**Assignment 2: Planning, Games**

University of Illinois at Urbana-Champaign

Spring 2019 CS 440/ECE 448

Heting Fu                        hfu6                       Section Q

Yuhao Liu                 yliu167                  Section Q

Yuhao Min                    ymin6                    Section Q

Submitted on: February 25th, 2019

**Section I: CSP**

The pentomino problems can be considered as an Exact Cover Problem where all the different placement of each pentomino tile at different orientation is a subset of the entire board. In other words, we can choose some of the pentomino tiles from the subset and cover the entire board.

A mathematic representation can be a matrix where each row represents a subset. Each column of the matrix represents a particular cell in the matrix. Values can be assigned to the matrix to identify the existence of tiles in the cell of the board. In this problem, “0” was chosen to represent an empty cell, while positive integer was used to represent the occupied cell.

In addition, more columns need to be added to specify the tile identity since they can only be used once. A “1” was added at the ith column of the matrix. An illustration of the matrix is shown in the following figure.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Represent the cell occupied by the tile | | | | | Represent the identity of the tile | | |
| 1 | 1 | … | 0 | 0 | 1 | … | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| … | | | | | … | | |
| 0 | 0 | … | 12 | 12 | 0 | … | 1 |
| **Figure 1.** An illustration of the problem representation. | | | | | | | |

To populate the matrix, all orientations of a tile are explored. For each orientation, all possible position of the tile is tested. The cells that are occupied by the possible placement are marked with a positive integer, and the entire board is flattened. The identity of the tiles is appended at the end of the placement 1-d array. This procedure is performed for all possible placement. For the 6x10 board, there are in total of 72 columns and 2057 rows.

To find a collection of the subset to satisfy the problem, we need to find rows in the matrix such that in the new matrix formed by stacking the rows, each column has only one positive integer. Algorithm X, proposed by Knuth, is effective in solving the problem. This is a recursive, back-tracking algorithm. Each time the function is called:

1. It first selects a column with the smallest number of positive integers. This column is deleted.
2. For every positive integer element in the column, it will delete the row of such element and add the row index to the solution.
3. For each positive integer in the delete row, the column such element is in will be deleted.
4. Lastly, rows contain a positive integer element in the deleted column will also be deleted.

The recursive call terminates successfully when there is nothing to be deleted.

To back-track, the algorithm will simply “undo” the deletion and choose another row with a positive integer to delete (step 2).

Each time the algorithm is run, constraints are applied in order to yield the solution. The deletion operation in the algorithm is essentially applying the constraints to reduce the number of possible solutions. Once a row is chosen (step 1), all other rows intersecting (have positive integer element) with the chosen row should not be in the solution. Therefore all such rows are deleted, and the size of the matrix is now smaller.

The heuristic used in the algorithm is the number of positive integers. The column with the smallest number of positive integers is chosen first since essentially there will be fewer possible outcomes. In other words, the back-tracking tree structure is smaller, making the running time faster.

A dancing link (DLX) data structure was used to represent the sparse matrix. This is a double-linked, four-directional, circular mesh of nodes. The manipulation of the pointers at each node made the operation very easy and fast.

**Section II: Algorithms**

The heuristic used for single dot A\* and greedy BFS is Manhattan distance. In the maze, a optimal path from an agent to a goal will be a clear path, unobstructed by any wall, between them. Since in the maze, the agent can only move unit distance every time and in at only the x or y direction. The optimal path must have the distance equal to the sum of the absolute difference between the x coordinates and the absolute difference between the y coordinates, which is the Manhattan distance between the agent and the goal. Any path with a wall will increase the path length. That is, the Manhattan distance will never overestimate the cost of reaching the goal.

For multiple dot A\*, our heuristic is based on the Manhattan distance. Initially, a priority queue is used to store the Manhattan distance from the start to all the goals. The agent will find the goal with the smallest Manhattan distance. At every step, we will update the list with the Manhattan distance from the current node to all the goals. If at the current node, another goal becomes the closet goal, the agent will switch from the current goal to the new goal. Since Manhattan distance is admissible and we update Manhattan at every step, our heuristic will never overestimate the cost of going to the next goal. That is, our heuristic is admissible.

