Snake AI

David Schoder
Emmanuel Sacristan

December 17, 2015

Abstract

The game snake was implemented using a 2D grid and tested various search algorithms. The algorithms used include A*, breadth-first search, and depth-first search. Each algorithm was tested on a 20x20 grid. It was found that A* with a heuristic of absolute distance performed the best of all algorithms tested. Obstacles were added to the grid and the algorithms were tested again on a 20x20 grid. In this case, A* with a heuristic of absolute distance + Manhattan distance performed best. The AI could not complete a game of snake but further detail about possible solutions are discussed.

1 Short Description:

The game of snake has been around for a while now and due to the nature of the game can create some interesting problems to solve using search approaches from AI. The objective of the game is to eat the red apples that spawn randomly throughout the n x n grid which confines the snake. Whenever the snake eats an apple he will grow by one cell thus making his grid more difficult to navigate through as he tries to eat more of the apples. Since the apples spawn randomly and the body of the snake is constantly moving and growing, this creates an interesting problem to solve.

There have been some generic approaches to the problem that involve walking in a predetermined path in with the intent of eventually eating the apple. This in theory would work by taking a non-optimal path and moving the snake row by row in a repeated fashion until the game is completed. Taking this approach it is easy to see many cases where an apple is randomly spawned and the snake completely avoids the apple and just waits until he eventually comes around. A similar scripted solution would be to spawn the apples in a known pattern and adjust your snakes movement to anticipate a path ahead of time. In our case the apples are random and can appear anywhere and its new destination will not be known until the current apple is consumed.

The two above cases are not truly a search or AI like approach which reacts according to the changes in its world, its oblivious to whats going on around it and simply patrolling in a known pattern. If there were static objects set down or potentially another snake contending for apples these approaches would not work you would need search approach to solve this problem to be able to react to these kinds of problems in the snake world.

Ideally we would like to solve the problem using a search approach such as A* with a heuristic to enable optimal movement thus completing the game in a much faster time rather than mindlessly moving in a scripted fashion and randomly consuming the apples. But problems arise when simply taking the most optimal path towards the apple with every spawn since the body of the snake grows thus making it difficult to have an escape route after having eaten an apple. Later we discuss some possible solutions to this problem.

Some interesting twists that we added were some static obstacles making a maze like grid and potentially up to 4 snakes playing on the board at the same time. This creates more contention as the snakes bodies grow and they need to avoid each others bodies as well as their own in the search for new apples to consume.

2 Background

The problem of finding the shortest path between two given nodes has literature reaching back to the mid 1950s. Normally one would represent a grid based graph

as a two dimensional grid where each nodes position is given by a set of coordinates (x,y). Each nodes neighbors are given by the coordinates (x,y-1), (x,y+1), (x-1,y), and (x+1,y). There have been many different approaches attempting to minimize time and space efficiency when traversing such a grid.

A list of common search algorithms used to determine a path between two nodes:

- **B**est-first search
- **B**ellman-Ford algorithm
- **B**readth-first search
- **D**epth-first search
- **D**ijkstras algorithm
- **A*** search

Depth-first search [1] works by traversing as far as possible along a branch before backtracking and searching other branches. The path it finds is not necessarily optimal. If a list of previously visited nodes is not kept, depth-first search can end up in an infinite loop. The time complexity of O(V+E) and the space complexity is O(V).

Breadth-first search [2] works by adding all neighbors of a given node to a queue. Then it removes a node from the queue and adds all of its neighbors to the queue. This continues until all nodes have been explored or the destination has been found. If the cost per edge is uniform, breadth-first search will find an optimal path. Both the time and space complexity of are O(V+E).

Dijkstras algorithm [3] (a.k.a. uniform cost search) works by expanding the node with the shortest accumulated path. The algorithm finds the shortest path to all accessible nodes from a given node. The time complexity is O(V+ElogE) and the space complexity is O(V^2). This algorithm does not work with negative weight edges.

Bellman-Ford algorithm [4] works by relaxing each edge in the graph V times, where V is the number of vertices. It will find the shortest path to all accessible nodes from a given node. Unlike Dijkstras algorithm, this algorithm works with negative weight edges. It will also detect negative weight cycles. The time complexity is O(V*E) and the space complexity is O(V).

