Solving Maze Using Different Algorithms

SayedMohammad Hashemi (110104372)
MohammadEhsan Akhavanpour (110081831)

Abstract

Maze is a historical structure. In ancient Egypt, they called it a labyrinth. It is a kind of puzzle, and in this project, we want to solve the grid-base version with a restriction of movement to 4 main directions. We use different algorithms and compare their performance to experiment with their advantages and disadvantages in solving mazes. The metrics Our chosen algorithms are Depth First Search, Breadth First Search, Greedy Best First Search, and A*. Our heuristics for informed algorithms are Chebyshev, Manhattan, and Euclidean distance, and our evaluation metrics are Completeness, Optimality, Time, and Space Complexity. The goal is to understand which algorithm can solve mazes better and in which conditions.

1 Introduction

Generally, a maze consists of a series of paths leading from one point to another. Reaching a destination from a given point, an agent can take as many unique paths as it wishes in a given maze. This problem is closely related to the shortest path in the graph by meeting some criteria (Shortest, Cheapest, Fastest). The solution that can solve the maze problem is relevant across all gaming programming theories. As we learn in our textbook, there are some features for the definition of a search problem: A) States B) Initial State C) Goal State D) Actions E) Transition [3]. The initial and goal states can be marked in our two-dimensional environment in Figure 1. Moreover, There are available four possible actions in our model. Up or North, Down or South, Left or West, and Right or East. We have to consider obstacles in all possible actions; obstacles act as a wall in the real world, so facing them means failed activity. There are several algorithms to solve this problem, and the algorithms are used to find a minimum-cost path from a source to a destination by avoiding obstacles. This is a standard programming technique. They are mainly known for navigation and games. In this report, we will learn how to apply the core algorithms to a wide range of problems. Our first focus will be on BFS (Breadth First Search) and DFS (Depth First Search), which are fundamental for traversing graphs (or trees) and are often the first step in many other kinds of analysis. The algorithm solves the problem by searching regardless of the distance to the goal. In the next step, we will use two enhanced algorithms focusing on heuristic functions based on search techniques. These algorithms are called Greedy Best First Search and A*. The primary approach in the heuristic function is estimating the path costs from each cell to the goal and eliminating useless paths. Although heuristic search can not guarantee

success when it does, it is very effective. Although several implementations exist for a heuristic algorithm, we have implemented three famous ones. In the final stage, we will briefly compare these algorithms in terms of time and space complexity in different maze problems.

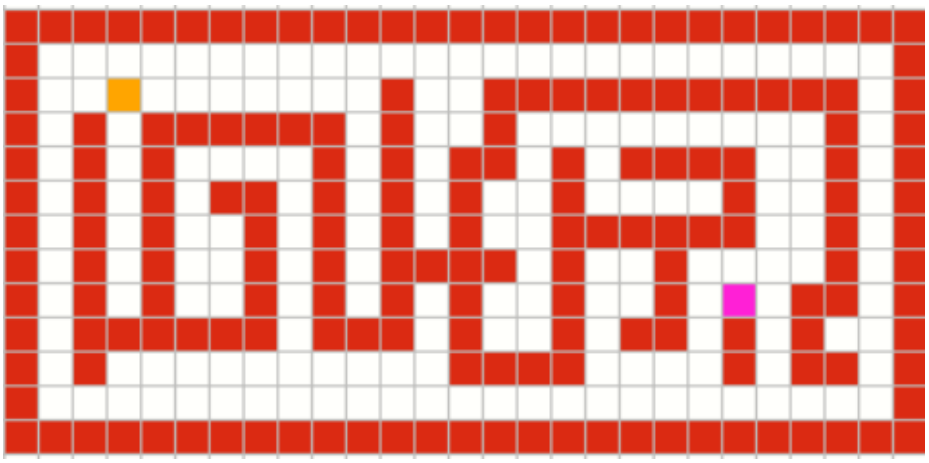

Figure 1: Orange cell as Initial state and pink cell as Goal state

2 Literature Review

In the first part, we will focus on two basic algorithms based on papers. Next, we will switch to the two enhanced algorithms. The first algorithm to solve a maze is Depth First Search (DFS). The first important point about DFS is that in the 19th century, a French mathematician called Charles Pierre Trémaux illustrated a version of depth-first search by solving mazes [4]. He marked a path on the floor with lines until he reached a dead end or found the exit. In the Graph Algorithms, Shimon mentioned that Deep-first search (DFS) is a method for scanning an undirected graph of finite depth [5]. A robust method for solving graph problems has been recognized since the papers of Hopcroft, and Tar-jan were published. Despite this, the algorithm has been known since the nineteenth century as a threading method. DFS is a particular case of another algorithm, suggested later by Tarry, that is just as good for threading mazes. The next step is finding how search works on DFS: In the Depth First Search paper written by Robert Tarjan: Let's assume that G is a graph we wish to explore. In the beginning, all vertices of G are unexplored. The process begins with choosing an edge from a vertex of G. Once the edge is traversed, a new vertex is created. Our journey continues along this edge. Edges lead to vertex, either new or already reached. Once we run out of edges leading from old vertices, we choose some unreachable vertex, if there is one, and begin a new exploration. We will eventually traverse each edge of G once. When such a process is carried out, it is called a search of G. In terms of time and space complexity in the CLRS book, it has been mentioned that a DFS traverses an entire graph in $O(|V| + |E|)$, where $|V|$ and $|E|$ denote the number of vertices and edges, respectively. There is a linear relationship between this and graph size [2]. Furthermore, in these applications, the stack of vertices on the current search path and the set of already visited vertices requires space $O(|V|)$ in the worst case. The second basic algorithm for solving a maze is Breadth-first search (BFS). The first point for BFS is that the breadth-first search and the method of finding all connected vertices of a graph were invented by Konrad Zuse in 1945, but the paper was published in 1972 [1]. E. F. Moore created a new version of BFS in 1959. His goal was to find the shortest path out of a maze using it. In a maze, Moore realized that if you take a small step along each path,