**Section III: Results (Basic Pathfinding)**

All screenshots can be found in the appendix. The solution costs and the number of expanded nodes are shown in Table 1.

**Table 1. Results of all algorithms on all mazes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Maze** | **Solution costs** | **Expanded states** |
| BFS | mediumMaze.txt | 111 | 646 |
|  | bigMaze.txt | 183 | 1286 |
|  | openMaze.txt | 52 | 559 |
| DFS | mediumMaze.txt | 175 | 374 |
|  | bigMaze.txt | 537 | 987 |
|  | openMaze.txt | 252 | 278 |
| Greedy | mediumMaze.txt | 147 | 348 |
|  | bigMaze.txt | 277 | 457 |
|  | openMaze.txt | 70 | 96 |
| A\* | mediumMaze.txt | 111 | 377 |
|  | bigMaze.txt | 183 | 1316 |
|  | openMaze.txt | 52 | 613 |

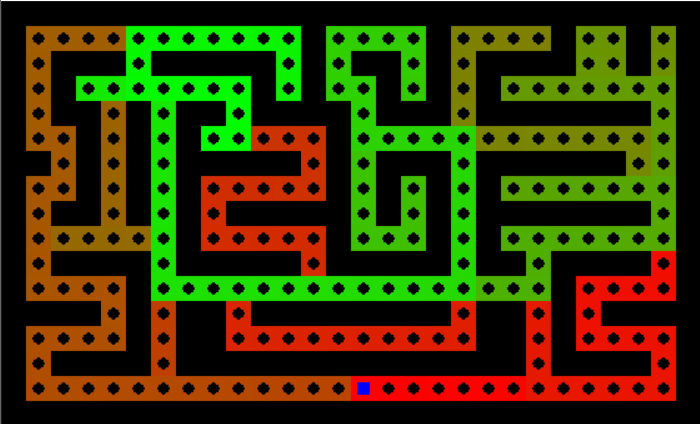
**Section IV: Results (Search with multiple dots)**

All screenshots are shown in the appendix. The solution costs and the numbers of states expanded are shown in Table 2. Although our heuristic is admissible. Our code fails to yield the optimal path. Possible sources of the error are potential edge cases which we have not considered, wrong calculation of the current cost and wrong path retrieval during back tracing.

**Table 2. Results of A\* algorithm for multiple dots**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Maze** | **Solution costs** | **Expanded states** |
| A\* | tinySearch.txt | 126 | 34 |
|  | smallSearch.txt | 327 | 196 |
|  | mediumSearch.txt | 556 | 363 |

**Extra Credit:**

We used a BFS based search strategy that does not guarantee an optimal solution but will always find a solution. The algorithm runs BFS from the start of the maze. Once it reaches an objective in the objective array, the objective is popped from the array, and another round of BFS starts at the objective location. The search ends when the frontier list (here we used a queue, FIFO) is empty or the objective array is empty. With this BFS based approach we were able to achieve a path length of 363 and states explored of 1041 on the bigDot maze.

**Statement of Contribution:**

All members mostly did all of the algorithm on our own different versions and compared the result to ensure the result is as required. But for A\* algorithm for single and multiple dots we worked collectively and ended up using Heting Fu’s version. Since for the Section I, all of our version of codes can run successfully with the predicted outcome, for BFS we also used his version. And for DFS and Greedy, we used Yuhao Ming and Yuhao Liu’s final version of codes respectively.