A* [5] is one of the most common pathfinding algorithms. It is a type of best-first search that works by using a heuristic to determine how promising a node Looks. If a node is not promising, it will not be expanded. This reduces both time and space efficiency over simple breadth-first search. The time complexity is O(E) and the space complexity is O(V). This is faster than most other search algorithms, but requires some knowledge of the search space to create a helpful heuristic.

Pathfinding is used extensively in the video game industry. An example being an enemy soldier finding the shortest path to a soldier under the players control. Generally, a rectangular grid is superimposed over an area and a search algorithm is used to find the shortest path. Each grid location has a weight associated with the cost to travel to that location. The algorithm generally used is A*. Most modern games are played in real time. This introduces a problem for pathfinding since the algorithm used must complete in a very short amount of time so as not to hinder the player or reduce the quality of the experience.

There are some interesting approaches to solve jittery ai movement when representing the grid as tiles. One approach is using square tiles that have four sides and changing the size of the tile which affects the granularity. The smaller the tiles the more you need to make up a grid, but the benefits in having a finer granularity is smoother and more realistic movement [6]. If the tiles are too large you get unrealistic movement where the object may look as though it's floating or teleporting along the board. The trade off with decreasing the size of the tile for more realism is the need for much more computation. The search space grows but the window you have for calculating a path stays the same which makes it difficult to come up with a timely solution.

Another approach that can be used to solve this unrealistic movement is utilizing hexagonal tiles instead of squares which now yield you a larger span of movement instead of just N,S,W,E you can now move in six directions. By making this change you can generate more realistic movement along the grid, but optimizations are needed to avoid having a large search space. Now you would be able to leverage larger tiles and still have realistic movement at the cost of some implementation details [6]. Although these are some interesting solutions, considering how fast CPUs are now we decided to use square tiles for our implementation. Also the complexity of our problem isn't on the scale of modern games like Diablo or Starcraft which has a lot more moving parts and obstructions.

3 Representation and Algorithms

We had decided to write the game of snake from scratch leveraging the SFML library in order to have a graphical representation of the game. We represent the grid as an n x n array of squares which contain attributes such as its color. The colors are defined as such:

- **G**ray is an empty cell
- **G**reen is the head of the snake
- **B**lack is the body of the snake
- **B**lue is an expanded node but does not lay on the solution path
- **Y**ellow represents cells that lay on the solution path

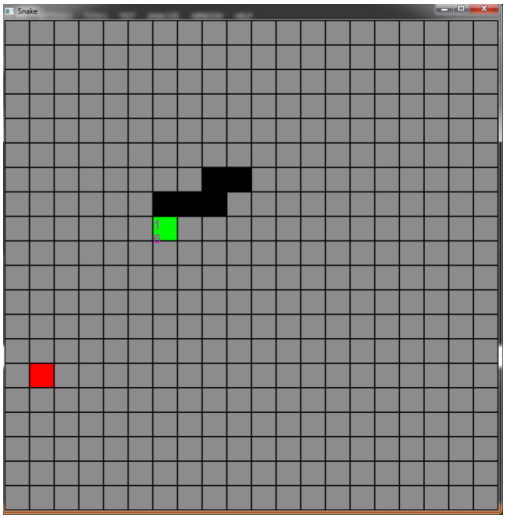

Figure 1: The space representation

- **R**ed is an apple on the grid (we only ever spawn 1 apple at a time)
- **M**agenta is an obstical that the snake cannot enter

The Snake is defined as its own class, which contains his length and a linked list of coordinate pairs that make up the body of the snake as it becomes superimposed onto the grid. When an apple is consumed and the process for growing the snake occurs we add a new node at the coordinates of the apple which becomes the new head. The previous head now becomes part of the body and pointers are adjusted accordingly naming the apples location as the new head. When moving the snake about the grid and not consuming an apple we simply remove the tail node from the linked list and repurpose it as the head so as to avoid the need for moving each coordinate pair which represents the body.

