Comparison of Heuristic Search Performance Using a Maze Solver Agent

Marion G. Sisk, *Student, MSSWE, Kennesaw State University*

Abstract—**Maze solvers are a common type of online search agent and are often used in applications like video games, robotics, path finding, and many other foci of research. In this exercise, a randomly generated maze provides an environment for an online search agent to explore, safely. In this environment, the efficacy of the online search agent depends upon the admissibility of the heuristic. We explore the efficacy of four different heuristics on a large number of random mazes using Monte Carlo analysis techniques. The online search agent uses a depth-first search, a random walk, a Euclidean distance, and a Euclidean distance with look ahead heuristic. The latter of which has been created as an attempt to improve upon the performance of the Euclidean solver, which is shown to be excellent in its own right. The analysis concludes with the Euclidean distance heuristic performing vastly better than the other three heuristics with the newly developed Euclidean distance with look ahead performing the worst out of all four.**

I. Introduction

THE use of heuristic enabled searches has been a fundamental tenet of artificial intelligence (AI) for some time. As is the case with these heuristic searches, the efficacy can vary tremendously over the configuration of the search space, and the admissibility of the heuristic itself. This is especially true for a subset of heuristic searches called online searches. Online searches only have access to local information regarding the current state space and are largely unaware of the state space beyond the location of the agent at a particular state. In this paper, four different types of online searches are tested using Monte Carlo methods to find statistically significant differences between them. The four heuristics include a depth-first search, a random-walk search, a Euclidean distance heuristic, and a Euclidean distance with look ahead heuristic. Each of these is measured according to the competitive ratio upon solving the maze. The pseudocode, and experimentation details are given later in the paper, however each has been implemented as an element in a JavaFX Application. Screenshots have been included alongside descriptions of the algorithms. The results are tabulated, and important findings are highlighted. Finally, implications for future work are discussed.

II. Background Information

Within the field of artificial intelligence exists a collection of search algorithms that excel in dynamic environments. Unlike offline searches, where a complete solution is computed before action is taken, online searches interleave computation and action. Online search agents are required for this to work correctly. An agent knows its current state within the state space and is able to make comparisons between the current state and adjacent states. This ability to make comparisons is where heuristics play a role.

There is a fair amount of literature available on the subject of heuristic searches, especially with respect to pathfinding, and other exploratory search algorithms. One of the most notable algorithms, A*, is often used because of its completeness and its ability to find efficient paths within a static state space. However, there are different limitations when a dynamic state space is involved.

The dynamic state space for this online search agent will be a randomly generated maze. The maze itself is a finite state space and is safely explorable. Provided that agent actions are reversible, a path between any two points can be found. (The maze itself is a spanning tree, created from a randomized Prim's algorithm). Brief discussions of the four approaches used in the maze solver agent are given below.

A. Depth First Search (DFS)

A depth first search is complete within a finite state space and is only capable of finding a solution if the agent actions are reversible. In the context of mazes, DFS can be compared to a "right hand rule." Where, when faced with a choice of next steps, the step to the right is taken until a dead-end is reached. At which point, the agent turns around, and makes the first possible right turn. This approach is excellent if the goal is exploration. If we want the agent to learn the entirety of a state space, this is optimal. However, for finding a solution between two points within a maze, this method can often be led astray, and will spend considerable time exploring sections of the maze that are entirely unnecessary. DFS is considered a baseline for our comparisons in this paper.

This paper was submitted on May 1, 2021 to meet the requirements of the CS7375 – Artificial Intelligence course held at Kennesaw State University. Acknowledgement should be made to Blue Bird Body Company who has graciously allowed for a flexible work schedule to permit completion.

B. Random-Walk Search

The random-walk search starts with the agent at a particular state and an adjacent state is chosen at random and the agent moves that particular state. This behavior is akin to hill-climbing search, in that it is only privy to local state information, but it differs in that the agent cannot perform a random restart when local maxima are encountered (in the context of a maze, this would be a dead-end path). To make a "random restart" effect possible, the agent moves to adjacent states randomly. This type of search has the possibility of being more efficient than a simple depth-first search, and it will certainly find a goal eventually, but it can be very slow in achieving its goal of finding a solution path, since it can encounter specific states more than twice, and backtracking consumes exponentially more steps depending on the size of the state space. It is expected that Random-Walk search will only be marginally better than depth-first search in terms of the competitive ratio.