you will probably eventually find the shortest path more quickly than if you go down each path one by one until a dead end is reached. Starting at a vertex, one explores its neighbor vertices first, then the neighbors of its neighbors, and so on. A BFS algorithm is typically used to find the shortest path between two vertices. In the CLRS book, the time complexity is expressed as $O(|V|+|E|)$, where every vertex and edge will be explored in a worst-case scenario [2]. Vertices are represented by V, and edges are represented by E. The number of edges may vary between $O(1)$ and $O(|E|)$, depending on the sparsity of the input graph. Next, Artificial intelligence illuminated by Jones and Bartlett Learning has some points in terms of completeness: breadth-first search is complete, but depth-first search is not. Breadth-first search will eventually locate the goal state on an infinite graph represented implicitly, but a depth-first search may miss the goal state in parts of the graph where it doesn't exist. Our textbook mentioned that a Greedy best-first search is a form of best-first search that first expands the node with the lowest $h(n)$ (the value of the node that must be closest to the goal) because this is likely to lead to a solution quickly. So, the evaluation function $f(n) = h(n)$. By expanding a node according to a rule specified in the search algorithm, best-first searches explore a graph. Best-first search is described by Judea Pearl as estimating node n's promise by using a heuristic evaluation function $f(n)$. He believed that $f(n)$ might depend on the description of n and the goal [4][5]. It is often referred to as "best-first search," a search based on a heuristic to predict when a path will end. It means that closer paths to a goal are extended first. This specific search type is called greedy best-first search [3] or pure heuristic search.[2]. Next, about the history of A*: It was created as part of the Shakey project, which aimed to build a mobile robot to plan its own actions. The Graph Traverser algorithm was first proposed by Nils Nilsson for planning Shakey's paths. An algorithm called Graph Traverser uses a heuristic function, h(n), to estimate the distance from node n to the goal node. This ignores the distance g(n), the distance between the start and goal nodes. In Bertram Raphael's proposal, he suggested using g(n) + h(n). Hart invented the notions of admissibility and consistency for heuristic functions. It was initially designed to find the least-cost paths when the cost of a path is its total cost. However, it can be used to find optimal paths for any problem satisfying the cost algebra conditions [2]. Furthermore, our textbook mentioned that A* is a graph traversal and path search algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency. In practice, this method has a significant drawback, as all generated nodes are stored in memory, which makes the system extremely complicated in terms of space. Hence, in practical travel-routing systems, it is generally outperformed by algorithms that preprocess graphs to improve performance, as well as memory-bound approaches; however, A* is still the best solution in many cases [7].

3 Method and Algorithms

Depth First Search (DFS) is the first basic algorithm for solving maze problems. First, we focus on the definition of this algorithm. Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So, the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked 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. A related maze problem point about DFS is that Charles Pierre Trémaux investigated depth-first search as a maze-solving strategy in the 19th century [4]. The

question is how it works for the Maze problem. The depth-first approach is a common natural way for many people to solve problems like mazes. We choose a path in the maze (for the purpose of this example, let's choose a path based on some rules we lay out ahead of time) and follow it until we reach a dead end or the end of the maze. When a given path fails, we take an alternative path from an earlier junction and try it. Each vertex and edge in the graph is visited once by depth-first search In the BFS section, we know that Artificial intelligence algorithms use BFS for state-space searches. BFS has also been described for parallel computation in the functional programming paradigm. In the breadth-first search method, all nodes at level n will be visited first before visiting nodes at level n+1 (next level). The search starts from the root node and continues to the Its level from left to right, then moves to level 2, and so on [6]. In terms of optimality, BFS is optimal for finding the shortest path. DFS is not optimal for finding the shortest path. For memory, BFS requires more memory. DFS requires less memory. And for completeness, we consider BFS a complete algorithm, whereas DFS is not complete. A greedy best-first search algorithm aims to find the best path to a goal based on a given starting point. The algorithm prioritizes paths that appear to be the most promising, regardless of whether they are the shortest. Initially, the algorithm evaluates all possible paths and then expands the path with the lowest cost. Once the goal is reached, the process is repeated. There are some benefits of this algorithm:

1. Straightforward to Implement: This algorithm is relatively easy since it uses greedy best-first search.
2. Quick and Efficient: Greedy Best-First Search is a high-speed algorithm; it is strongly recommended for applications where quickness plays a significant role.
3. Low Memory Needs: Due to the small amount of memory required, Greedy Best-First Search is suitable for applications with limited memory resources.
4. Adjustable: Many problems can be addressed with Greedy Best-First Search, and they can be easily extended to more complex ones.

On the other hand, there are some cons to Greedy Best-First Search:

1. Inaccurate Outcomes: It cannot guarantee to find the optimal solution, as it is only concerned with finding the most promising path.
2. Local Optima: If Best-First Search gets stuck in local optima, the chosen path may not be the best.
3. Heuristic Process: It is complicated to implement Greedy Best-First Search because the algorithm requires a heuristic function.

The main question is, what is a Heuristic function? Informed Search uses this function to determine the best path. By taking the agent's current state as input, it estimates how close it is to the goal. Although the heuristic method may not always give the best solution, it guarantees that a suitable solution can be found in a reasonable amount of time. According to the heuristic function, a state is closer to the goal when it is closer to the heuristic function. h(n) represents an optimal path between two states and calculates the cost of that path. This value must be positive. We have three heuristics: 1. Euclidean distance: The length of a line segment between the two

Criterion	DFS	BFS	GBFS	A*
Complete	No	Yes	No	Yes
Optimal	No	Yes	No	Yes
Time	$O(b^m)$	$O(b^m)$	$O(b^m)$	$O(b^m)$
Space	$O(bm)$	$O(b^m)$	$O(b^m)$	$O(b^m)$

Table 1: Time and space complexity, Optimally, and Completeness

points. 2. Manhattan distance: The distance between two points measured along axes at right angles 3. Chebyshev distance: This examines the absolute magnitude of the differences between the coordinates of a pair of objects All the heuristic functions are consistent in this problem, a heuristic function is said to be consistent if its estimate is always smaller than or equal to the estimated distance from any neighboring vertex to the goal plus the cost of reaching that neighbor, which is one. Generally, for each node N and every successor P of N, the calculated cost of reaching the target from N is no greater than the step cost of getting to P plus the calculated cost of reaching the destination from P. That is:

$$h(N) \leq c(N, P) + h(P)$$

$h(G) = 0$
h is the consistent heuristic function.
N is any node in the graph.
P is any descendant of N.
G is any goal node.
$c(N, P)$ is the cost of reaching node P from N.
This search algorithm can be used to calculate the optimal paths from two nodes of a graph. It is a simple and efficient method. The shortest path will be determined by it. It is one of the best and most popular algorithms for finding paths and traversing graphs. Like Dijkstra, it builds a system by making a path tree from a start node to a target node at the lowest possible cost. The most significant advantage of A*, is that it uses a function f(n) for each node, which estimates the total cost. An algorithm is a provable method; however, a heuristic function is merely a guess, not a proven one. A* start to expand paths that are already less expensive by using this function:

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

$f(n)$ = total estimated cost of the path through node n
$g(n)$ = cost so far to reach a node
$h(n)$ = estimated cost from n to goal, which is a guess.
To have a brief comparison of A* and greedy BFS: Greedy BFS is not complete, not optimal, has a time complexity of $O(b^m)$, and a space complexity that can be polynomial. A* is complete, optimal, and has a time and space complexity of $O(b^m)$. So, in general, A* uses more memory than greedy BFS. A* becomes impractical when the search space is huge. All those, as mentioned earlier, were detailed definitions of the algorithms we picked. Now we want to employ these algorithms in actual circumstances for maze problems. In this area, we will evaluate them in terms of space and time complexity, optimality, and completeness. Furthermore, in the final stage, we will briefly compare these algorithms.

