variable that indicates whether both the start and goal positions have been found.
* goalReached: This is a boolean variable that indicates whether the goal position has been reached.
* directions: This is a list that stores the directions that the algorithm has taken to reach the current position.

The algorithm works by repeatedly doing the following:

1. Add the start position to the openSet list.
2. While the openSet list is not empty:
   * Remove the node with the lowest F value from the openSet list and set it as the current node.
   * Mark the current node as visited.
   * Add the current node's neighbors to the closedSet list.
   * If the current node is the goal node, return the directions that were taken.
   * For each neighbor of the current node that is not in the closedSet:
     + Calculate the F value for the neighbor.
     + Add the neighbor to the openSet list.

# Features/Bugs/Missing

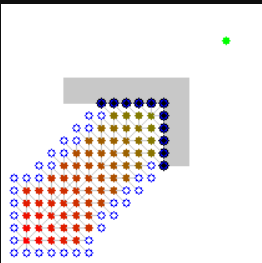
In terms of bugs sometimes the Console will overlay the last output. In terms of missing, I made sure to abide by the guidelines provided. The features necessary are all present to the best of my knowledge.

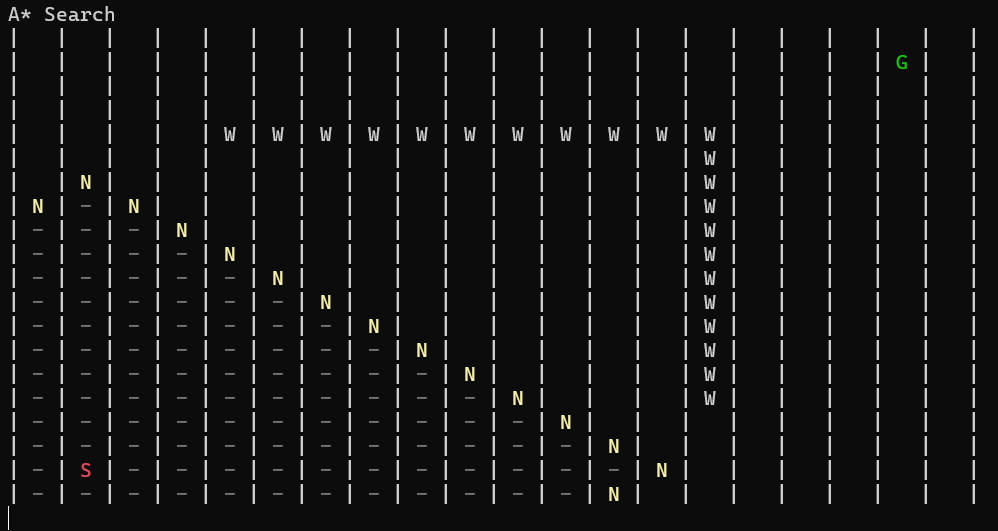
Research

Diagonal movement and the diagonal distance calculation

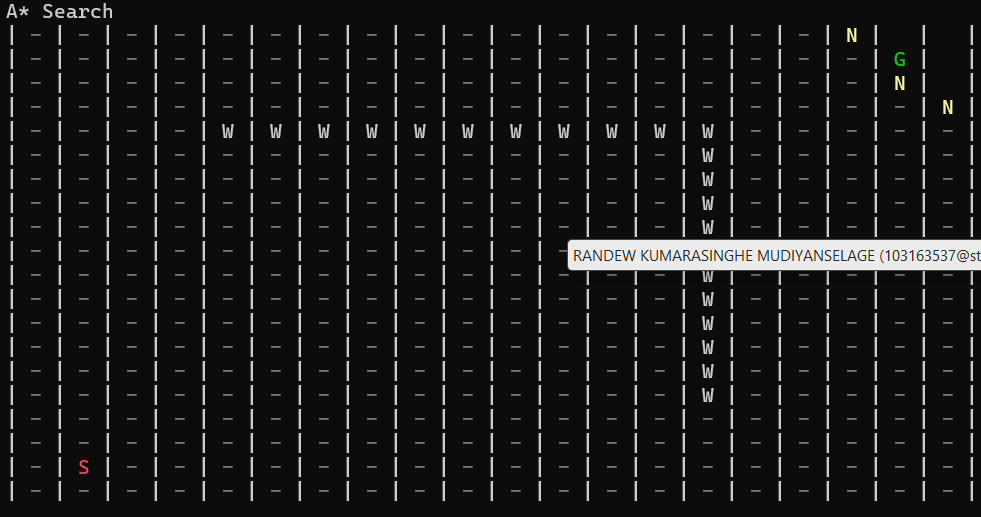
The custom 1 program is a clone of the A\* search, but with 8 directions of movement as compared to 4. This leads to lesser number of nodes being viewed. The manhattan distance formula was insufficient to calculate the diagonal distance so it was changed to a diagonal distance formula, which is what it was theoretically designed for.

