**SolvingMazeby Searching**

**[EXTRA] Overall Program Information (PLEASE READ):**

Our program contains **SIX** search algorithms, **TWO** uninformed and **FOUR** informed:

1. Breadth First Search
2. Depth First Search
3. A\* Search
4. Greedy Best First Search
5. Hill Climb Search
6. Dijkstra’s Search Algorithm

This program also contains a **GUI** that displays mazes along with **SAVE** and **LOAD** features for different maze tile-maps. We also have a **maze editor tool** and can run the mazes in **real-time** to see how they traverse through the maze.

1. **Team Members and Time Report:**

|  |  |  |  |
| --- | --- | --- | --- |
| **First Name** | **Last Name** | **Total Time** | **Contr Main Contributions** |
| **Levon** | **Swenson** | **16 hours** | **BFS, DFS, Hill Climb** |
| **Ryan** | **DeBusk** | **16 hours** | **A\* Search, Greedy Best First Search** |
| **Matthew** | **Burgess** | **16 hours** | **Dijkstra** |

1. **Problem Description:**

For our project, we wanted to answer a simple question; which search algorithm is best for solving randomly generated mazes? Using a custom random maze generator, we ran different types of search algorithms against one another to collect information such as: processing time, path length, and total nodes visited. Using these metrics, we can better estimate on average which searching algorithm works best for mazes given different imposing situations.

1. **Problem Modeling:**

Initial state:

- Start X-Location, Start Y-Location

Actions and transition Models (operators):

- Move Up (x = x, y = y - 1)

- Move Right (x = x + 1, y = y)

- Move Left (x = x - 1, y = y)

- Move Down (x = x, y = y + 1)

Goal state test:

- X-Location = Goal X-Location

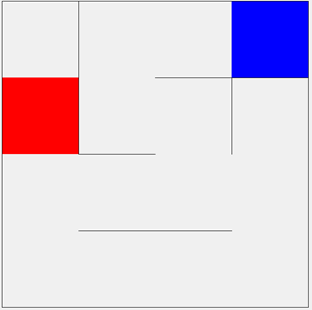
- Y-Location = Goal Y-Location

Path cost function:

- 1 cost per action executed

Example:

- 4 x 4 Maze (Top Left = 0,0)



Initial State:

- 3,0

Action:

- Move Up (x = x, y = y - 1)

- Move Right (x = x + 1, y = y)

- Move Left (x = x - 1, y = y)

- Move Down (x = x, y = y + 1)

Goal:

- 0,1

Cost:

- 1 cost per action executed

1. **Implementation**

The programing language we used was C#. The graphical framework we used was windows presentation foundation (WPF). C# was chosen because most of our group had previous experience using it. Both WPF and C# are integrated into visual studio which allowed for easy creation of an app with a graphical user interface. The data structure used to represent the state space was a two dimensional array of nodes. Each of these nodes has a reference to neighboring nodes if there is no “wall” in the way.

1. **Uninformed Search Algorithm: *Breadth First Search (Level Search)***
   1. **Algorithm Description**

This search starts at a root node and puts its children into a queue. The first thing from the queue is removed. This node is checked and its children are put into the queue. This search continues until the end goal is found or the queue is empty.

* 1. **Algorithm (pseudo code)**

BFS(M,S,E) // M is map, S is start node, E is end node

Let Q be a queue

Q.enqueue(S)

While(Q is not empty)

N = Q.dequeue()

set N to visited

if(N is E)

Create Path()

return

else

foreach (neighbor node in N)

if (node is not visited)

Q.enqueue(node)

set node to visited

* 1. **Algorithm Properties**

**·**  Complexity:

o Time Complexity: O(N) where N is the number of Nodes in the map

o Space Complexity: O(N) where N is the number on Nodes in the map

· Completeness: It will find the goal state if one exists.

· Admissibility: Since the cost for moving from one tile to another is uniformly 1, it will find a best solution.

· Irrevocability: Tentative

* 1. **Results**

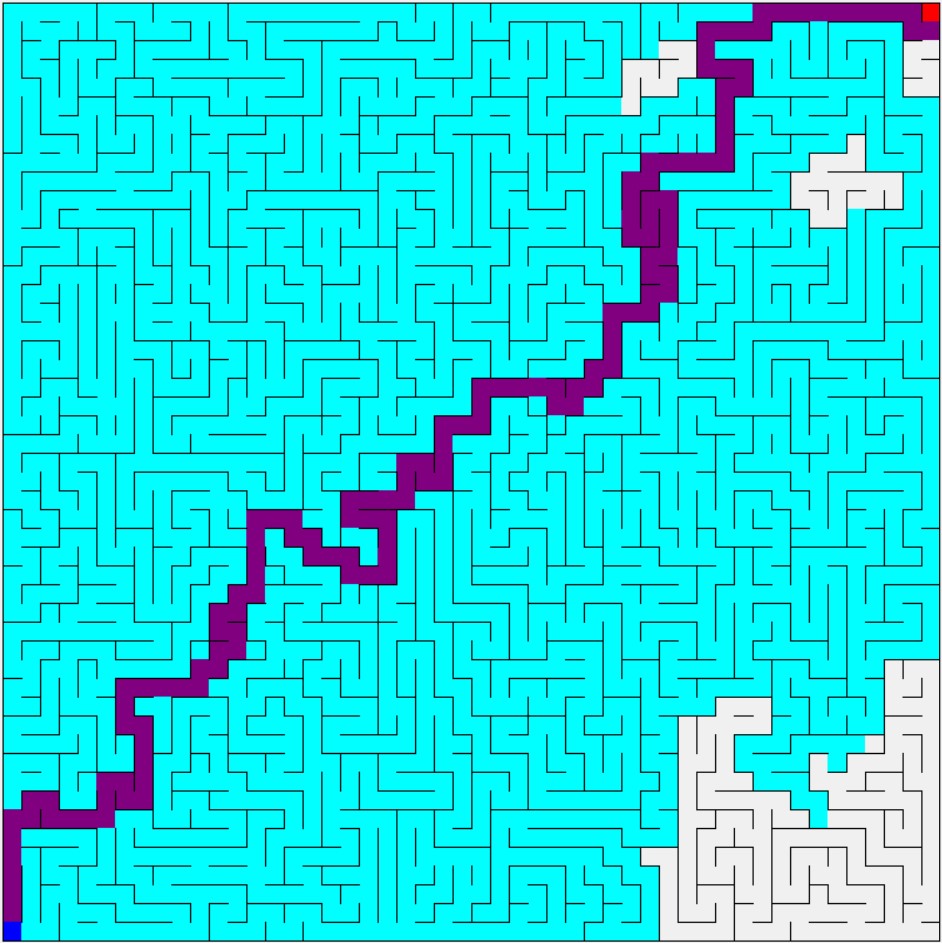


Fig. 1: 50 by 50 breadth first search (2,304 nodes visited in less than 1 ms; 136 node path length)