4 Experiment

In this section, we explain our experiments. We implemented the data structure of the Maze and our algorithms to solve it, and our goal is to experience how each of our selected algorithms solves different mazes with various sizes and numbers of walls. In the previous section, we discussed our chosen algorithms regarding time and space complexity, completeness, and optimality. These are the scientists' claims, and they used them when they compared these algorithms. Here, we want to experiment to determine whether their claims are correct. So, we designed many mazes to run our algorithms to solve them and observe how each algorithm performs in our selected metrics. We must clarify how we calculate and compare algorithms for each metric (time and space complexity, completeness, and optimality).

For time complexity, we calculate the CPU time of execution of each algorithm. For memory, for BFS, A*, and Greedy BFS, the maximum size experienced for the data structure is used, and for DFS, we used the maximum depth. For Completeness, we made some challenging mazes (small and large) to see whether an algorithm can find a path in an affordable time and memory. For optimality, first, we check whether our algorithm can find the optimal path in small mazes by visualizing them. Then we compare the length of these optimal paths with other given paths by algorithms.

We divide our experiment into small visual cases and large input files. Each of these two helps us show the difference in algorithms' performance in some metrics.

4.1 Small Mazes

We developed a user interface to enter minor visual and practical mazes for better observation and experience, leading to better evaluation and comparison. Among these mentioned mazes, we have exceptional cases to compare the algorithms. You can read about our user interface and the user instructions in Appendix C.

We will introduce the designed mazes in our user interface below and the output for running each of the algorithms. The discussion of the findings here will be elaborated in the next section.

Sample Visual Maze:

This maze is designed to challenge DFS and Greedy Best First Search to check whether they can find the optimal path. The maze is depicted in figure 2. The size of this test is $12 \times 24 = 288$. The output of running DFS, BFS, Greedy BFS, A* using Manhattan distance as a heuristic function are available in figures 3, 4, 5, and 6. As we observed, A* and BFS found a path of length 18. However, Greedy BFS found a path of length 20, and DFS found one with a length of 106. Another important observation, in this case, is the CPU time of solving the maze. For DFS, BFS, A*, and Greedy BFS, it is as follows: 0.312 seconds, 0.109 seconds, 0.297 seconds, and 0.109 seconds.