Now after having described our Grid and Snake representations we will go over a high level flow of the program as it executes. In the main function of our solution we create the Window which comes from the graphics library which generates the GUI seen above (Figure 1). The Grid is created and colored Gray to represent an empty board, if obstacles are generated they are placed after the grid is generated. The apple is then generated at random and painted red followed by the creation of the snake which we default to coordinate (0,0) for a 1 snake game this cell is painted Black.

Once the init steps are completed we enter the game loop which generates a FPS value then applies the intended algorithm (A*, BFS, DFS, RAND) on the snake generating a movement scheme. The grid then gets repainted and redrawn onto the window.

The approach we took involved running different algorithms in order to generate movements for the snake. We compared the performance of the different algorithms and found out that we would need a safe move in order to help us get out of situations where the snake traps himself with his own body. Simply running a vanilla version of these algorithms was not sufficient. We will describe in more detail some possible solutions in later sections that we had not been able to implement.

4 Experiments and Results:

For our setup we decided to run our AI on a 20x20 grid using the following algorithms:

- **A*** (with different heuristics)
- **B**readth-First-Search
- **D**epth-First-Search
- **R**andom

The two performance measures used were the number of Nodes Expanded and the length of the Snake at the time of death. There were 10 different RNG seeds used across the 4 algorithms in order to generate similar patterns for the apple spawns. There was some variance due to different paths being taken by the snake because its position would vary per algorithm making it impossible for certain apple spawns to fall on the same coordinate. When an apple would be generated on a coordinate with an obstruction we recalculate a new spawn location for the apple.

Figure 13 below depicts the amount of nodes expanded per seeded run using different algorithms. Figure 14 below depicts the length of the snake per seeded run using different algorithms.

Below we have some snap shots of the snake moving through the grid, the blue cells are nodes it had expanded in the process of figuring out the path to take. The yellow cells are the path it has decided to take for that iteration.

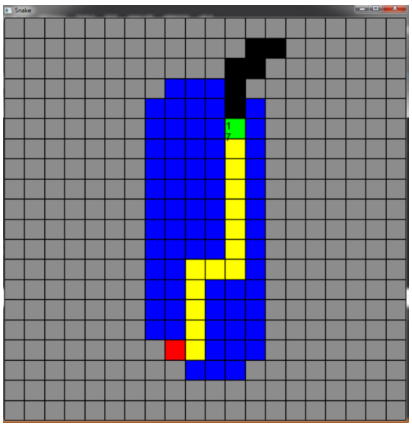

Figure 2: A* with Manhatten distance heuristic.

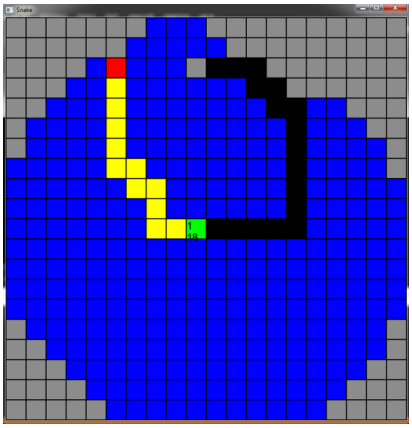

Figure 3: Breadth-first search.

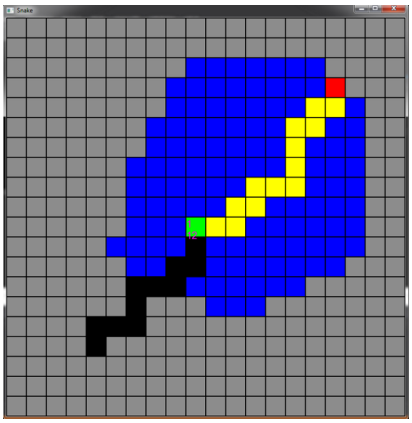

Figure 4: A* with absolute distance heuristic.

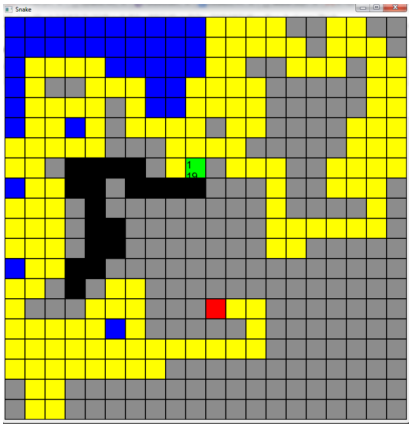

Figure 5: Depth-first search.