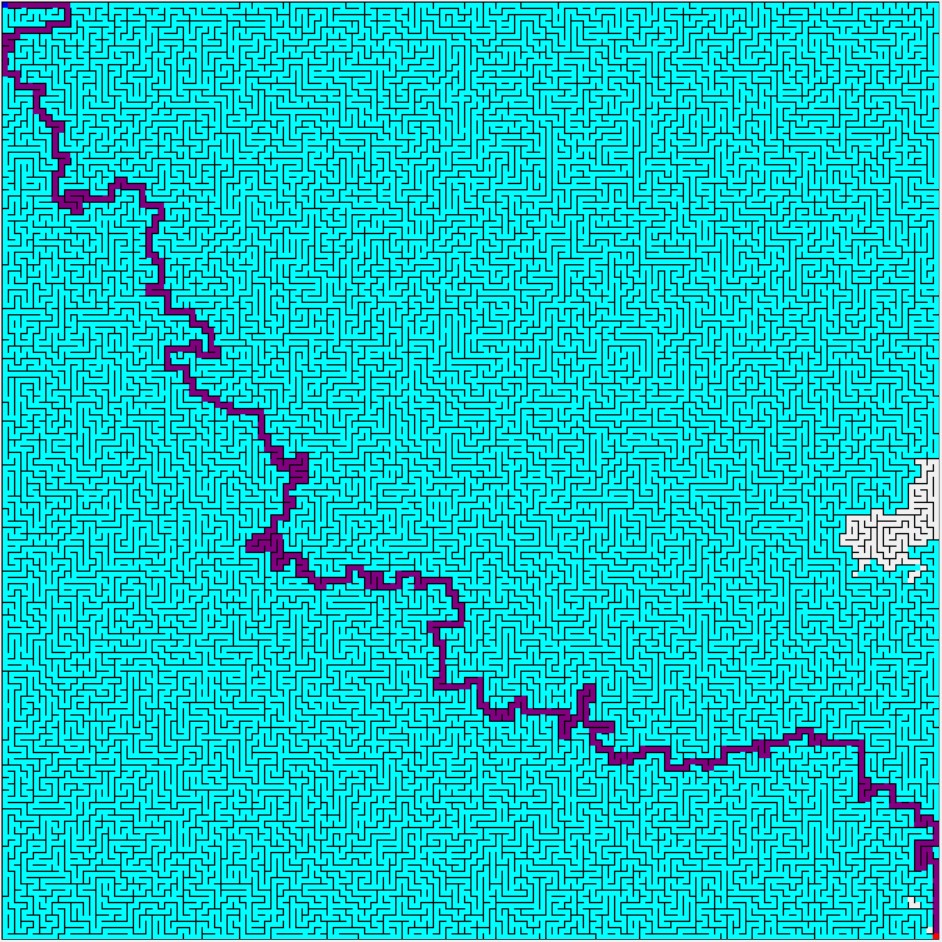


Fig. 2: 150 by 150 breadth first search (22,343 nodes visited in 2.1 ms; 526 node path length)



Fig. 3: 300 by 300 breadth first search (89,829 nodes visited in 11.7 seconds; 1,090 node path length)

1. **Uninformed Search Algorithm 2: *Depth First Search***
   1. **Algorithm Description**

This search starts at a root node. A child branch is explored down until at can’t do so anymore. The search then backtracks until there is a child branch unexplored. It then explores as far as possible down theat branch. It repeats these steps until the goal node is found or the entire maze is searched.

* 1. **Algorithm (pseudo code)**

DFS(M,S,E) // M is map, S is start node, E is end node

Let St be a stack

St.push(S)

While(St is not empty)

N = St.Pop()

set N to visited

if(N is E)

Create Path()

return

else

foreach (neighbor node in N)

if (node is not visited)

St.Push(node)

set node to visited

* 1. **Algorithm Properties**

· Complexity:

o Time Complexity: O(N) where N is the number of Nodes in the map

o Space Complexity: O(N) where N is the number on Nodes in the map

· Completeness: Since the maze doesn’t have any infinite length paths, It will find the goal node if one exists.

· Admissibility: It will Most likely not find the best solution, so it is not admissible.

· Irrevocability: Tentative

* 1. **Results**

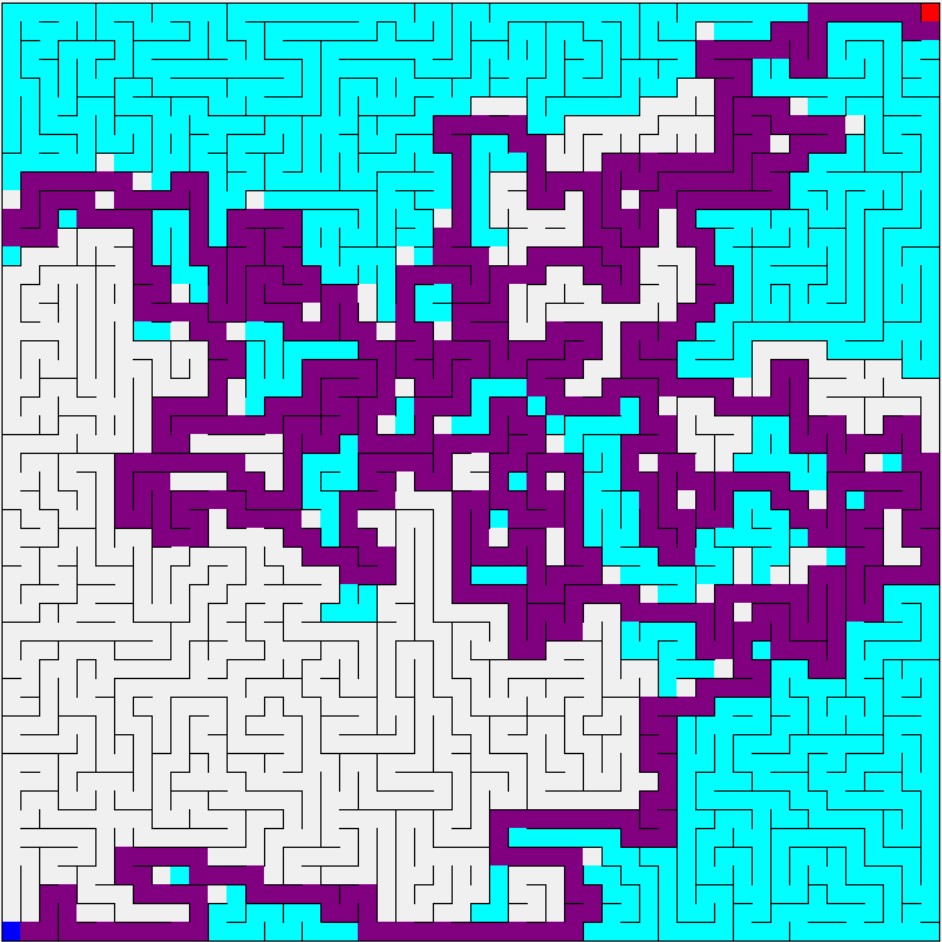


Fig. 4: 50 by 50 depth first search (1,616 nodes searched in less than 1 ms; 690 node path length)

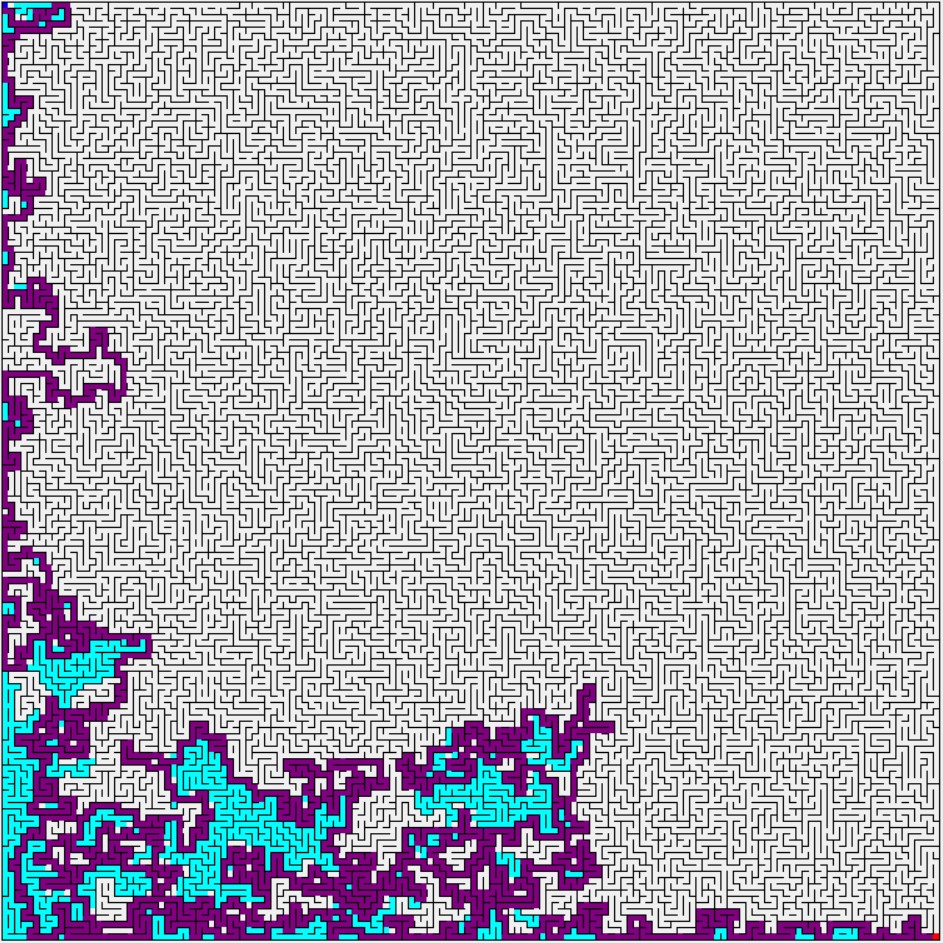


Fig. 5: 150 by 150 depth first search (3,107 nodes searched in less than 1 ms; 2,042 node path length)