For reference and to test performance of each algorithm I used the wikipedia entry map present in their A\* search page.

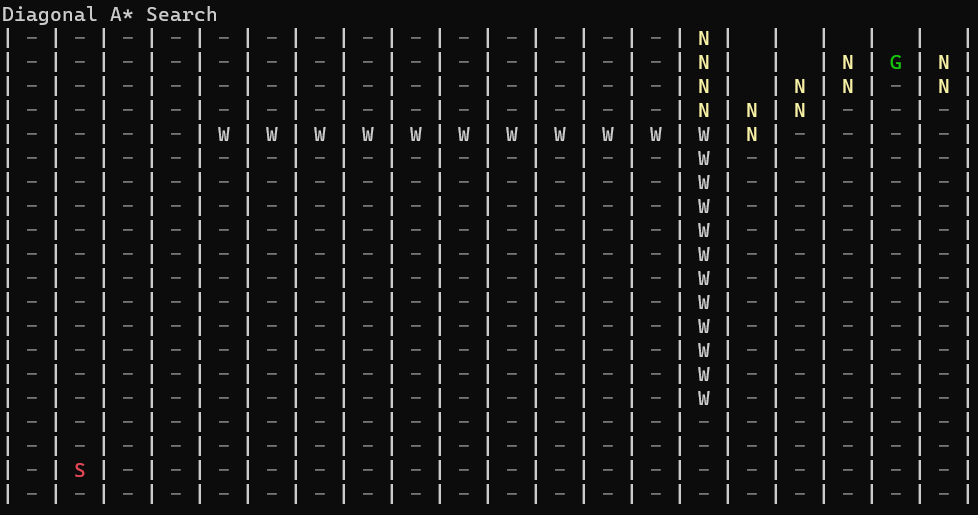




I wanted to see the difference in performance gained or lost through the diagonal motion versus horizontal/vertical motion.



A\* search algorithm took 21102 milliseconds, opened 370 nodes to reach the goal.



Diagonal A\* took 201416 milliseconds, opened 357 nodes to reach the goal in the wikipedia stock test.

The reason the animation has a long run time is since I have slowed down the drawing of the cells to allow it to be visible by the human eye. Due to that fact the 956-millisecond difference means a lot. Therefore, in conclusion the A\* search algorithm is being bottlenecked by its range of motion.

I did several tests on the algorithms to gauge which was the fastest overall. The maps were varied to not be biased to one specific algorithm.

Tests 1 to 4 was for gauging performance on different maps.

Test 1



This was conducted to measure straight line depth.

Test 2



The standard data set provided.

Test 3



This test was conducted to see how the algorithms would react to a choice between longer and shorter path (map was taken from a lecture)

Test4



This test was conducted to see how the algorithms would react with multiple goals and more obstructions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | DFS | BFS | A\* | GBFS | Diagonal | UCS |
| Test1 | 1182ms  38 nodes | 1120ms  36 nodes | 1132ms  36 nodes | 499 ms  16 nodes | 1088ms  34 nodes | 1085ms  35 nodes |
| Test2 | 718ms  25 nodes | 862 ms  28 nodes | 870 ms  28 nodes | 372 ms  12 nodes | 658ms  21 nodes | 840 ms  27 nodes |
| Test3 | 430 ms  14 nodes | 811 ms  26 nodes | 806 ms  26 nodes | 840 ms  27 nodes | 682 ms  22 nodes | 867 ms  28 nodes |
| Test4 | 964 ms  31 nodes | 961 ms  31 nodes | 958ms  31 nodes | 715 ms  23 nodes | 531 ms  17 nodes | 962 ms  31 nodes |

There are several conclusions that can be drawn from the results.

1. When the goal is positioned linearly GBFS is on average the fastest.
2. Diagonal motion benefits the A\* algorithm a lot as seen by every result being faster than the traditional
3. Depth first algorithm is useful when the goal is near to the starting position.