C. Euclidean Distance Heuristic

Up until this point, the search methods discussed did not really act in an intelligent manner but rather either followed an overly simple process, or just randomly moves through the state space until it hits the goal. Neither of which is really ideal since they only run across the goal state by happenstance and are not purposefully seeking out the goal state. With the addition of some additional information (some of it intrinsic to the state space itself) it becomes possible to make a more intelligent online search agent.

The maze itself is defined using an X-Y Grid with each cell representing a particular element in the state space, and each cell also being defined with unique pairs of X-Y coordinates.

Using the Euclidean Distance Heuristic can help provide the online search agent with a focus. In the context of a maze, the heuristic calculates the straight-line distance between the current state and the goal state according to the following equation:

$$d = \sqrt{(L_{S^1})^2 + (L_{Sg})^2} \tag{1}$$

Where L_{S^1} is the location of the search agent in the state space, and L_{Sg} is the location of the goal state within the state space.

When the online search agent determines which adjacent state to move to, it calculates the Euclidean distance using (1) and chooses the adjacent state with the smallest value (e.g.: it chooses the adjacent state that is closest to the goal). This inherently greedy approach can lead the agent astray, and it is possible still that large portions of the state space may be explored before the goal state is found. Additionally, when the agent reaches a dead end, it must have some means of keeping track of where it has been, such that when it back tracks, it will not repeat the same path twice.

The state space itself keeps track of how many times the agent has visited. This process could compare to leaving a bread-crumb trail through a forest. This permits the agent to keep track of where it has been, without keeping that information internally. Even still, the agent will often reach local minima (dead-ends) and will still backtrack to explore the most recently encountered unexplored branch. Because of this limitation, it may be prudent to have some additional knowledge of the immediate area surrounding the agent, not just the adjacent states, when determining which way to go. Thus, we explore:

D. Euclidean Distance with Look Ahead

This heuristic uses the same location-based functions as the Euclidean Distance Heuristic but adds a look-ahead element such that the agent can make "less greedy" decisions.

The way in which the look-ahead element works would permit the search agent to determine (within the look ahead limit) if a particular path leads astray, or if it continues closer to the goal state.

Instead of using the immediately adjacent states, the search agent sends "mini agents" in each possible direction to report back what they find, given a certain look-ahead limit. At the look-ahead limit, the "mini agents" send back the Euclidean distance between the end states and the goal state. The search agent then makes the determination on which adjacent state to move based on the results given by the "mini agents." This method is expected to provide the search agent with a bit more insight into the local state area, and thus be able to make more appropriate decisions in order to find the solution path with fewer stray paths.

Naturally, the spawning of "mini agents" will increase the computational and memory footprints of the search agent, but it's a finite increase that is a function of the look-ahead distance. One would only expect a scaling factor of 4n for computational time and memory footprint, with 'n' being the look-ahead distance.

III. Algorithms and Pseudocode

The implementation of the maze generator is based on a randomized version of Prim's Algorithm. The maze itself is comprised of a custom graph class that contains all the associated attributes and methods to keep track of walls and passages alongside the actual cells. This is the environment in which the search agent will be operating. The agent, initially, will only know what cell it is starting at, and what the goal cell is.

The metrics of evaluation used in this comparison are based on the competitive ratio of the agent's actions for each of the heuristics used. The competitive ratio is defined as:

$$r_c = {}^{|V|}/_{|S|} \tag{2}$$

Where S is the set of cells along the solution path from start to finish, and V is the set of visits to each cell. Note that also:

$$S \in V \tag{3}$$

In an ideal situation, the cardinality of visits should be equal to the cardinality of the solution path set. Thus, the closer to one the competitive ratio is, the better the agent has performed. The comparisons made in this exercise center around aggregate values (averages, standard deviations, ranges). These values provide for a common framework to compare the performance of each heuristic. The way in which these values are reached is given below in the discussion of the subsequent Monte Carlo analysis.