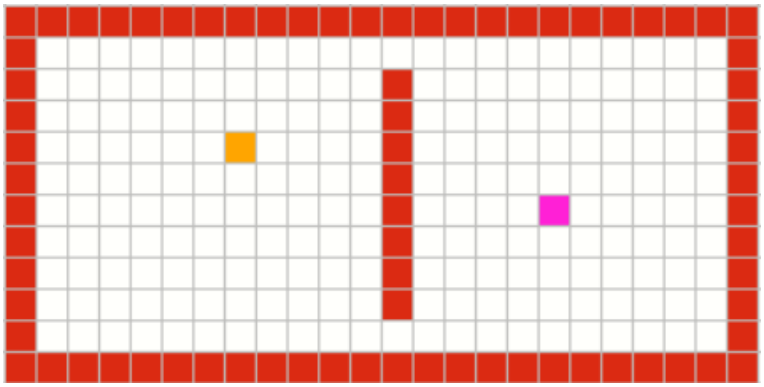

Figure 2: Small maze number 1

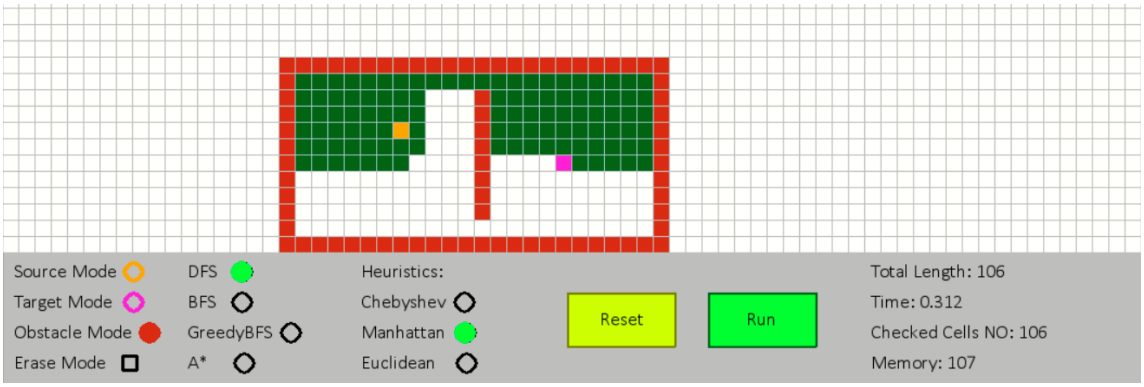

Figure 3: Small maze number 1 solved by DFS

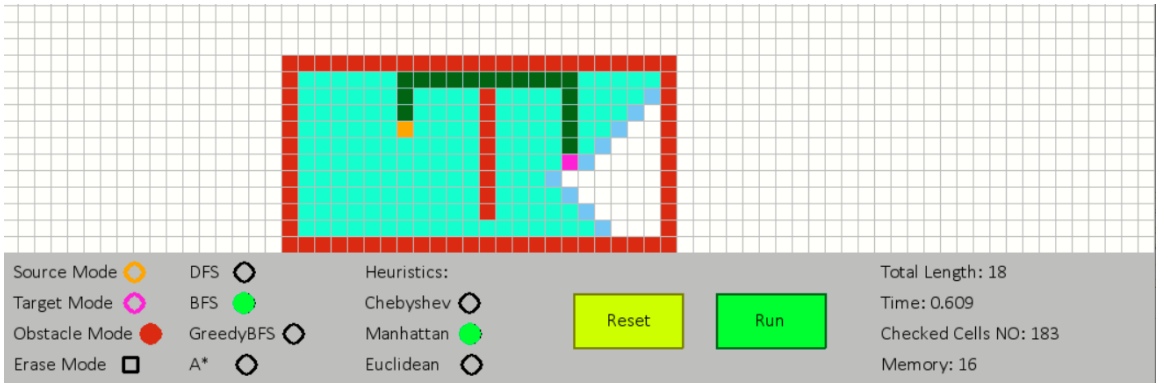

Figure 4: Small maze number 1 solved by BFS

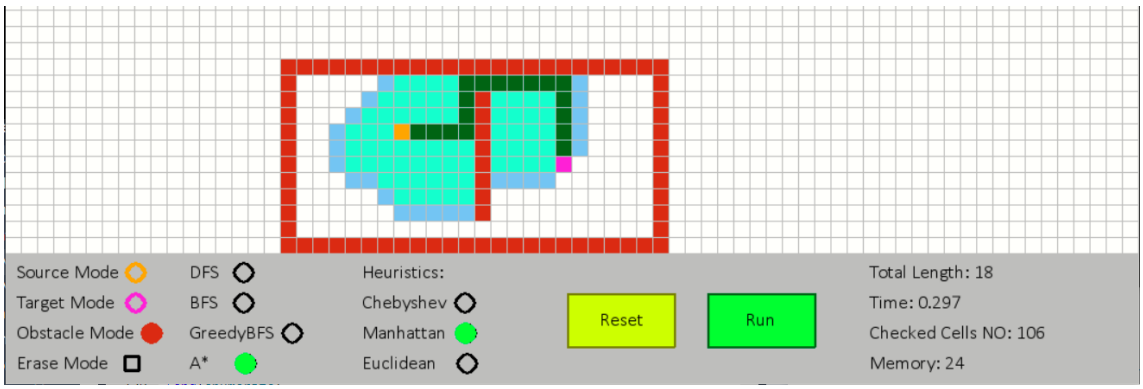

Figure 5: Small maze number 1 solved by A* using Manhattan distance

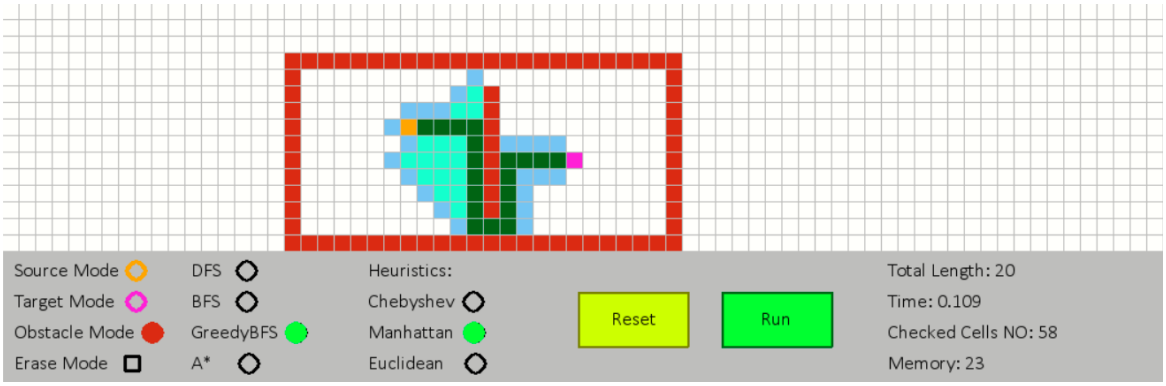

Figure 6: Small maze number 1 solved by Greedy BFS using Manhattan distance

Notice: There are many more visual cases experimented with. Here for the limitation on the report, we omitted them and tried to summarize our findings. You can find some of these mazes in Appendix E.

4.2 Large Mazes

Since the growth rate is significant in algorithm analysis, enormous test cases are essential in our experiment. Therefore, we developed a random test generator to build giant mazes (relative to our system's computation power). Our maze generator's manual is available in Appendix D. We will explain some of the important giant mazes below with the output of running algorithms. Based on these outputs, we also have data tables at the end of this section. To clarify, we built many mazes and experimented on them, and here we briefly explain some cases to introduce our findings. In fact, these cases help us analyze algorithms in terms of time and space complexity. They are also helpful for talking about the completeness of DFS. In the next section, we will discuss our findings by plotting a diagram of the resulting data.

First, we have four tests which are square-shaped mazes of different sizes with no obstacles.

1. Size of the maze: 80 rows, 80 columns: The output result is in table 2.
2. Size of the maze: 90 rows, 90 columns: The output result is in table 3.
3. Size of the maze: 100 rows, 100 columns: The output result is in table 4.

4. Size of the maze: 200 rows, 200 columns: The output result is in table 5 in appendix E.

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	No	-	-	-
BFS	Yes	53.9375	96	6378
GBFS (Chebyshev)	Yes	1.03125	125	242
A* (Chebyshev)	Yes	45.35937	184	5384
GBFS (Manhattan)	Yes	0.9375	237	354
A* (Manhattan)	Yes	30.0	238	3776
GBFS (Euclidean)	Yes	0.96875	125	242
A* (Euclidean)	Yes	43.78125	226	5167

Table 2: Results for test 1 of No-Obstacle mazes

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	No	-	-	-
BFS	Yes	81.671875	90	8063
GBFS (Chebyshev)	Yes	1.6875	180	346
A* (Chebyshev)	Yes	73.54687	171	7376
GBFS (Manhattan)	Yes	1.54687	256	422
A* (Manhattan)	Yes	70.67187	257	7288
GBFS (Euclidean)	Yes	1.79687	180	346
A* (Euclidean)	Yes	89.734375	172	7375

Table 3: Results for test 2 of No-Obstacle mazes

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	Yes	2.14062	199	198
BFS	Yes	156.9375	100	9999
GBFS (Chebyshev)	Yes	2.95312	198	395
A* (Chebyshev)	Yes	125.67187	198	9999
GBFS (Manhattan)	Yes	2.73437	197	394
A* (Manhattan)	Yes	120.28125	275	9987
GBFS (Euclidean)	Yes	2.828125	198	395
A* (Euclidean)	Yes	131.17187	198	9999

Table 4: Results for test 3 of No-Obstacle mazes

Furthermore, we generate some cases containing obstacles. Below three tests are available, and their results can be found in appendix E. The size of these mazes is equal to make the comparison more reasonable.

1. Size of the maze: 200 rows, 200 columns, 10 percent of maze's size obstacles: The output result is in table 6.
2. Size of the maze: 200 rows, 200 columns, 20 percent of maze's size obstacles: The output result is in table 7.
3. Size of the maze: 200 rows, 200 columns, 30 percent of maze's size obstacles: The output result is in table 8.

More mazes and findings, like the data tables, are available in appendix E.

5 Discussion of Findings and Comparison

In this section, we elaborate on our findings in the previous section. We will discuss our evaluation and comparison based on the acquired data.

As the past section explains, we compare our algorithms based on four metrics. Let's discuss each of them and go through comparisons one by one.