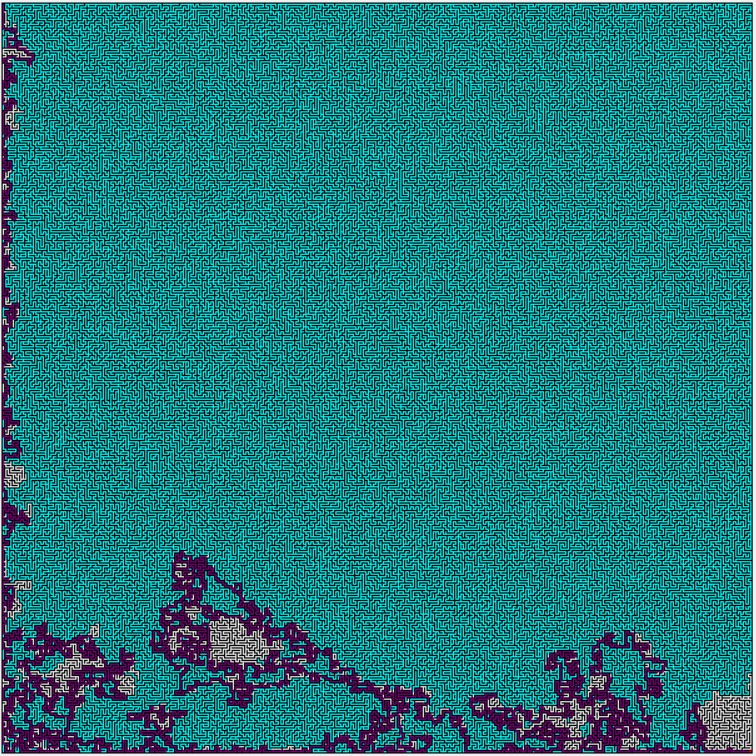


Fig. 6: 300 by 300 depth first search (87,241 nodes searched in 9.9 ms; 4,856 node path length)

1. **Heuristic Search Algorithm 1: *A\* Search*** 
   1. **Heuristic Description**

A\* uses two pieces of data to calculate its heuristic, the distance from the goal node and the current amount of nodes traversed. Using these two pieces of data, it takes the sum of them and compares that sum to every other node, choosing the one with the least sum as its next node to traverse.

***f(n) = g(n) + h(n) where g(n) is distance traveled and h(n) is distance to end node***

* 1. **Algorithm (pseudo code)**

**Generate heuristic values based off distance to endNode**

**Add starting node to openNodes list**

**while openNodes.Count != 0**

**currentNode is first element in openNodes**

**foreach valid neighbor of currentNode**

**if currentNode is endNode**

**visited = true**

**set parent to currentNode**

**return**

**else**

**set parent to currentNode**

**if currentNode is not in openNodes**

**set traverse distance to current plus parent**

**add to openNodes list**

**visited = true**

**remove currentNode from openNodes**

**sort openNodes list by shortest heuristic value (f(n) = g(n) + h(n))**

**return**

* 1. **Algorithm Properties**

This algorithm almost always searches an optimal path while keeping its amount of nodes visited reduced versus uninformed searches.

Time Complexity: O(NLog(N)) where N is number of nodes

Space Complexity: O(N)

Completeness: This algorithm is complete as it will check all nodes starting from lowest heuristic to highest

Admissibility: It will almost always choose the most optimal path, sacrificing time to find it.

Irrevocability: This is *tentative* as it does backtrack and revisit nodes

* 1. **Results**

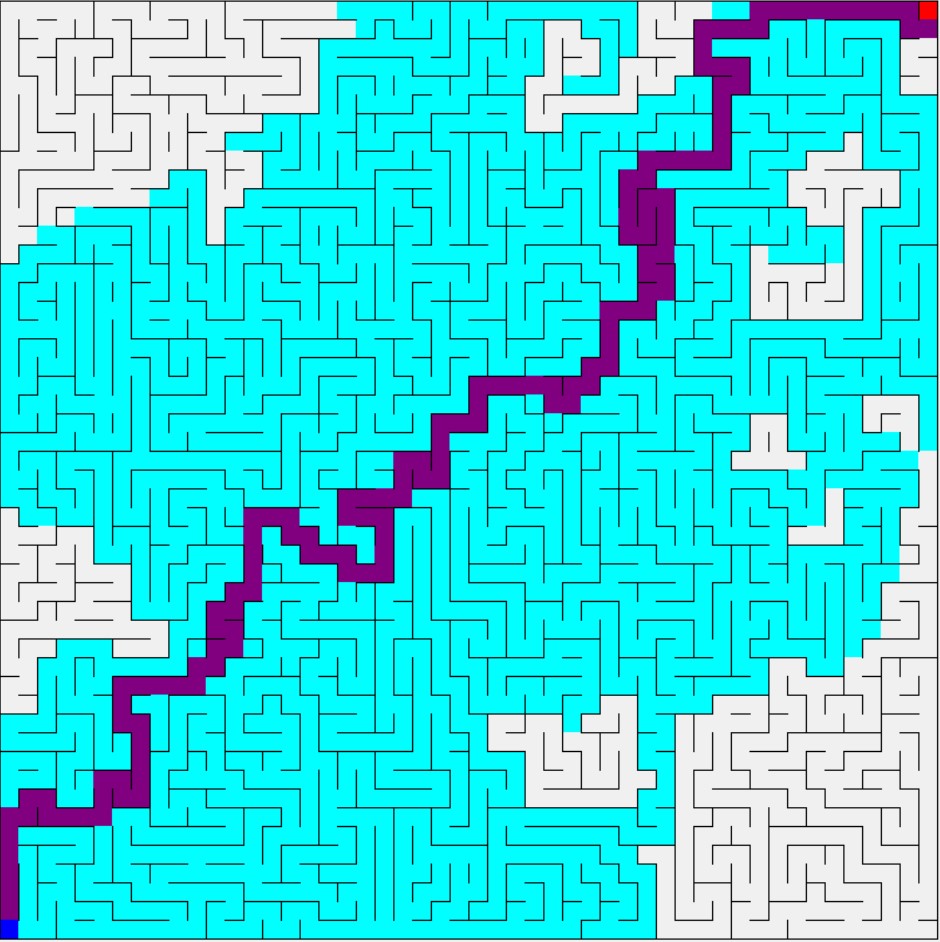
****

Fig. 7: 50 by 50 A\* search (1,908 nodes visited in 2.3 ms; 136 node path length)

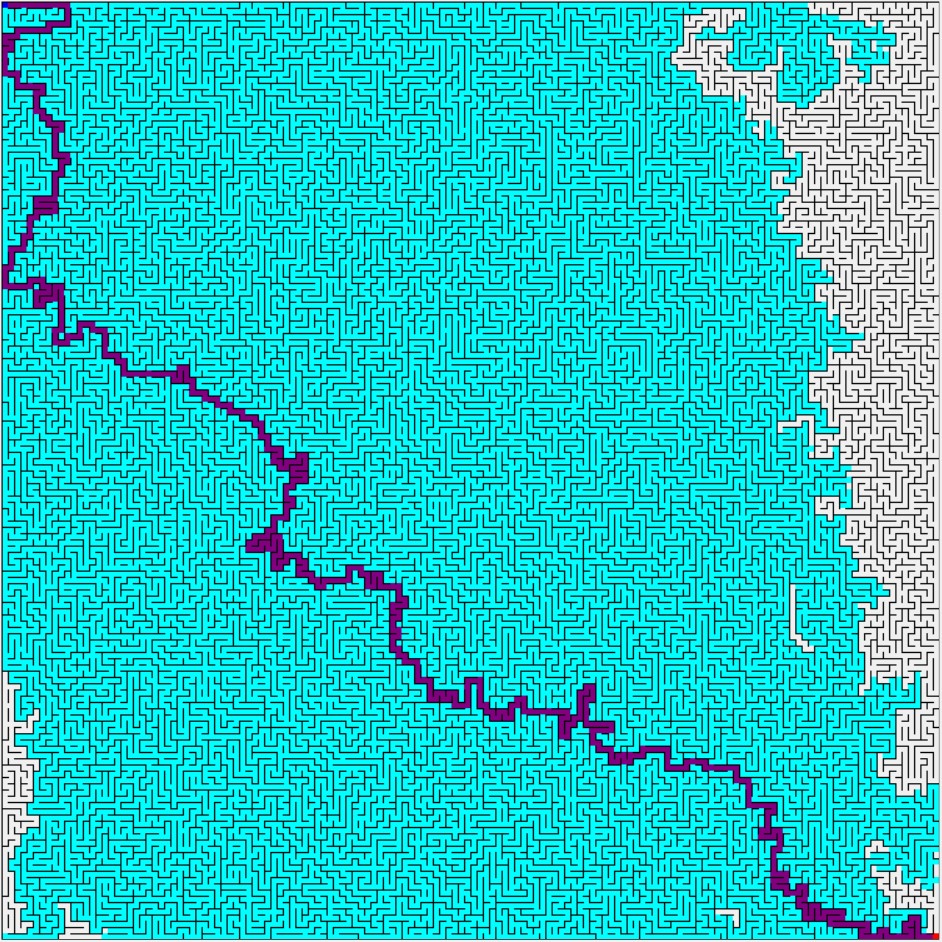
****

Fig. 8: 150 by 150 A\* search (20,048 nodes visited in 45.2 ms; 528 node path length)