A. Monte Carlo Analysis

Because of the random nature of the maze generation, normal deterministic approaches will not be sufficient to produce a meaningful quantitative comparison between the four heuristics in question. Monte Carlo methods are used when exact deterministic analyses are difficult to impossible. The essence of this analysis is that a large number of random seeded analyses are performed, and data collected from each iteration. With a sufficiently high number of iterations, an overall trend can be shown through statistical analysis. In Fig. 1, the pseudocode of the Monte Carlo Analysis is provided.
The parameters used for the Monte Carlo analysis used in this exercise includes 5,000 different randomly generated mazes, with each maze having a set of 10 randomly generated start/finish pairs. Each heuristic algorithm is run on each maze and each start/finish pair and the results are written to a *.CSV output file. This file is then imported into Microsoft Excel for subsequent statistical analysis. The results are tabulated and discussed in Section IV. Experimental Results.

```
FUNCTION monteCarloTests()
FOR EACH MAZE ITERATION
        GENERATE a RANDOM MAZE
        FOR EACH START ITERATION
                GENERATE a RANDOM START and FINISH
                SOLVE maze with DFS
                RECORD SOLUTION PATH LENGTH and VISIT COUNTS
                RESET visit counts and solution path
                SOLVE maze with RANDOM WALK
                RECORD SOLUTION PATH LENGTH and VISIT COUNTS
                RESET visit counts and solution path
                SOLVE maze with EUCLIDEAN HEURISTIC
                RECORD SOLUTION PATH LENGTH and VISIT COUNTS
                RESET visit counts and solution path
                SOLVE maze with LOOK AHEAD HEURISTIC
                RECORD SOLUTION PATH LENGTH and VISIT COUNTS
        END FOR
END FOR
WRITE RECORDED RESULTS to OUPUT CSV FILE
END FUNCTION
```

Fig. 1. Pseudocode for Monte Carlo analysis. Subsequent pseudocode for specific solving heuristics is provided is other figures.

B. Depth First Search Algorithm

The depth first search algorithm is fairly straight forward. The algorithm basically picks the first available move at each time. It does not attempt to pick the cell closer to the goal, nor does it have any randomized logic associated with it. It merely picks the first available move as determined by a built-in method. The pseudocode for the depth first search heuristic is given below in Fig. 2. Note that the basic mechanics of the DFS heuristic is replicated with the other specific heuristics. The way in which the next cell is chosen is the primary difference between each of them.

```
FUNCTION depthFirstSearch()
WHILE the CURRENT CELL is NOT EQUAL to the GOAL CELL
        ADD the CURRENT CELL to the SOLUTION PATH
        MARK the CURRENT CELL as VISITED
        DETERMINE POSSIBLE MOVES FROM CURRENT CELL
        IF there are NO POSSIBLE MOVES available:
                REMOVE the CURRENT CELL from the SOLUTION PATH
                SET the CURRENT CELL as next in SOLUTION PATH
                REPEAT the WHILE LOOP
        ELSE IF there are POSSIBLE MOVES available:
                CHOOSE the FIRST POSSIBLE MOVE
                SET the CURRENT CELL from FIRST POSSIBLE MOVE
                REPEAT THE LOOP
        END IF
END WHILE
RETURN solutionPathStack
END FUNCTION

FUNCTION determinePossibleMoves()
FOR each NEIGHBORING CELL from CURRENT CELL
        IF the WALL between CURRENT CELL and ADJACENT CELL is a passage ...
            AND IF the ADJACENT CELL has not been VISITED
                ADD ADJACENT CELL to POSSIBLE MOVES
        END IF
END FOR
RETURN possibleMovesArray
END FUNCTION
```

Fig. 2. Pseudocode for the depth first search heuristic.

Note that the function defined in Fig. 2: determinePossibleMoves(), is a common method that is used throughout all of the heuristics in this experiment. The method does keep track of what cells have been visited, and as such, all of the heuristics given here do have a form of “memory” to help make better choices, even in the absence of any other heuristic.