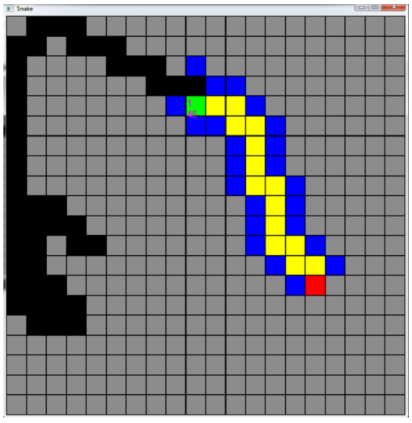

Figure 6: A* with Manhatten + absolute distance heuristic.

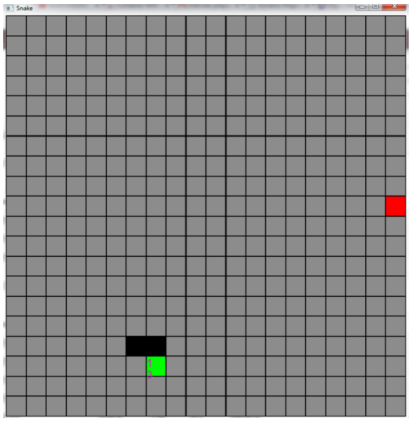

Figure 7: Random movement. No search algorithm used.

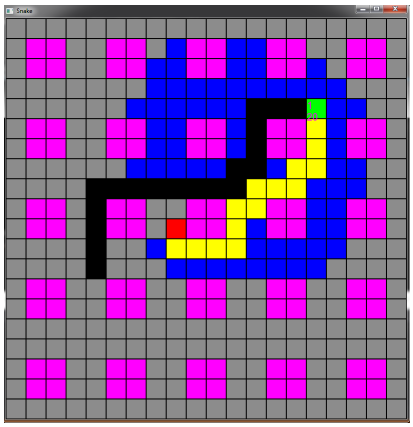

Figure 8: A* with Manhatten distance heuristic with Obsticles.

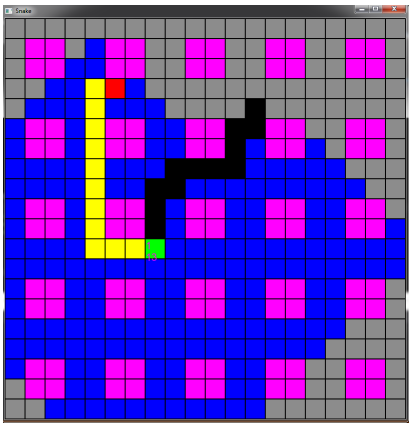

Figure 9: Breadth-first search with Obsticles.

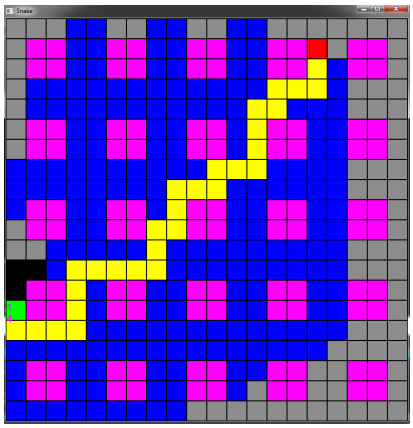

Figure 10: A* with absolute distance heuristic with Obsticles.

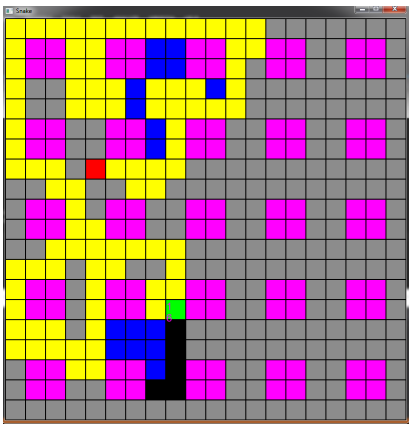

Figure 11: Depth-first search with Obsticles.