1. Completeness:
   (a) DFS cannot solve mazes, especially when they get larger. Because most of the time, they have to explore the whole space and go deep in a branch which leads to too much recursion that is not applicable for the system to handle. As we observed most of the time in giant tests, DFS failed to find a path in a reasonable time and memory.
   (b) BFS always gives us not only a path to the target but also an optimal one. It is evident in the outputs of both visual cases and huge cases. The problem is that BFS explores the whole space, which spends a lot of time in gigantic cases. Although BFS uses a lot of memory in general cases, in our problem, the branching factor is usually less than four. So, memory is not a big deal.
   (c) Greedy BFS can find a path in almost all cases. It is because our graph of the problem is dense, and all cells have a degree of four. Therefore, in the worst case, even if the Greedy approach guides us to the wrong way, it can still find its path to the target by continuing the search.
   (d) A* consumes significantly more time and memory when we compare it to Greedy BFS. Using our various heuristics does not affect completeness.
2. Optimality:
   (a) DFS is not complete and even when it can solve the maze (maybe by chance or exploring everywhere) it costs too much. Our findings show that catching the optimal path is too rare for DFS.
   (b) BFS is cost-optimal; however, it is not optimal in using time or space!

(c) Greedy BFS is not optimal in all cases, and one of our evidence for this claim is the visual maze we illustrated in the previous section. It is too fast to find a path but does not guarantee optimality.

(d) A* uses more memory and time but always finds the optimal path. As we proved in the method section, our heuristics are consistent. It is not strange to get the best path to the target using A* with a consistent heuristic. Our various heuristics may cost differently, but they all do their best to find the path.

3. Time:

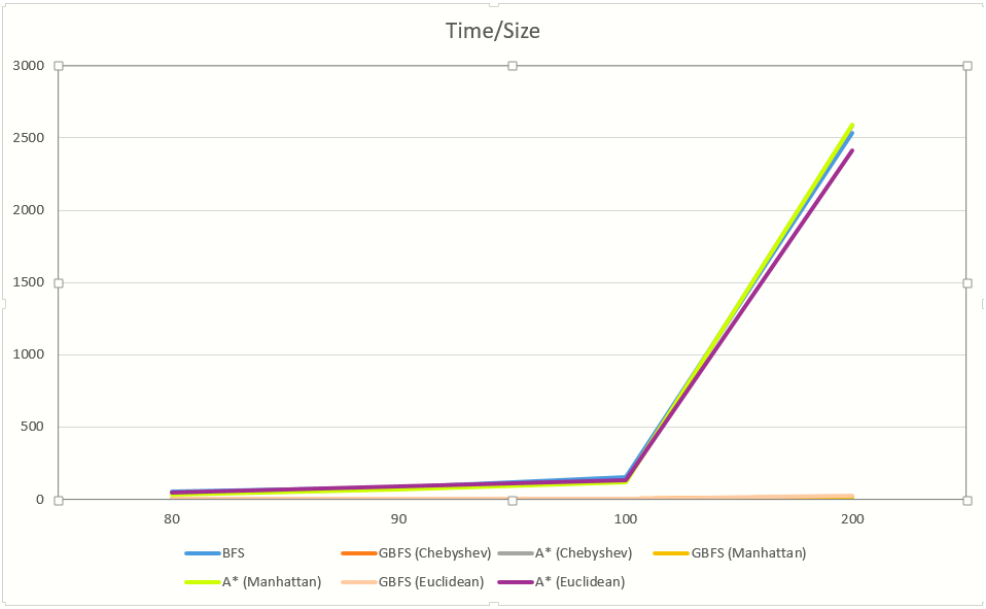

Figure 7: Comparison between algorithms' CPU time by growing size of the maze

(a) DFS cannot find the path in large cases, so we put it aside in this metric as its time of finding the path is unknown.

(b) BFS is one of our algorithms whose CPU time grows fast by increasing the maze size. As we observe in the plot in figure 7, it has an exponential time complexity in our problem.

(c) Greedy BFS is wonderful in terms of time. Its growth rate in our problem is slow relative to the others. It is one of the advantages of this algorithm over A*. Although it cannot always find the best path, it can find a good enough path at a great time.

(d) A* is similar to BFS in this metric. As we observed in figure 7, the CPU time of A*, regardless of using any of the three heuristics, is exponential by growing the maze's size.

4. Memory:

(a) DFS: Like what we had for the previous metric, again, according to being incomplete in solving the large mazes, we put DFS aside from comparing with this metric.

(b) BFS, Greedy BFS, and A*: We mentioned that the branching factor is small in this problem, and regarding the computational power of our system, making the mazes

bigger than this puts us in trouble. Therefore, the memory usage of algorithms in this experiment is not high, and the growth rate in terms of space is not noticeable through our test cases. After all, based on the branching factor and maximum depth, we expect the space complexity to be tolerable for all the mentioned algorithms.

6 Conclusion

Our goal in this project was an experiment on solving mazes using different algorithms to evaluate their performance and compare them in terms of four metrics (Completeness, Optimality, Time, and Space Complexity). We implemented algorithms and other stuff, like a user interface or test generator, which you can read about them more in appendices B, C, and D. Using our developed system, we have experimented with different scale mazes, both visual and non-visual. You can find a part of the outputs in the Experiment section, and you can find more in appendix E. We evaluate and compare the performance of our chosen algorithms in solving mazes using three different heuristics for informed ones. The evaluation is available in the previous section. We believe our experiment and results are reasonable and justify our claims about the algorithms in maze solving.

7 Future Work

There are many ideas to extend this work to be a complete analysis of solving maze problems.

Basically, adding more algorithms to the pile of selected algorithms can always be a good choice. However, first, we have many better ideas too. First of all, most of the time, our actions in solving the maze can contain diagonal moves, which was not part of our project. We guess it affects the use of the heuristics we chose, like Manhattan distance.

Moreover, the cost of each action in our problem is one. There are many cases when actions cost differently, like when some special cells exist in the maze. Thus, adding weight or letting the cells have random costs is another idea.

In this project, our focus was on the versions of a maze without obstacles or with few ones. They can affect the performance of all the algorithms when their frequency increases and when we have special distributions of them in the maze.

References

[1] Konrad Zuse. "Der Plankalkül (in German)". In: *Addison-Wesley* (1984), p. 48.

[2] Thomas H. Cormen et al. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[3] Peter Russell Stuart J.; Norvig. *Artificial intelligence : a modern approach*. Pearson, 2003.

[4] professor Jean Pelletier-Thibert. "Charles Pierre Trémaux". In: *Annals academic* (2010).

[5] professor Jean Pelletier-Thibert. "Graph Algorithms". In: *Cambridge University Press* (2011), pp. 46–48.

[6] professor Jean Pelletier-Thibert. "Graph Algorithms". In: *Cambridge University Press* (2011), pp. 46–48.

[7] Stuart J. Russell. *Artificial intelligence a modern approach*. Boston: Pearson, 2018.