C. Random Walk Algorithm

The random walk heuristic extends upon the depth first search by changing the way in which the adjacent cell is chosen. Unlike in DFS, where the first available move is chosen and subsequently made, the Random Walk algorithm chooses a move from the list of possible moves based on a random number value when the solver is called. The pseudocode for the Random Walk algorithm is given in Fig. 3 below.

```
FUNCTION randomWalkSearch()
WHILE the CURRENT CELL is NOT EQUAL to the GOAL CELL
        ADD the CURRENT CELL to the SOLUTION PATH
        MARK the CURRENT CELL as VISITED
        DETERMINE POSSIBLE MOVES FROM CURRENT CELL
        IF there are NO POSSIBLE MOVES available:
                REMOVE the CURRENT CELL from the SOLUTION PATH
                SET the CURRENT CELL as next in SOLUTION PATH
                REPEAT the WHILE LOOP
        ELSE IF there are POSSIBLE MOVES available:
                CHOOSE the FIRST POSSIBLE MOVE
                SET the CURRENT CELL from RANDOM POSSIBLE MOVE
                REPEAT THE LOOP
        END IF
END WHILE
RETURN solutionPathStack
END FUNCTION
```

Fig. 3. Pseudocode for the Random Walk search heuristic.

D. Euclidean Distance Algorithm

The searches up until this point have been largely non-intelligent by design. The algorithms (DFS and Random Walk) provide a baseline of performance. The Euclidean distance algorithm makes the choice of what adjacent cell to move to, based on the distance between the adjacent cell and the goal cell. Because the agent makes a decision based on locally available information, it fulfills the role of an online search agent, and exhibits behavior that would classify it as artificial intelligence. The Euclidean Distance algorithm does require that the agent know the location of the goal cell, and it does require that the agent keep track of what cells have been visited (since it does use the determinePossibleMoves() method given in Fig. 2). The pseudocode for the Euclidean Distance Algorithm is given below in Fig. 4.

```
FUNCTION euclideanSolve()
WHILE the CURRENT CELL is NOT EQUAL to the GOAL CELL
        ADD the CURRENT CELL to the SOLUTION PATH
        MARK the CURRENT CELL as VISITED
        DETERMINE POSSIBLE MOVES FROM CURRENT CELL
        IF there are NO POSSIBLE MOVES available:
                REMOVE the CURRENT CELL from the SOLUTION PATH
                SET the CURRENT CELL as next in SOLUTION PATH
                REPEAT the WHILE LOOP
        ELSE IF there are POSSIBLE MOVES available:
                CHOOSE the FIRST POSSIBLE MOVE
                SET the CURRENT CELL from SHORTEST DISTANCE
                REPEAT THE LOOP
        END IF
END WHILE
RETURN solutionPathStack
END FUNCTION

FUNCTION shortestDistance()
        SET X_1 POSITION, Y_1 POSITION from CURRENT CELL
        SET X_2 POSITION, Y_2 POSITION from GOAL CELL
        RETURN SQRT((X_1 – X_2)^2 + (Y_1 – Y_2)^2)
END FUNCTION
```

Fig. 4. Pseudocode for the Euclidean Distance search heuristic.

E. Euclidean Distance with Look Ahead

This heuristic is an attempt at improving the performance of the Euclidean Heuristic above. The general idea behind it is that the search agent will send "mini agents" out along each possible move. Each of these "mini agents" will traverse a look ahead limit (for example, in this exercise a look ahead limit of 3 cells was chosen arbitrarily). The "mini agents" then return back the Euclidean distance between the furthest cell they have traversed and the goal cell. The main agent then chooses the adjacent cell based on the distances presented to it by the "mini agents."