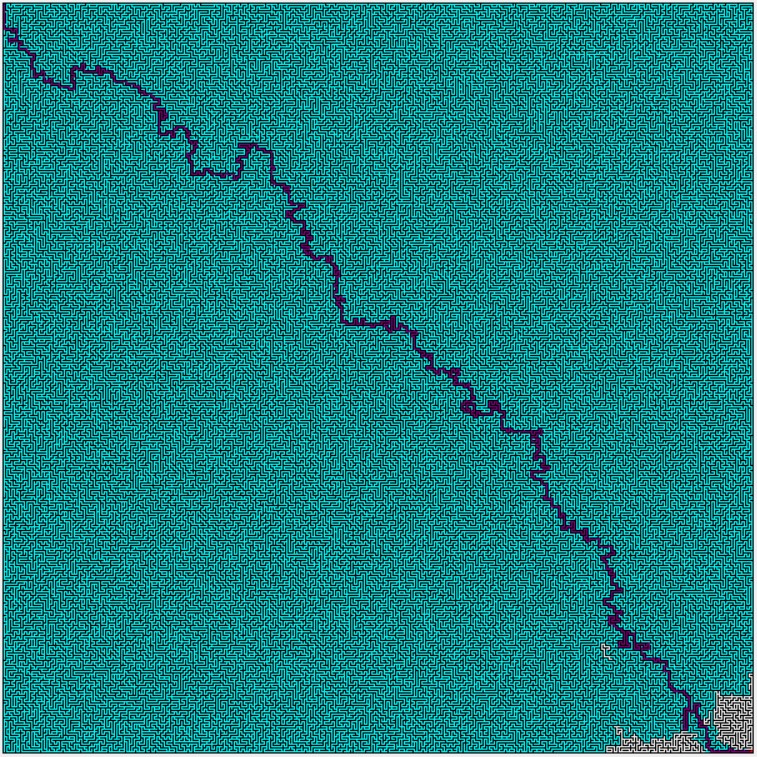
****

Fig. 9: 300 by 300 A\* search (89,393 nodes visited in 368.1 ms; 1,092 node path length)

1. **Heuristic Search Algorithm 2: *Greedy Best First Search***
   1. **Heuristic Description**

Greedy Best First Search uses one piece of data as its heuristic, the distance from the current node to the end node. This search algorithm will always choose its next node as the node which is closest distance-wise to the end node.

***f(n) = h(n) where h(n) is distance to end node***

* 1. **Algorithm (pseudo code)**

**Generate heuristic values based off distance to endNode**

**Add startingNode to openNodes list**

**while openNodes.Count != 0**

**currentNode is first element in openNodes**

**foreach valid neighbor of currentNode**

**if currentNode is endNode**

**visited = true**

**set parent to currentNode**

**return**

**else**

**set parent to currentNode**

**if currentNode is not in openNodes**

**add currentNode to openNodes with heuristic value**

**visited = true**

**remove currentNode from openNodes**

**sort openNodes list by shortest heuristic value (sorting by h value only, g is not used)**

**return**

* 1. **Algorithm Properties**

This algorithm will find the end goal very fast, but time is sacrificed for accuracy as it usually does not find the optimal path but instead usually slightly longer than the optimal path.

Time Complexity: O(NLog(N)) where N is number of nodes

Space Complexity: O(N)

Completeness: This search algorithm is complete as it will search from lowest heuristic value to highest, checking every one along the way until it finds the goal node.

Admissibility: This algorithm is not admissible as it sacrifices path optimizing for quickness of processing time.

Irrevocability: This is tentative as it does backtrack and revisit nodes

* 1. **Results**

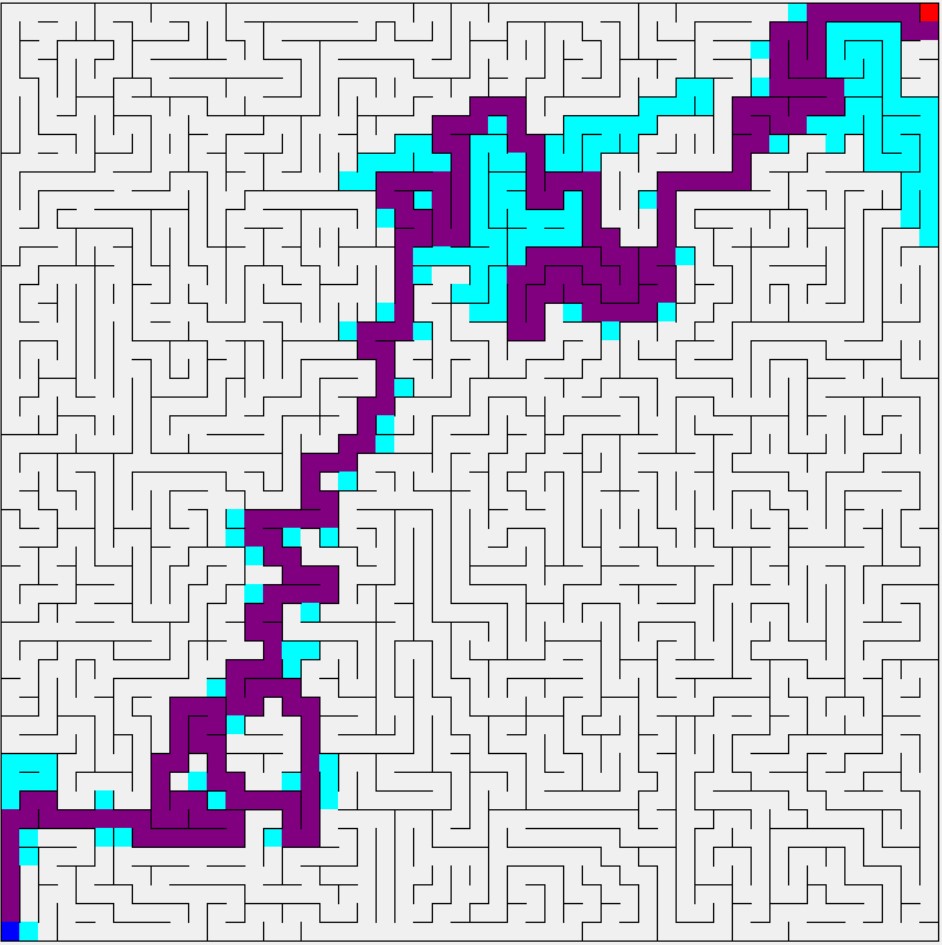


Fig. 10: 50 by 50 greedy best first search (389 nodes visited in less than 1 ms; 228 node path length)