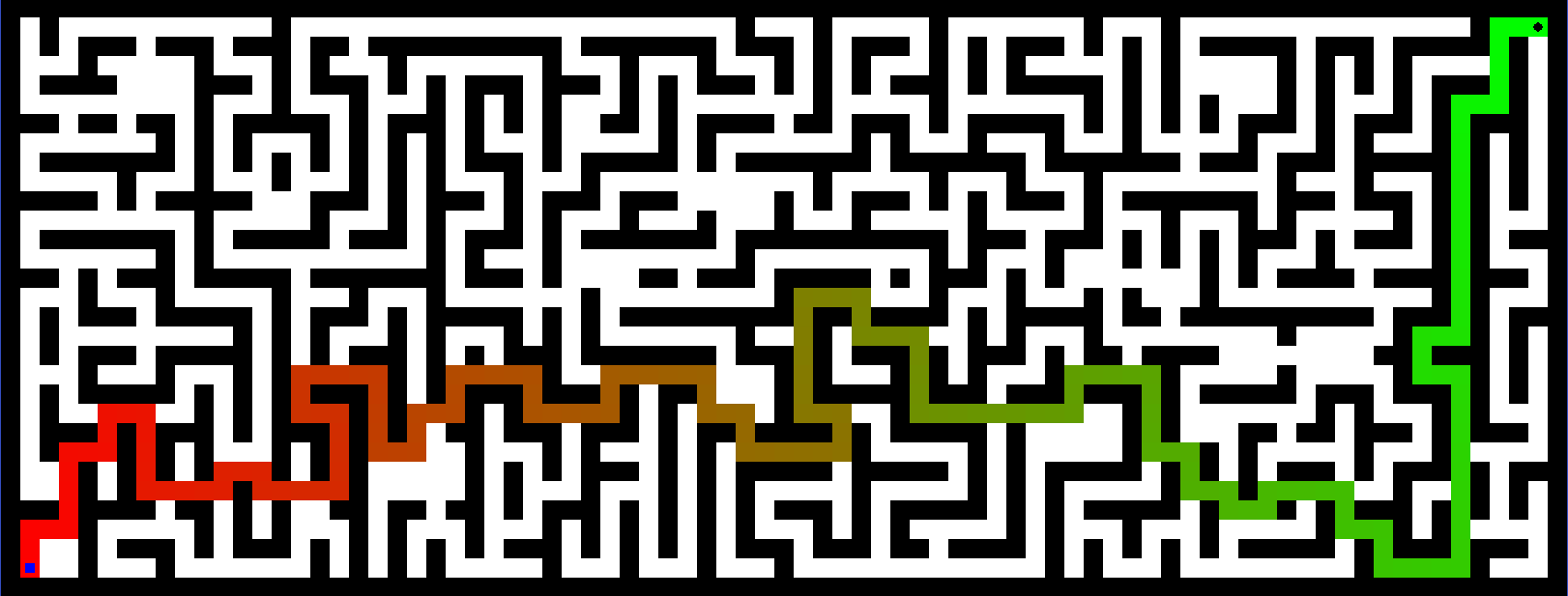
**References:**

1. Knuth, Donald E. "Dancing links." arXiv preprint cs/0011047 (2000).
2. Kapanowski, Andrzej. "Python for education: the exact cover problem." arXiv preprint arXiv:1010.5890 (2010).