Appendices

A Appendix: Workload Distribution

SayedMohammad Hashemi

- Implementation of algorithms such as A* and Greedy BFS
- Implementation of User Interface
- Designing special visual tests
- Documentation in Abstract, Experiment, Comparison, Conclusion, and Future Works
- Evaluation and Comparison: Analysis and Design phase and implementing the Main method
- GitHub and project management

MohammadEhsan Akhavanpour

- Implementation of algorithms such as DFS and BFS
- Implementation of Test Generator
- Designing special visual tests
- Documentation in Introduction, Literature Review, Method and Algorithms
- Evaluation and Comparison: Analysis and Design phase

B Appendix: GitHub Repository and Running Instructions

All the codes, documents, tests, and other stuff are available in the repository of the project: `https://github.com/smh997/Maze-Problem`

There are setup and running instructions in the README file in GitHub. However, it is also available here:

Setup:

```
git clone https://github.com/smh997/Maze-Problem.git
cd code
pip install -r requirements.txt

```

Run (User Interface):

```
cd code
python main.py

```

Run (Input Test File):

```
cd code
python main.py ./tests/name_of_the_json_file.json

```

C Appendix: User Interface Manual

The user interface is developed to make it easy for users to work with our code, designing arbitrary mazes and getting the answer for their maze. It is also helpful in evaluating algorithms by visualizing the mazes and algorithms' performance in solving them. Here is the manual for using this user interface. It is available in our GitHub repository too.

After following setup instructions from appendix B, follow the steps below to run the code in UI mode:

1. Be sure to be in the Code directory.
2. Open CMD or terminal
3. Insert "python main.py"
4. Now, UI must open.

- Cell Mode: As you can see in figure 8, there are three options you can choose, and after selecting each radio button, you can put the selected cell in the maze.

Figure 8: Cell Mode Radio Buttons

- Erase Mode: This mode is used for removing obstacles and any cell except the source and target. In figure 9, you can see erase mode checkbox.

Figure 9: Erase Mode Checkbox

- Algorithm Mode: As you can see in figure 10, there are four buttons you can select, and by choosing each radio button, the algorithm is set.

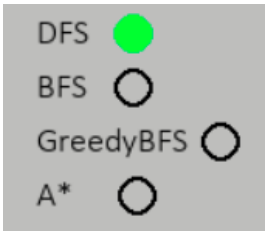

Figure 10: Algorithm Mode Radio Buttons

- Heuristic Mode: You can see three buttons in figure 11, you can select, and by choosing each radio button, the heuristic is set.

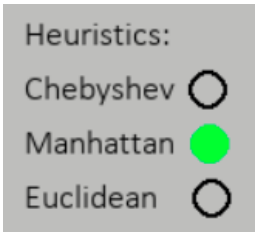

Figure 11: heuristic Mode Radio Buttons

- Run and Reset buttons: You can see two buttons in figure 12; they are for resetting the maze and running the algorithm.

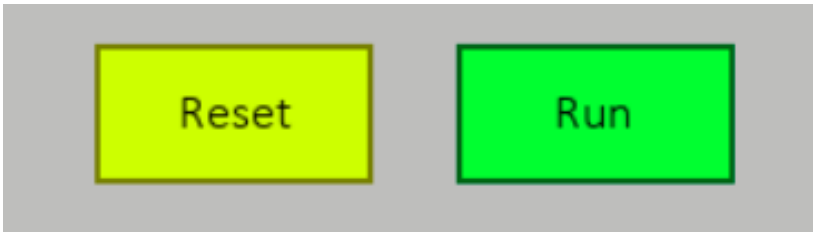

Figure 12: Reset and Run Buttons

D Appendix: Test Generator Manual

The test generator receives at most 6 variables for running:

1. Number of tests to be generated. Its default value is 10.
2. Start of the range for choosing the number of rows. Its default value is 50.
3. End of the range for choosing the number of rows. Its default value is 500.
4. Start of the range for choosing the number of columns. Its default value is 50.
5. End of the range for choosing the number of columns. Its default value is 500.
6. Percentage of obstacles in the maze. Its default value is 25.

It sets them to the default value if we do not give it any variables. The test generator starts creating tests choosing the number of rows randomly from the given range. it does the same for the number of columns. Then, it chooses the source and target randomly. Finally, it chooses obstacles. The output of the generated test is a JSON file stored in the "tests" directory beside the test generator.

There are instructions for using the test generator in the README file in GitHub. However, it is also available here:

Test generator instruction:

```
cd code
python test_generator.py <numberOfTests> <startRowRange> <endRowRange> <startColumnRange> <endColumnRange> <percentageOfObstacles>

```

E Appendix: More Mazes and Results

There are some visual tests and data tables. Our large test cases are available on the GitHub repository.