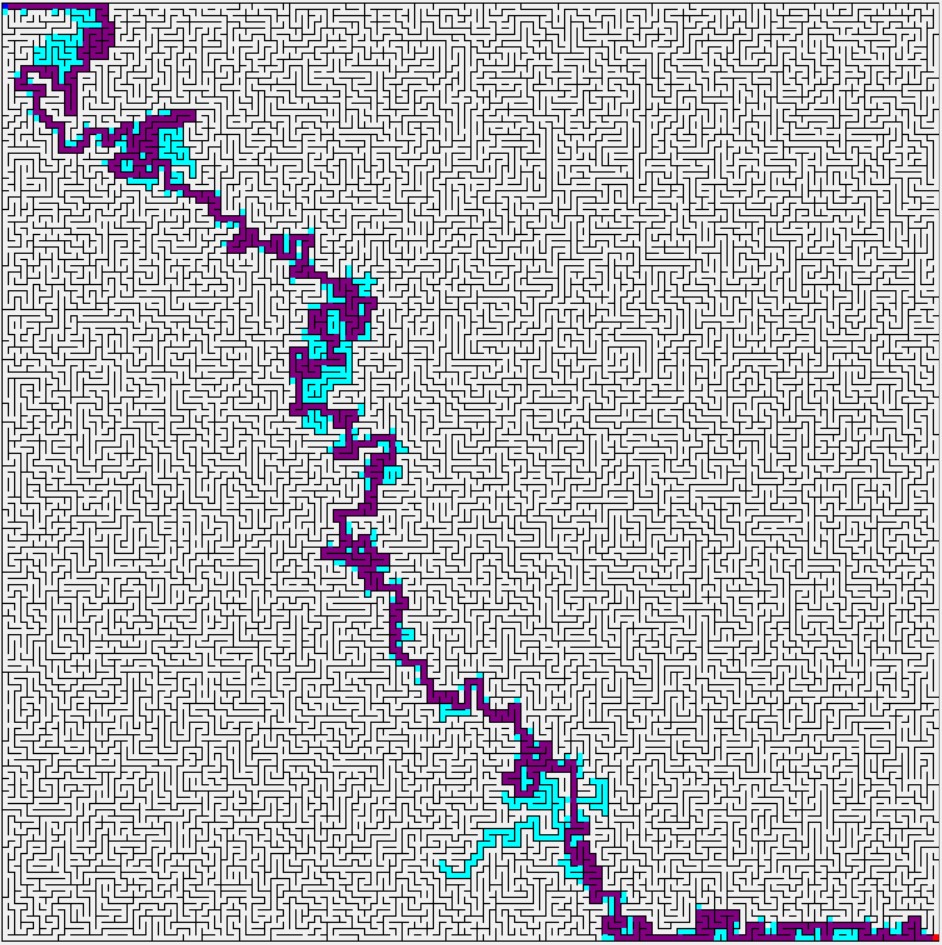


Fig. 11: 150 by 150 greedy best first search (1,261 nodes visited in 1.4 ms; 772 node path length)

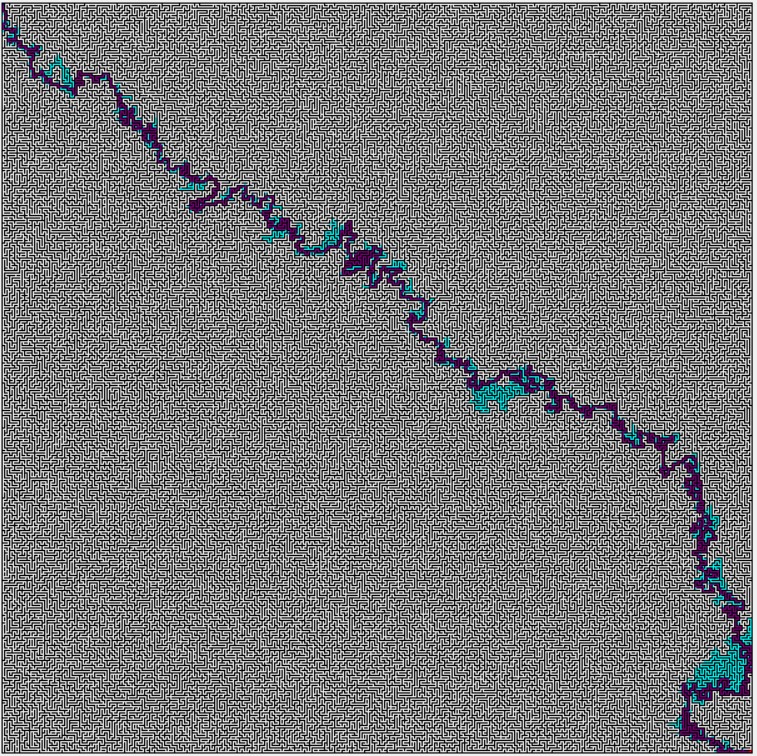


Fig. 12: 300 by 300 greedy best first search (3,042 nodes visited in 11.4 ms; 1,720 node path length)

1. **Heuristic Search Algorithm 3: *Hill Climb Search***
   1. **Algorithm Description**

This search starts at a root node. The children nodes are then searched. The child node with the best heuristic value is then searched. This continues until the goal node is found or a local maximum is hit.

* 1. **Heuristic Description**

**The heuristic that was used for this search is called the Manhattan Distance. It is simply the x distance from the current node to the goal node plus the y distance from the current node to the goal node. This was choose for this search because we didn’t move in any diagonal directions, therefore, we wanted the minimum cost to move from a node to the end node. The heuristic function and evaluation function are below respectively.**

**H(n) = |current node x value – goal node x value| + |current node y value – goal node y value|**

**F(n) = H(n)**

* 1. **Algorithm (pseudo code)**

**HillClimb(M,S,E) // M is map, S is start node, E is end node**

**Generate H values()**

**Set Current Node to S**

**while(Current Node is not null)**

**set Current Node to visited**

**if(Current Node is E)**

**Create Path()**

**return**

**else**

**set Next Node to null**

**foreach(neighbor node in Current Node)**

**if(neighbor node's h < current lowest h of neighbors)**

**set Next Node to neighbor node**

**set Current Node to Next Node**

* 1. **Algorithm Properties**

· Complexity:

o Time Complexity: O(∞) since it is not complete

o Space Complexity: O(n) where n is number of local edges

· Completeness: It is not guaranteed to find a goal state if one exists.

· Admissibility: If it finds a solution the solution may not be the best solution

· Irrevocability: Yes

* 1. **Results**

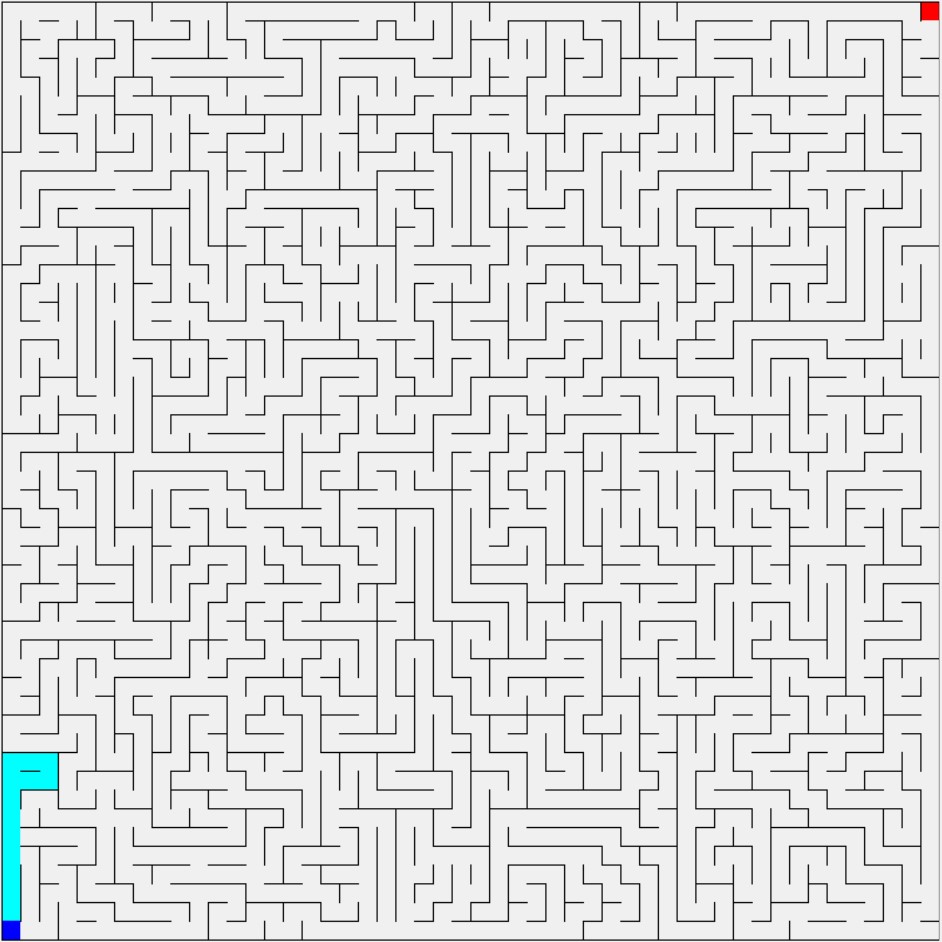


Fig. 13: 50 by 50 hill climb search (14 nodes searched in less than 1 ms; finish point not found)