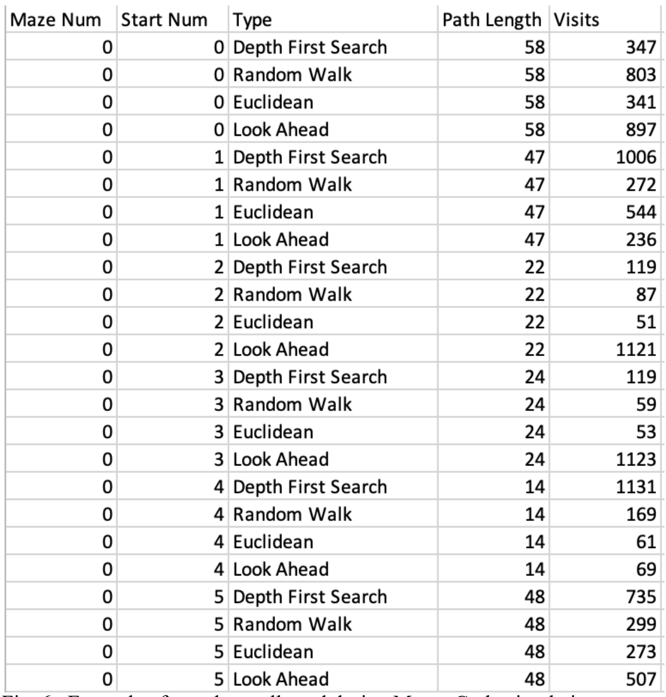
Based on this look ahead information, it is hypothesized that the performance of this heuristic will be improved over that of the standard Euclidean Distance heuristic. A full discussion of the results is given in subsequent sections. The pseudocode for the Euclidean distance with Look Ahead is given below in Fig. 5.

```
FUNCTION lookAheadSolve()
WHILE the CURRENT CELL is NOT EQUAL to the GOAL CELL
        ADD the CURRENT CELL to the SOLUTION PATH
        MARK the CURRENT CELL as VISITED
        DETERMINE POSSIBLE MOVES FROM CURRENT CELL
        IF there are NO POSSIBLE MOVES available:
                REMOVE the CURRENT CELL from the SOLUTION PATH
                SET the CURRENT CELL as next in SOLUTION PATH
                REPEAT the WHILE LOOP
        ELSE IF there are POSSIBLE MOVES available:
                CHOOSE the FIRST POSSIBLE MOVE
                SET the CURRENT CELL from LOOK AHEAD RESULT
                REPEAT THE LOOP
        END IF
END WHILE
RETURN solutionPathStack
END FUNCTION

FUNCTION lookAheadResult()
FOR all POSSIBLE MOVES
        CREATE new MINI AGENT from ADJACENT CELL
        CALL MINI AGENT TRAVERSE with DEPTH LIMIT
        RETURN THE ADJACENT CELL with CLOSEST TRAVERSE
END FOR
END FUNCTION


FUNCTION miniAgentTraverse()
SET DEPTH LIMIT
WHILE CURRENT CELL is NOT GOAL CELL AND CURRENT DEPTH is LESS THAN DEPTH LIMIT
ADD the CURRENT CELL to the LOOK AHEAD PATH
        MARK the CURRENT CELL as VISITED
        DETERMINE POSSIBLE MOVES FROM CURRENT CELL
        IF there are NO POSSIBLE MOVES available:
                REMOVE the CURRENT CELL from the LOOK AHEAD PATH
                SET the CURRENT CELL as next in LOOK AHEAD PATH
                LOWER CURRENT DEPTH by 1
                REPEAT the WHILE LOOP
        ELSE IF there are POSSIBLE MOVES available:
                SET the CURRENT CELL from SHORTEST DISTANCE
                INCREASE CURRENT DEPTH by 1
                REPEAT THE LOOP
        END IF
END WHILE
RETURN lookAheadStack
END FUNCTION
```

Fig. 5. Pseudocode for the Euclidean Distance with Look Ahead search heuristic.

IV. Experimental Results

The experimental results are given as a *.CSV file as shown in the Monte Carlo Analysis pseudocode. An example of the raw data is also given in Fig. 6. The data collected includes the maze iteration number, the start/finish iteration number, the heuristic type that performed the solve, the length of the solution path, and the total number of visits (note that this can be, and often is, much greater than the total number of cells in the maze itself; this is because the agent can visit a cell multiple times during back-tracking).



Maze Num	Start Num	Type	Path Length	Visits
0	0	Depth First Search	58	347
0	0	Random Walk	58	803
0	0	Euclidean	58	341
0	0	Look Ahead	58	897
0	1	Depth First Search	47	1006
0	1	Random Walk	47	272
0	1	Euclidean	47	544
0	1	Look Ahead	47	236
0	2	Depth First Search	22	119
0	2	Random Walk	22	87
0	2	Euclidean	22	51
0	2	Look Ahead	22	1121
0	3	Depth First Search	24	119
0	3	Random Walk	24	59
0	3	Euclidean	24	53
0	3	Look Ahead	24	1123
0	4	Depth First Search	14	1131
0	4	Random Walk	14	169
0	4	Euclidean	14	61
0	4	Look Ahead	14	69
0	5	Depth First Search	48	735
0	5	Random Walk	48	299
0	5	Euclidean	48	273
0	5	Look Ahead	48	507

Fig. 6. Example of raw data collected during Monte Carlo simulation.