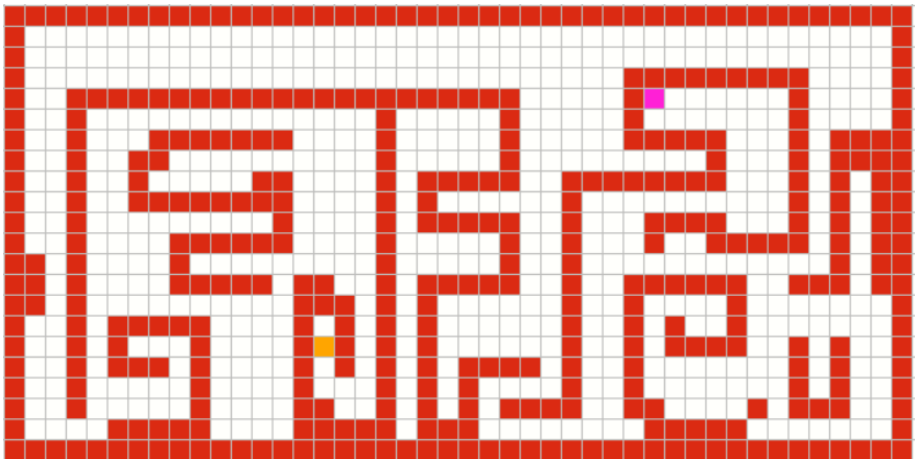

Figure 13: Small maze number 2

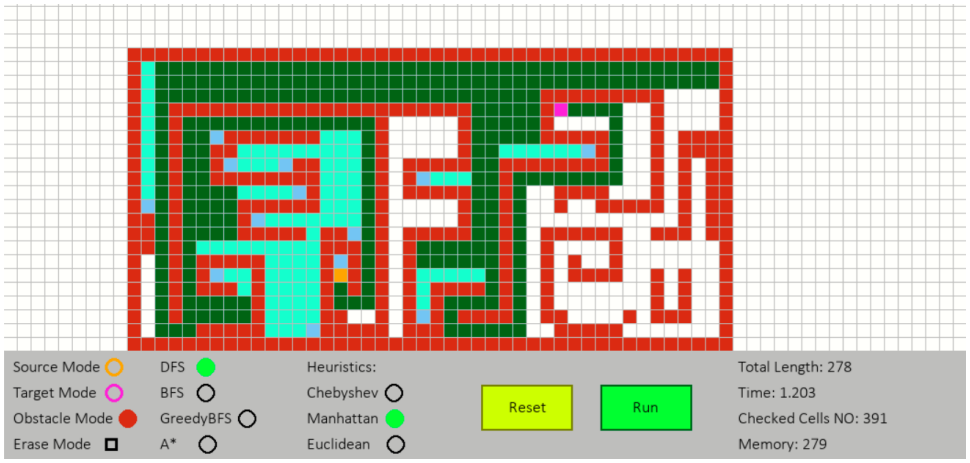

Figure 14: Small maze number 2 solved by DFS

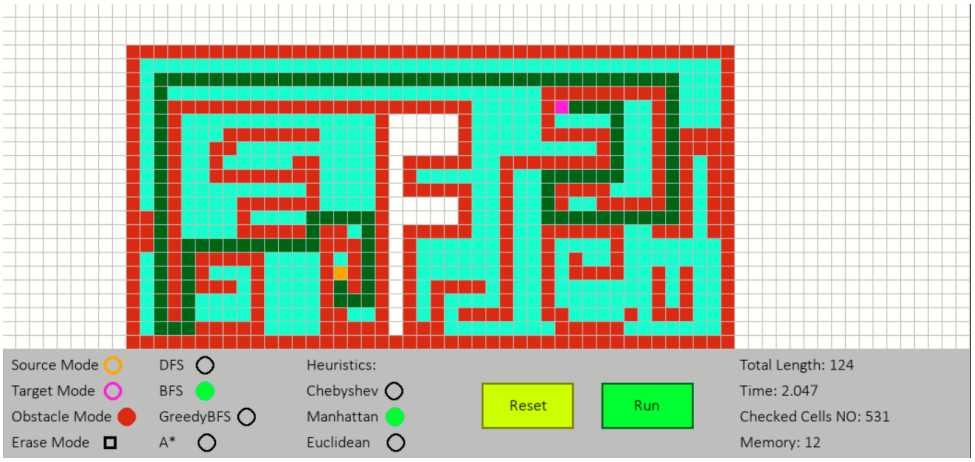

Figure 15: Small maze number 2 solved by BFS

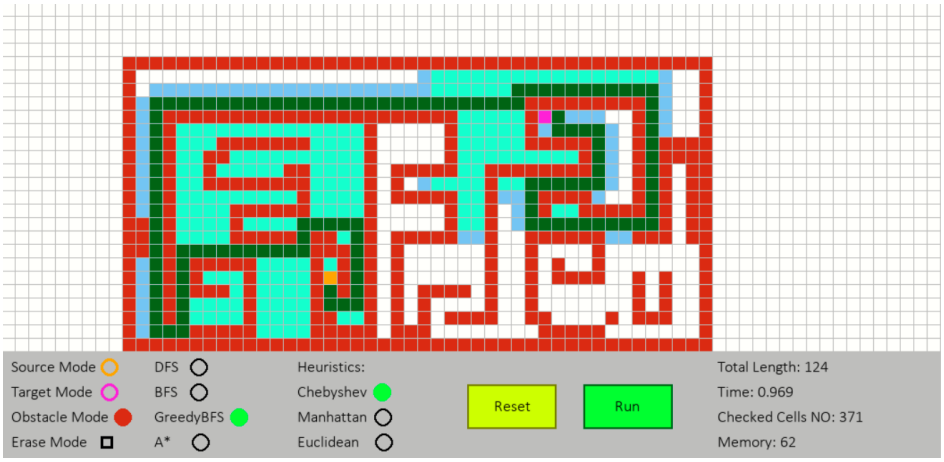

Figure 16: Small maze number 2 solved by Greedy BFS using Chebyshev distance

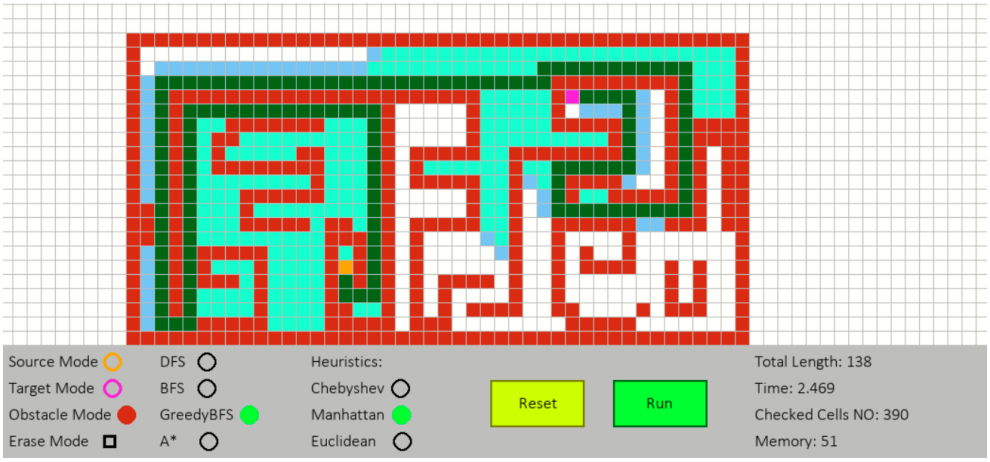

Figure 17: Small maze number 2 solved by Greedy BFS using Manhattan distance

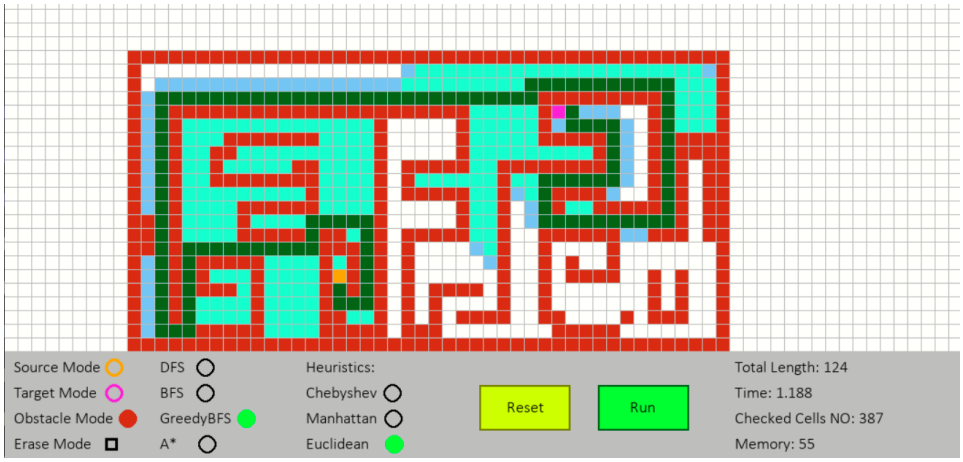

Figure 18: Small maze number 2 solved by Greedy BFS using Euclidean distance

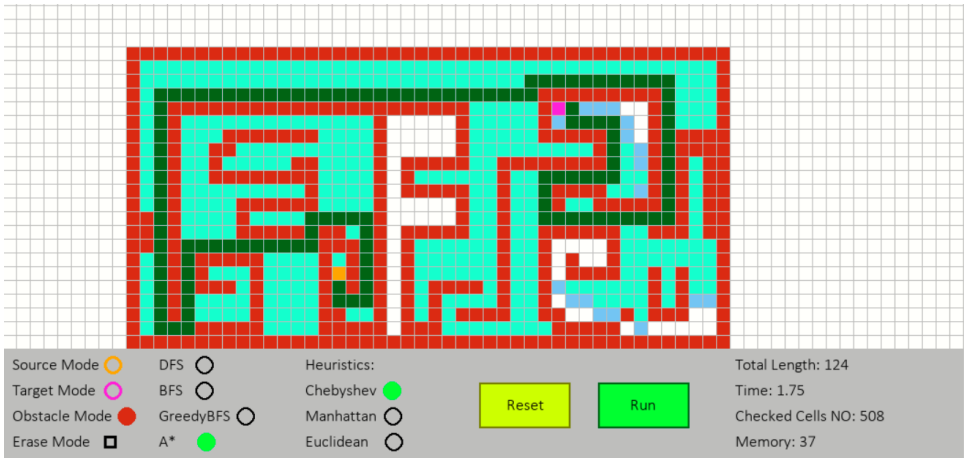

Figure 19: Small maze number 2 solved by A* using Chebyshev distance

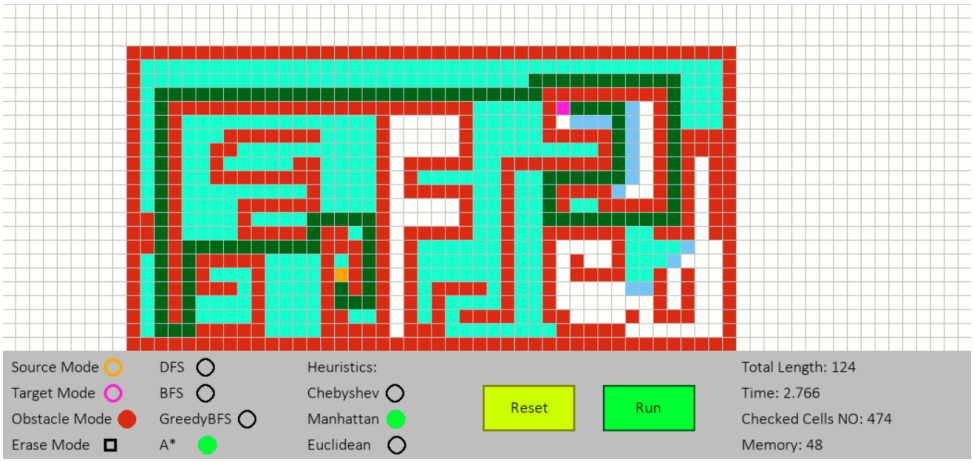

Figure 20: Small maze number 2 solved by A* using Manhattan distance

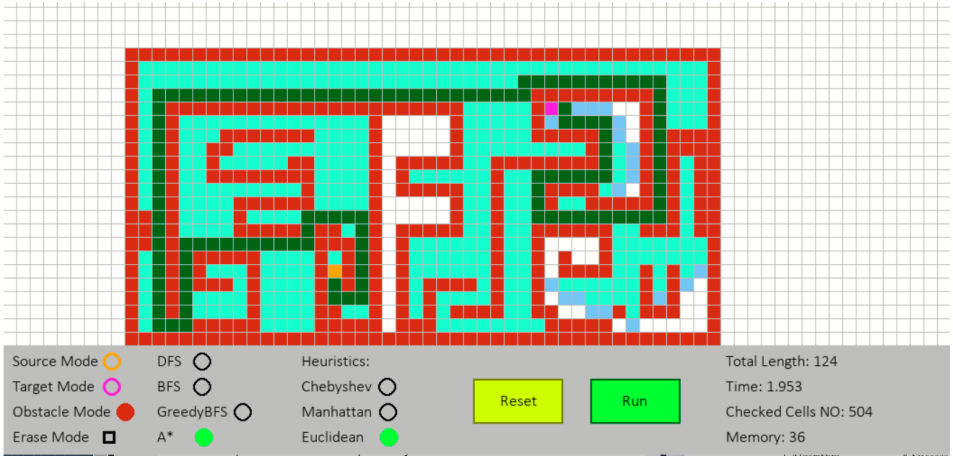

Figure 21: Small maze number 2 solved by A* using Euclidean distance

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	No	-	-	-
BFS	Yes	2537.75	200	39999
GBFS (Chebyshev)	Yes	21.64062	398	795
A* (Chebyshev)	Yes	2584.17187	398	39999
GBFS (Manhattan)	Yes	20.15625	397	794
A* (Manhattan)	Yes	2595.48437	200	39999
GBFS (Euclidean)	Yes	24.35937	398	795
A* (Euclidean)	Yes	2416.1875	398	39999

Table 5: Results for test 4 of No-Obstacle mazes