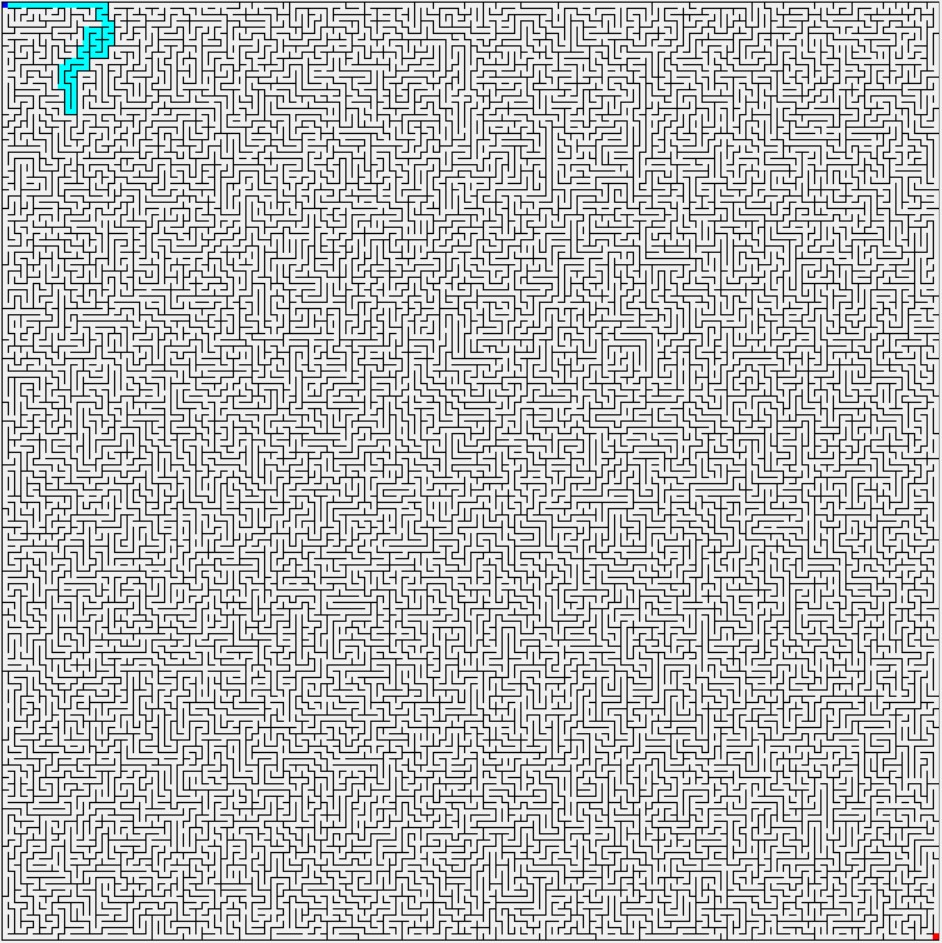


Fig. 14: 150 by 150 hill climb search (74 nodes searched in less than 1 ms; finish point not found)

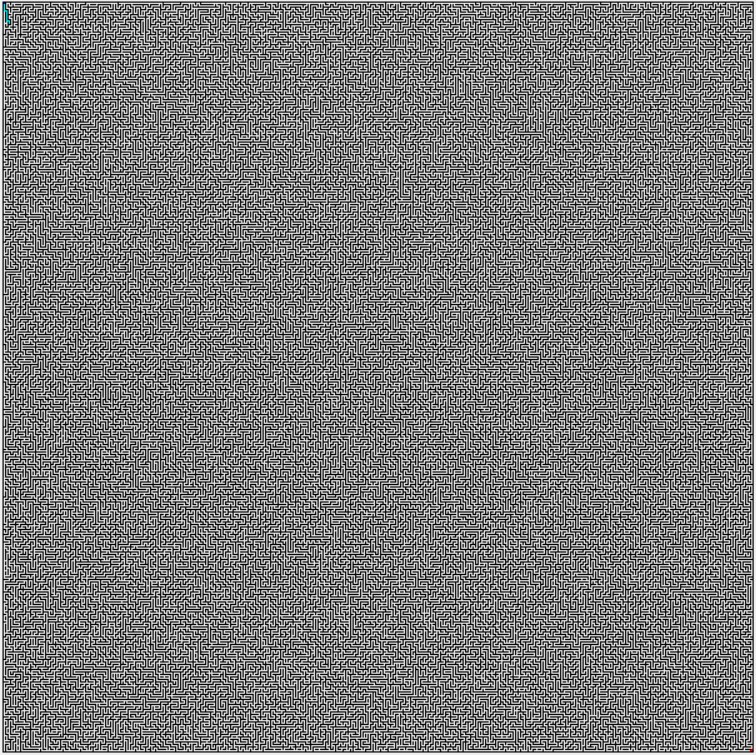


Fig. 15: 300 by 300 hill climb search (11 nodes searched in less than 1 ms; finish point not found)

1. **Heuristic Search Algorithm 4: *Dijkstra’s Algorithm***
   1. **Algorithm Description**

Dijkstra’s search algorithm is an informed search algorithm that is guaranteed to find the shortest path from a start point to an end point provided such a path exists. Although the algorithm doesn’t use a heuristic value to predict the distance from a given location to the end point, the algorithm does maintain the shortest distance between the start point and each visited location. If a shorter path is found from the start point to a given location, the distance from the start point to the location is updated accordingly. This, in conjunction with the fact that the location closest to the start point is always the next location visited, assures that when the end point is discovered, it will be found with the shortest possible path to the start point.

* 1. **Algorithm (pseudocode)**

let C be the current node, G be the goal node, and O be the list of open nodes

while G hasn't been visited and O isn't empty

C = the first element in O

for each node N in O

let distance\_N = the previously calculated distance from the start node to N

let distance\_C = the previously calculated distance from the start node to C

if distance\_N < distance\_C

C = N

for each neighbor node N of C

if N hasn't been visited

let distanceFromStart = the distance from the start node to C + 1

let distance\_N = the previously calculated distance from start node to N

if distanceFromStart < distance\_N || N hasn't been reached before

distance\_N = distanceFromStart

the parent of N = C

if N isn't in O

add N to O

remove C from O

mark C as visited

* 1. **Algorithm Properties**
* Complexity
  + Time Complexity: Assuming the worst-case scenario where all nodes are visited before the end point is found, Dijkstra’s algorithm has O(ElogV) for time complexity where E is the number of edges and V is the number of vertices. This is because the algorithm essentially contains two components: doing all work with the current node and picking the next node to visit. Due to the nature of the nature of our implementation, the general processing of a node will occur E + V times (each node is visited and evaluated, and the neighbors of each node are re-evaluated for distance) for a O(E+V). Searching for the next node has a time complexity of O(logV) simply because this is the Big-O for searching through a dictionary in C#. Combining these Big-Os together produces O[(E+V)logV] = O(ElogV + VlogV) = O(ElogV) for the algorithm as a whole.
  + Space Complexity: Assuming the worst-case scenario where all nodes are visited before the end point is found, Dijkstra’s algorithm will need to have data for each node in addition to the distance of each node from the start point. This results in O(V2), where V is the number of vertices.
* Completeness: If a solution exists, Dijkstra’s algorithm is guaranteed to find the solution
* Admissibility: Dijkstra’s algorithm doesn’t use a heuristic function. As a result, the heuristic value assigned to each node can be said to be 0. Therefore, Dijkstra’s algorithm can be said to be admissible because the heuristic value assigned to a node (0) will never be greater than the actual distance of the node from the end point.
* Irrevocability: Dijkstra’s algorithm never backtracks, so it does feature irrevocability
  1. **Results**