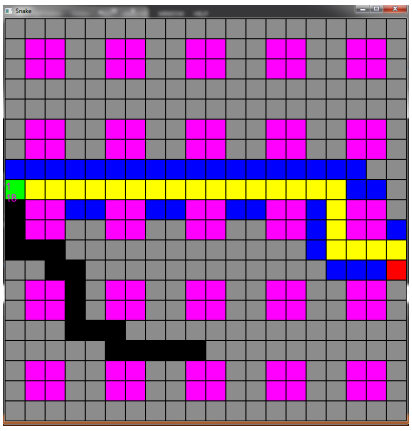

Figure 12: A* with Manhatten + absolute distance heuristic with Obsticles.

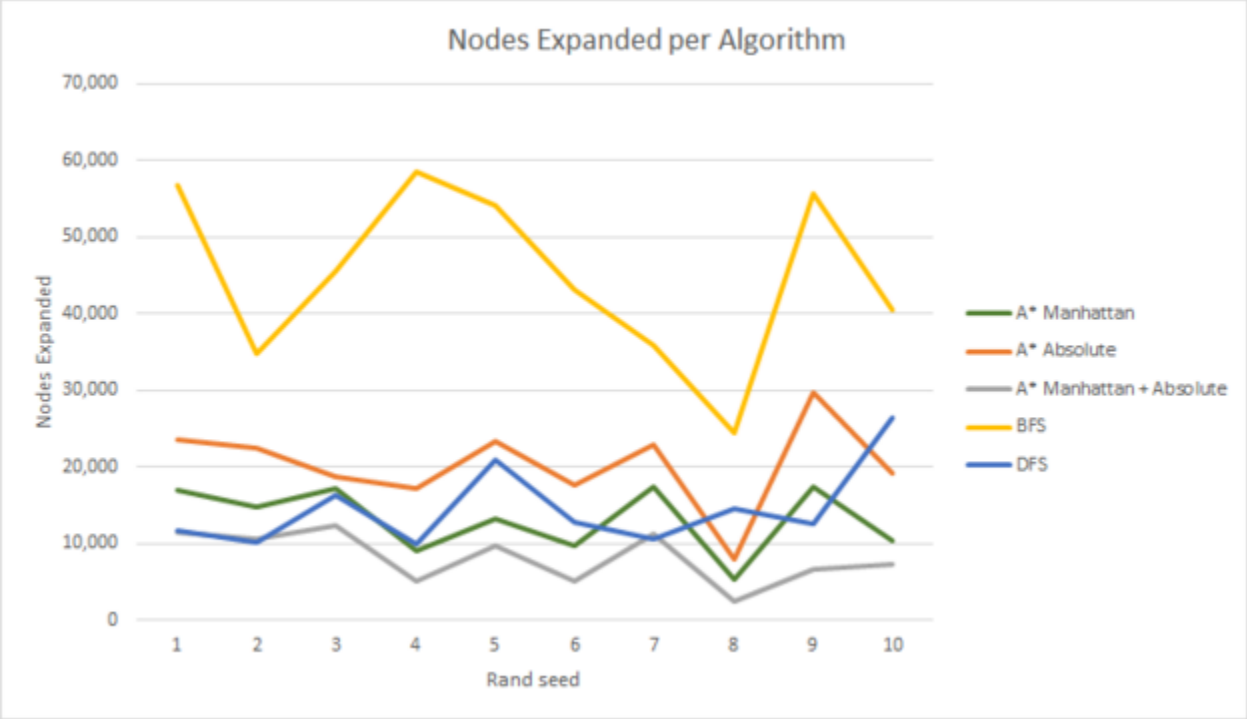

Figure 13: Number of nodes expanded per algorithm

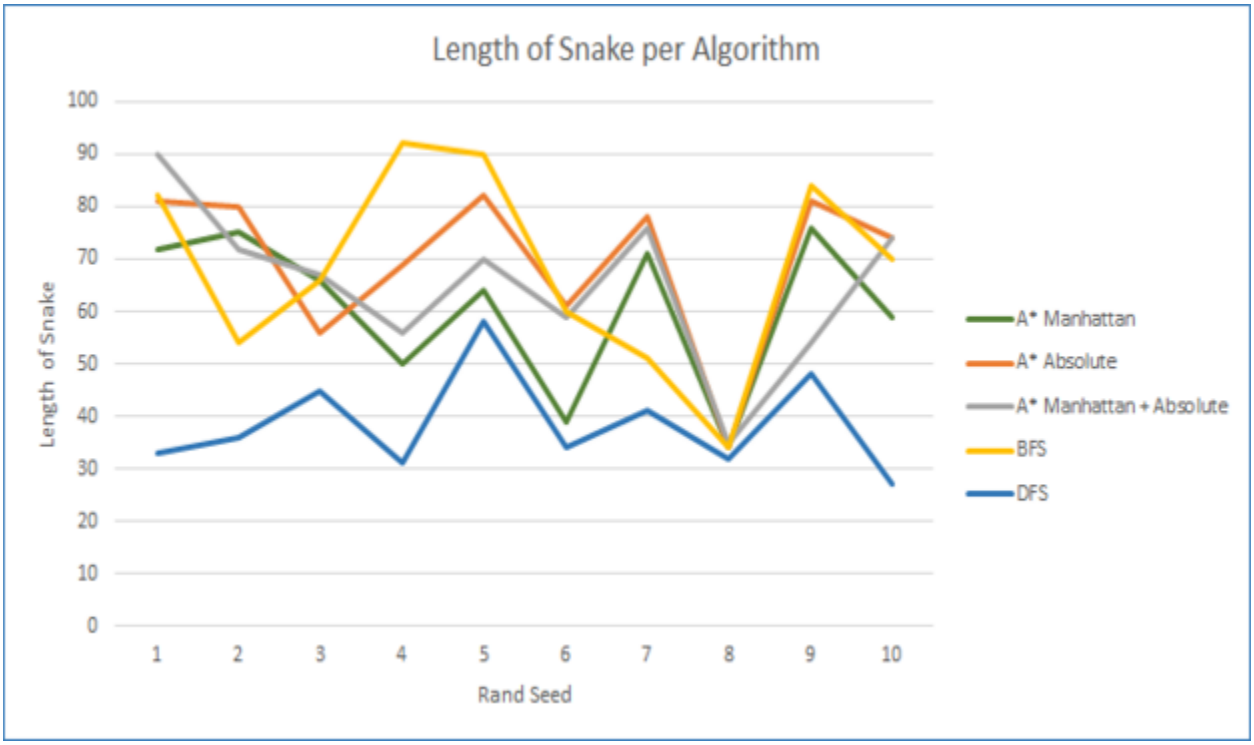

Figure 14: Length of snake per algorithm

5 Analysis:

The data in Figure 13 gives us a good understanding of the node expansion effort used in order to find paths to the apple. Breadth-first-search expanded the most nodes compared to the other algorithms tested but was producing optimal paths as its solution. The A* algorithms had an interesting turn out in terms of node expansion. When using absolute distance which would produce a smaller heuristic compared to Manhattan, it had expanded more nodes on average and selected different optimal paths throughout the runs. Since Manhattan distance provided a larger heuristic by giving the length of the two sides to the goal instead of the hypotenuse for absolute distance this enabled us to find an optimal solution without the need of expanding more nodes. After having seen this behaviour we decided to combine the two heuristics and create a new one which was the summation of the two heuristics, since both are admissible the summation of the two is also admissible. The runs performed using the combined heuristic expanded the fewest nodes while still finding optimal paths. DFS surprisingly expanded a similar amount of nodes as A*(not the combined heuristic version) but found very lengthy paths. We feel this is due to the size of the grid only being a 20x20. Had the grid been much larger we suspect DFS would have expanded more nodes on average causing its nodal expansion to be larger than A*.