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	No	-	-	-
BFS	Yes	1644.89062	202	34127
GBFS (Chebyshev)	Yes	12.75	291	573
A* (Chebyshev)	Yes	1122.40625	336	23159
GBFS (Manhattan)	Yes	13.25	462	766
A* (Manhattan)	Yes	724.26562	639	16123
GBFS (Euclidean)	Yes	12.26562	296	574
A* (Euclidean)	Yes	1012.60937	386	22226

Table 6: Results for test 1 of mazes with obstacles, in this one, the amount is 10 percent

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	No	-	-	-
BFS	Yes	1644.75	189	31756
GBFS (Chebyshev)	Yes	14.82812	300	616
A* (Chebyshev)	Yes	996.875	342	21117
GBFS (Manhattan)	Yes	16.92187	396	754
A* (Manhattan)	Yes	393.21875	751	8783
GBFS (Euclidean)	Yes	14.01562	290	585
A* (Euclidean)	Yes	871.65625	345	18380

Table 7: Results for test 2 of mazes with obstacles, in this one, the amount is 20 percent

Algorithm	Success in reasonable time	CPU time	Memory metric	Checked cells
DFS	Yes	232.73437	2501	5285
BFS	Yes	1205.29687	187	26712
GBFS (Chebyshev)	Yes	18.9375	280	689
A* (Chebyshev)	Yes	780.375	238	18449
GBFS (Manhattan)	Yes	27.51562	341	898
A* (Manhattan)	Yes	175.5625	1784	5884
GBFS (Euclidean)	Yes	19.76562	275	674
A* (Euclidean)	Yes	782.39062	269	17455

Table 8: Results for test 3 of mazes with obstacles, in this one, the amount is 30 percent