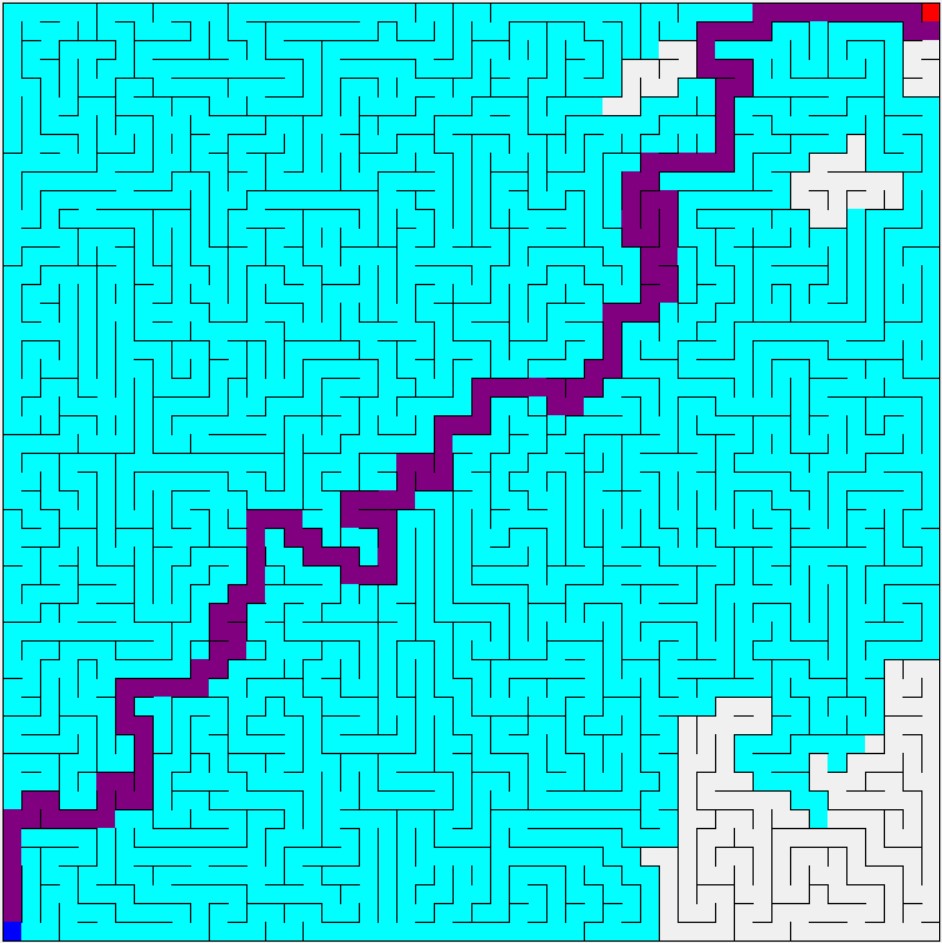
****

Fig. 16: 50 by 50 Dijkstra’s algorithm search (2,303 nodes visited in 4.4 ms; 136 node path length)

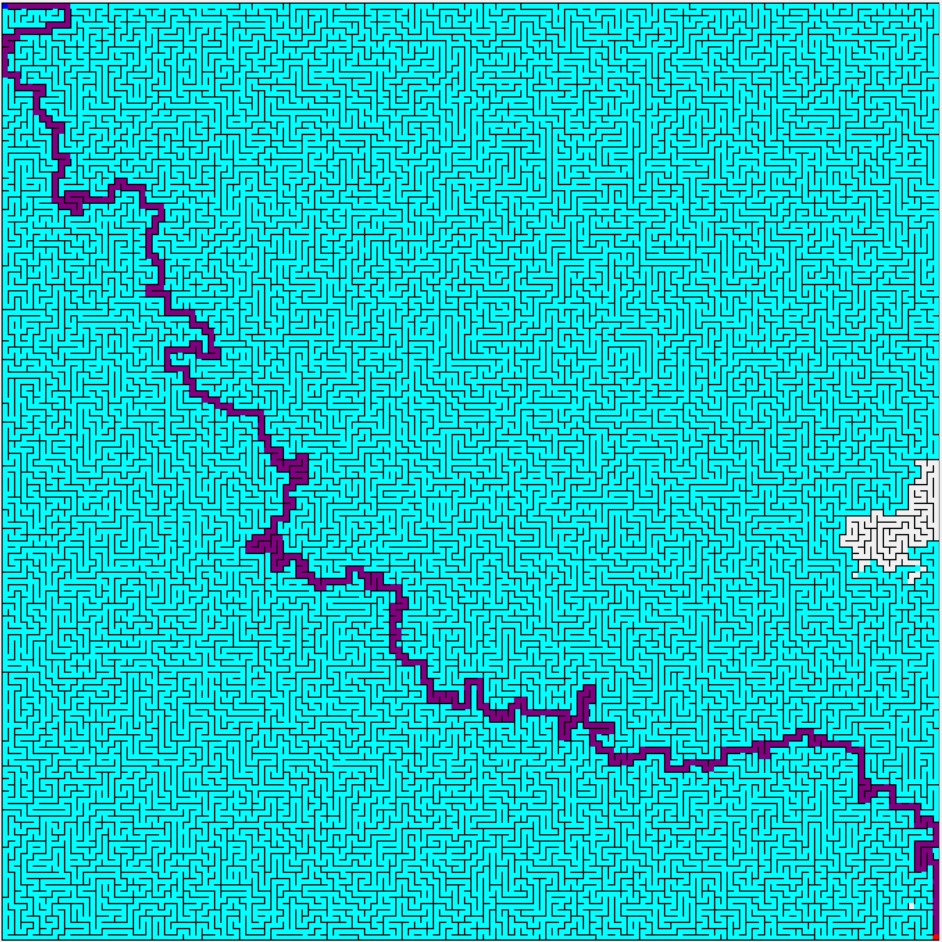
****

Fig. 17: 150 by 150 Dijkstra’s algorithm search (22,346 nodes searched in 69.8 ms; 526 node path length)

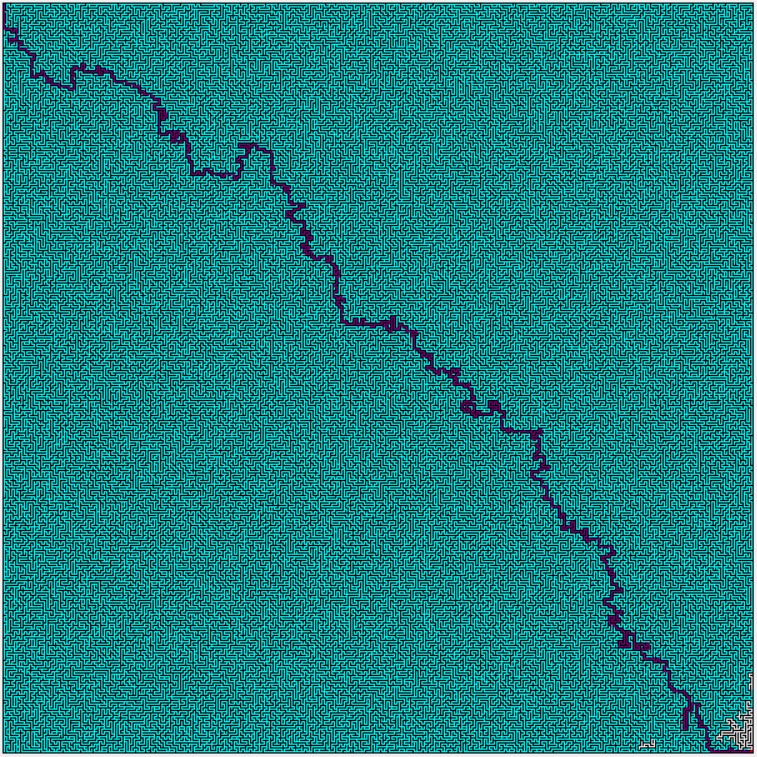
****

Fig. 18: 300 by 300 Dijkstra’s algorithm search (89,825 nodes searched in 467 ms; 1,090 node path length)

1. **Empirical Analysis**

Determining which of our six searches tested is the best overall depends on a few different characteristics: total elapsed time, number of nodes visited, and resulting path length. To explore this information, we picked three maze sizes, 50 by 50, 150 by 150, and 300 by 300, and generated three mazes for each of these sizes (a total of nine mazes). We then ran each search on each maze and recorded the data for each run (the data can be viewed in its entirety in Table 1).

In order to summarize this data into an easier to understand format, we took the averages of each of these characteristics for each search on the three mazes for a given dimension (for example, the 50 by 50 elapsed time values for the breadth first search on maze 1, maze 2, and maze 3 were averaged, and so on). These averages can be seen in Fig. 19, 20, and 21 below, which contain the averages for total nodes visited, path length, and elapsed time, respectively. It should be noted here that there were no mazes in our dataset where the hill climb search actually found the finish point in the maze. As a result, the total nodes visited and elapsed time for the search may appear to be low, but this is only because the search terminated early. Additionally, the path length for the hill climb search is 0 because no final path was ever produced. Because of this, although the hill climb search can certainly have its uses in other applications, it’s not appropriate for most mazes and will be ignored for the remainder of this analysis.

Regarding total nodes visited, it’s apparent that, although the various searches might not show a significant difference in smaller mazes, the greedy best first search is definitely the most efficient even in larger mazes. It should also be noted that breadth first, A\*, and Dijkstra’s all searched roughly the same number of nodes for each maze size, which makes sense given the fact that these three algorithms all generally follow the same rules.