Although nodal expansion is important in deciding whether an algorithm can come to a solution in a timely fashion. The length of the snake at the end of the program shows how successful the AI was navigating the grid. In Figure 13 we see a breakdown of all the snake lengths for each algorithm. The top performers were A* (Absolute Distance Heuristic) and BFS with an average snake length of 69.6 and 68.4 respectively. Whereas DFS with an average length of 38.5 was the worst performer aside from rand move, which would take paths that were more scenic. This caused the snake to wrap its body around the apple before consuming it which creates a more difficult path when moving to the next apple that spawns. A* and BFS in choosing more optimal paths could successfully consume the apples in less movements and create less entanglement around the apple, but eventually we reach a threshold where the length of the body becomes an issue as we envelope ourselves. In order to try and solve this entanglement issue we would switch into safe move mode which would occur when an algorithm could not find a path to the apple due to being confined to a subset of the grid because of its body. What safe move would do is randomly move into valid empty spaces in the confined area and check to see if it eventually opened a path to the apple while buying time. This did not completely solve our problem and we would still end up entangling ourselves with no way out, if we had approached the apple and consumed it while still leaving ourselves a relatively safe path to the next would have been ideal. We go into some speculation as to how we this could have been done differently in the conclusion.

The last set of experiments ran laid static blockades uniformly throughout the grid. As can be seen in figures 8-12 the magenta blocks represent these blockages. We had done this in hopes of creating an interesting problem for the snake to solve.

After gathering data it seems all of the algorithms used in the previous experiments performed noticeably worse. The length of the snake had dropped by roughly 15-25 nodes compared to the vanilla grid. Less nodes were expanded compared to the first set of experiments. This was due to the blockages leading to dead ends much quicker than before. Since there were blockages this this shrunk the amount of valid cells, so what began as a 20x20 grid really behaves as a 15x15 grid. By doing this we reduce the max length the snake can attain assuming it played the perfect game. The snake also had issues maneuvering through the narrow hallways which caused our entanglement issue to happen sooner than the vanilla grid. As the snake tried to find the shortest path to the apple he was more prone to closing himself off by wrapping his body around a blockade with no escape. Securing a sufficient escape route or finding a smarter way to approach the apple becomes more difficult with blockages introduced.

6 Conclusion:

The approach we had taken in implementing the Grid and snake along with the algorithms led to interesting findings. We learned that simply finding the shortest path when trying to maneuver the snake around the grid to the apple would not provide ideal results. By leveraging the different algorithms such as A* (with different heuristics), BFS, DFS and Rand we were able to get a good idea of the snake's movement patterns and which approaches were more advantageous. A* with a heuristic of Absolute distance performed the best in both experiments in regards to the snake's length. BFS performed almost as good on average but had a much larger nodal expansion which would require more processing power. It had been interesting to see that A* with the combined heuristic had the least nodes expanded and still performed reasonable well in terms of the snake's length. None of the above algorithms dealt with the issues the game of snake brought up as the snake began to grow.

Although we tried to solve the entanglement issue by moving around in the empty spaces randomly until a clear path opened we would still eventually fail. Although we extended the life of the snake we still needed a better approach. Instead of calculating an optimal path and taking it to the apple we would do some post processing. We would look at the state of the grid and the snakes body after having theoretically eaten the apple to see if we have left ourselves a path with enough empty spaces for our body to move out of the way. If we do not see enough contiguous empty cells we will opt to move in a predetermined sweeping motion and fill in the cells with our body and minimize the amount of pockets created. While doing so we will still calculate a new optimal path and see if we have better chances of making an escape when entanglement occurs after eating the apple.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540549.

[2] Lee, C. Y. (1961). "An Algorithm for Path Connections and Its Applications". *IRE Transactions on Electronic Computers.*

[3] Dijkstra, E. W. "A Note on Two Problems in Connexion with Graphs" *Numerische Mathematik* (1959): 269-71. Print.

[4] Bellman, Richard "On a routing problem." *Quarterly of Applied Mathematics* (1958): 87-90. Print.

[5] Hart, Peter et al "A Formal Basis for the Heuristic Determination of Minimum Cost Paths" *Stanford Research Institute* (24 November, 1967): 101-07. Print.

[6] Yap, Peter "Grid-Based Path-Finding" *Advances in Artificial Intelligence* (28 May, 2002): 44-55. Print.