During the Monte Carlo analysis there were a total of 5000 random mazes generated, and each maze had 10 random start/finish pairs. Each of these pairs, as stated in a prior section, have all four heuristics solve the maze.

Subsequent analysis of the raw data involved the computation of the competitive ratio for each iteration of each solver. Once the competitive ratio was calculated, the average of all values is taken, alongside the standard deviation. These values permit a comparison on equal footing, additionally, with a lower standard deviation value, it can be shown that the variation of performance varies between all of the heuristics as well. The processed results are given in Fig. 7 below.

	DFS	**Random Walk**	**Euclidean**	**Look Ahead (3)**
Average:	21.70	23.89	7.81	26.06
Std. Dev.:	30.52	37.09	9.55	41.59

Fig. 7. Processed results from Monte Carlo analysis. The values given are the competitive ratios for each heuristic. Note that the Euclidean average is the lowest by far. Indicating superior performance compared to the other three heuristics.

An interesting result is shown above. The Look Ahead heuristic performs much worse than even a random walk. The Euclidean solver performs much better than the other three. Additionally, the much lower standard deviation exhibited by the Euclidean solver indicates that this type of performance is much more commonplace than the indicated performance for each of the other heuristics. A discussion of possible causes and future efforts is made in the next section.

V. Advantages and Drawbacks

The hypothesis that the look ahead heuristic would outperform the Euclidean heuristic was clearly incorrect. As shown in Fig. 8, the average competitive ratio of the Look Ahead heuristic is almost three times that of the Euclidean heuristic. Also, the standard deviation of the look ahead heuristic is higher than even the Random Walk heuristic. While there was not much exploration done to determine why these results were so much beyond the other heuristics. However, it is likely due to the "mini-Agents" leading the agent itself astray. Currently, the agent only moves one cell at a time, perhaps changing the look ahead mini agents to move the agent to the full depth of the look ahead would help achieve a better result. Given that, one would expect a reduction of about a factor of 3 in the average. This is because the agent itself would have 3 times fewer opportunities to spawn mini agents that visit multiple cells and drive up the total visit count.

Given that the computational and memory footprint for the Look Ahead heuristic is so much larger than that of the Euclidean heuristic (in fact, it's more than 4 times as large), there aren't too many advantages to using this heuristic, but given its lackluster performance and extensive computational overhead, there are clearly a number of drawbacks.

A note of relevance though: notice the similarity between the DFS and Random Walk values. We see that the Random Walk performs less well compared to the DFS heuristic, but the differences are negligible when compared to the values in the Euclidean heuristic. This exercise serves to underline just how effective the Euclidean heuristic is when solving a randomly generated maze.

VI. Summary and Conclusions

In this exercise, the results clearly show the efficacy of good, admissible heuristics when shown in the context of an online search agent operating within a maze-type environment. The first two, non-intelligent, heuristics show a lot of wasted overhead, the Euclidean heuristic shows excellent performance, and despite high expectations, the look ahead heuristic performed abysmally by comparison. The ultimate invalidity of the look ahead heuristic is not stated here however, as there is plenty of work to understand where the heuristic is leading the agent astray and tweaking the algorithm to better inform the agent as to where is best to move. Note that all code and other artifacts can be found in an online repository hosted at https://www.github.com/mgarrettsisk/heuristicCompare.

References

[1] C. Hernandez, P. Meseguer "Improving LRTA*(k)," IJCAI-07, pp. 2312-2317

[2] V. Bulitko, G. Lee, "Learning in Real-Time Search: A Unifying Framework," Journal of Artificial Intelligence Research 25, pp. 119-157, Feb. 2006

[3] V. Bulitko, Y. Bjornsson, "kNN LRTA*: Simple Subgoaling for Real-Time Search, 2009

[4] S. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach," 3rd Edition, Pearson, December 1, 2009