For path length, a smaller maze, once again, doesn’t demonstrate a significant difference between the different searches, but as the maze increases in size, a significant difference arises particularly between depth first and all other searches tested. Additionally, breadth first, A\*, and Dijkstra’s all produce roughly the same short path length for each maze size, which makes sense considering that each of these algorithms are supposed to find the shortest path from the start to the finish. From Table 1, it can be seen that A\* typically produces a path length a few nodes longer than breadth first and Dijkstra’s. This is likely due to a slight implementation error that we were unable to resolve.

The average elapsed time for each search is once again non-differentiable for the smaller maze dimensions, but as the maze size increases, it’s clear that A\* and Dijkstra’s are the least efficient. Even at the larger maze sizes, however, the breadth first and greedy best first searches take very little time, and the depth first search takes almost no time at all. Interestingly, although breadth first, A\*, and Dijkstra’s all search roughly the same number of nodes, breadth first is far faster than A\* and Dijkstra’s. This is because of differences in implementation, namely, that breadth first searching uses a queue for determining the next node to search instead of having to search through each open node to check distance or heuristic values.

With all of this information in mind, we made the following determinations regarding what can be considered the best search algorithm for a maze. In the event that time is of the utmost importance, depth first is clearly the best choice. For a focus on path length, breadth first, A\*, and Dijkstra’s all produce equally short paths, so any of these could be used to find the best path. Although, if the other two variables are used as a tie-breaker in this case, the breadth first search is the better shortest-path search since it completes in a shorter amount of time. If space constraints are a problem, greedy best first searches through the fewest nodes, making it the best for this situation. In the end, however, after considering each of these three variables together, we determined that the best all-around search out of the six that we tested for a maze is the greedy best first search. Although using this search will produce a path length greater than that of the shortest path, both the required time and total nodes visited are very small, a combination that none of the other searches feature. Additionally, while the path length produced by the greedy best first search isn’t the absolute best path, it only exceeds the best path length by a relatively small amount.

Table 1: Data collected from the search algorithms being performed on nine different mazes (three mazes for each dimension)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 50 X 50 | | | 150 X 150 | | | 300 X 300 | | |
|  |  | Elapsed Time (ms) | # of Visited Nodes | Path Length | Elapsed Time (ms) | # of Visited Nodes | Path Length | Elapsed Time (ms) | # of Visited Nodes | Path Length |
| Breadth First | Maze 1 | 0 | 2304 | 136 | 2.1 | 22343 | 526 | 11.7 | 89829 | 1090 |
| Maze 2 | 0 | 2500 | 180 | 2.4 | 22376 | 510 | 11.2 | 89956 | 984 |
| Maze 3 | 0 | 2273 | 170 | 2.4 | 22489 | 508 | 11.9 | 89607 | 1002 |
| Depth First | Maze 1 | 0 | 1616 | 690 | 0 | 3107 | 2042 | 9.9 | 87241 | 4856 |
| Maze 2 | 0 | 1035 | 490 | 0.3 | 3278 | 1982 | 0.2 | 6486 | 4198 |
| Maze 3 | 0 | 373 | 350 | 0 | 2622 | 1750 | 0 | 6331 | 4098 |
| Hill Climb | Maze 1 | 0 | 14 | - | 0 | 74 | - | 0 | 11 | - |
| Maze 2 | 0 | 79 | - | 0 | 16 | - | 0 | 54 | - |
| Maze 3 | 0 | 11 | - | 0.2 | 85 | - | 0 | 48 | - |
| A\* | Maze 1 | 2.3 | 1908 | 136 | 45.2 | 20048 | 528 | 368.1 | 89393 | 1092 |
| Maze 2 | 2.3 | 2462 | 180 | 45.8 | 20768 | 512 | 347.1 | 82307 | 990 |
| Maze 3 | 1.4 | 1759 | 170 | 49.6 | 21677 | 510 | 358.2 | 86170 | 1002 |
| Greedy Best First | Maze 1 | 0 | 389 | 228 | 1.4 | 1261 | 772 | 11.4 | 3042 | 1720 |
| Maze 2 | 0 | 247 | 196 | 2.1 | 1271 | 656 | 6.1 | 2173 | 1472 |
| Maze 3 | 0 | 328 | 202 | 1 | 1079 | 688 | 6.6 | 2340 | 1498 |
| Dijkstra's | Maze 1 | 4.4 | 2303 | 136 | 69.8 | 22346 | 526 | 467 | 89825 | 1090 |
| Maze 2 | 5 | 2500 | 180 | 71.5 | 22374 | 510 | 474.8 | 89958 | 984 |
| Maze 3 | 4 | 2275 | 170 | 72.3 | 22489 | 508 | 480.8 | 89602 | 1002 |

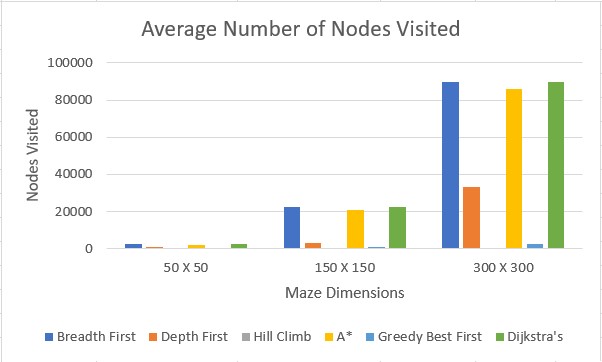


Fig. 19: A graph which compares the average number of nodes visited during each search for each dimension.

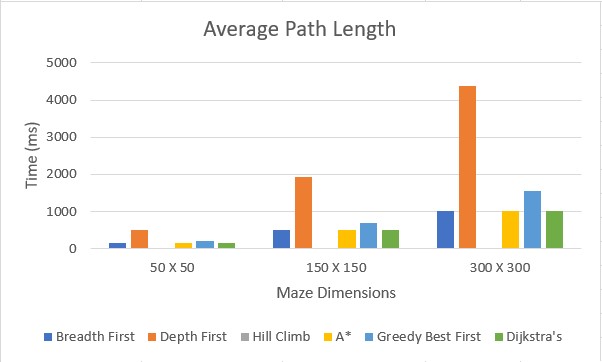


Fig. 20: A graph which compares the average length of the path generated during each search for each dimension (note that Hill Climb has an average of 0 for each dimension because the search never found the finish point).

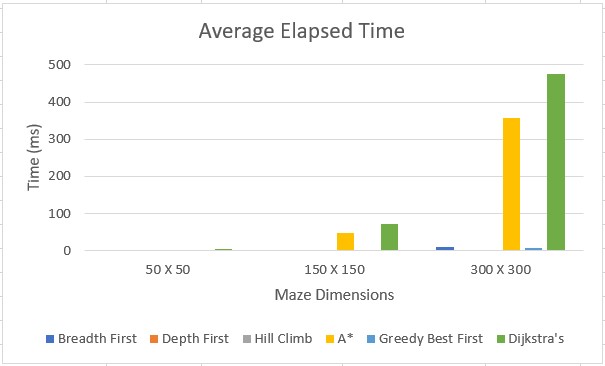


Fig. 21: A graph which compares the average time taken by each search for each dimension.

1. **Conclusion**

We created several searches to search through a maze that we randomly generated. While some of the searches are implemented simularily, some were very different. The biggest difference is between the informed and uninformed searches. The informed searches required creating a dictionary which stored the visited nodes with their evaluated value, so the dictionary could be looked through to find the next node to search. This leads to gaining benefits from the information, but it requires more time to process each node. This extra complexity in the informed searches also led to then taking longer to plan and implement than the uninformed searches. The best search heavily depends on the situation.

Due to the original implementation of most of our searches, multiple revisions were made to increase the speed. This led to large decreases in the time to complete the affected searches. More planning of the searches’ implementations before coding would have helped in reducing the amount of refactoring.

After analyzing our different searches we determined that each has its specific advantages. If the shortest path is needed, either Breadth First, Dijkstra's, or A\* with admissible heuristics should be used. With the path weight being uniform, Breadth First would be the best for this problem in relation to the shortest path. If space is the most important factor, Best First should be used because it visits the least amount of nodes. Lastly, if speed is most important, Depth First or Best First should be used. Taking these different factors into account, we determined that Best First is the best search for this particular problem. As a final note, Hill Climbing for this situation should be avoided or modified because it rarely if ever found the goal.