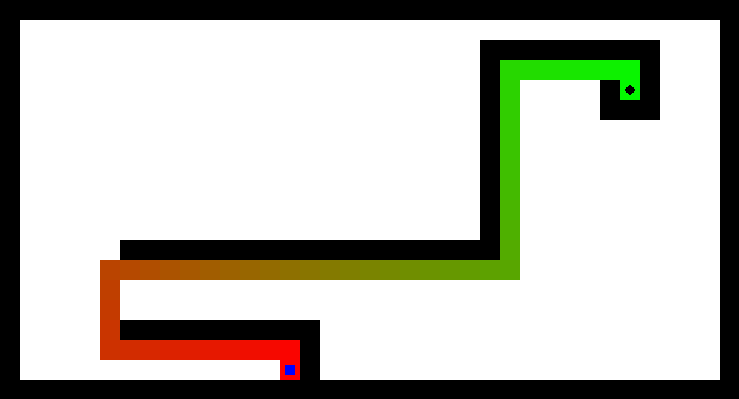
**Appendix**



**Figure 1. BFS on medium maze**

****

**Figure 2. BFS on big maze**

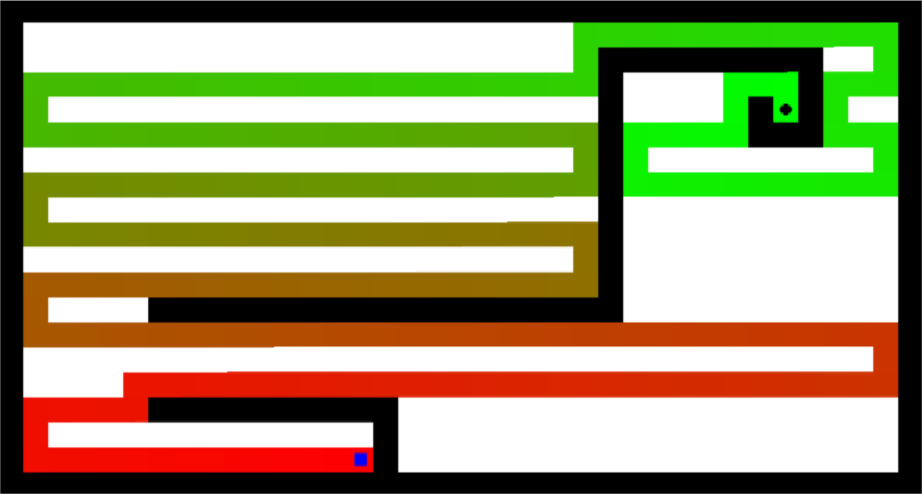
****

**Figure 3. BFS on open maze**

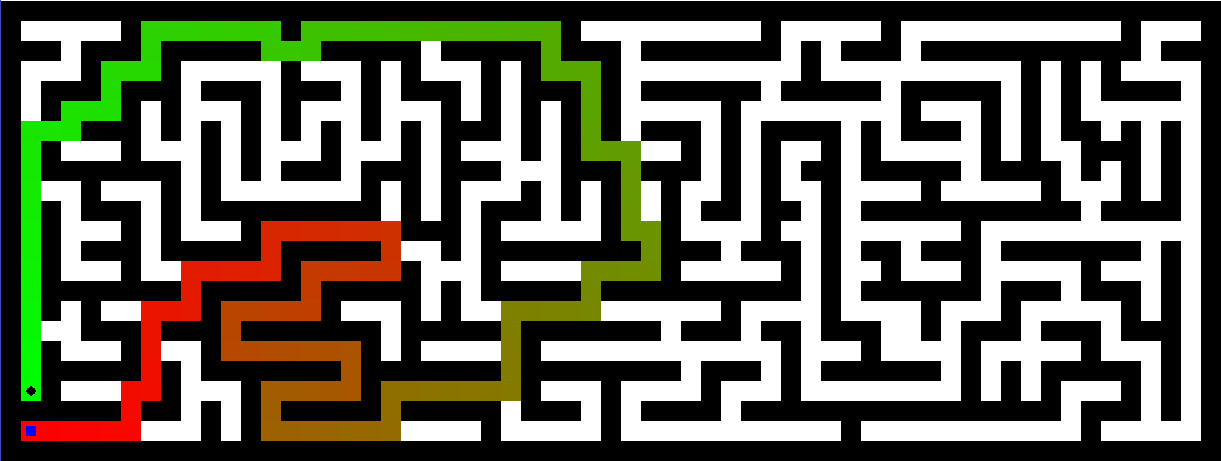
****

**Figure 4. DFS on medium maze**



**Figure 5. DFS on big maze**

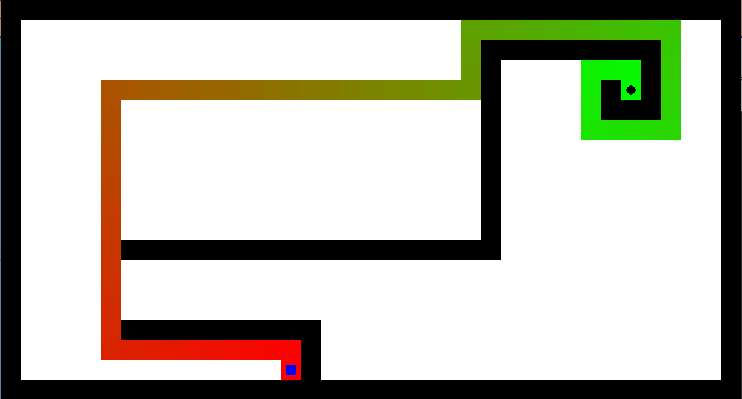
**Figure 6. DFS on open maze**

****

**Figure 7. Greedy on medium maze**

****

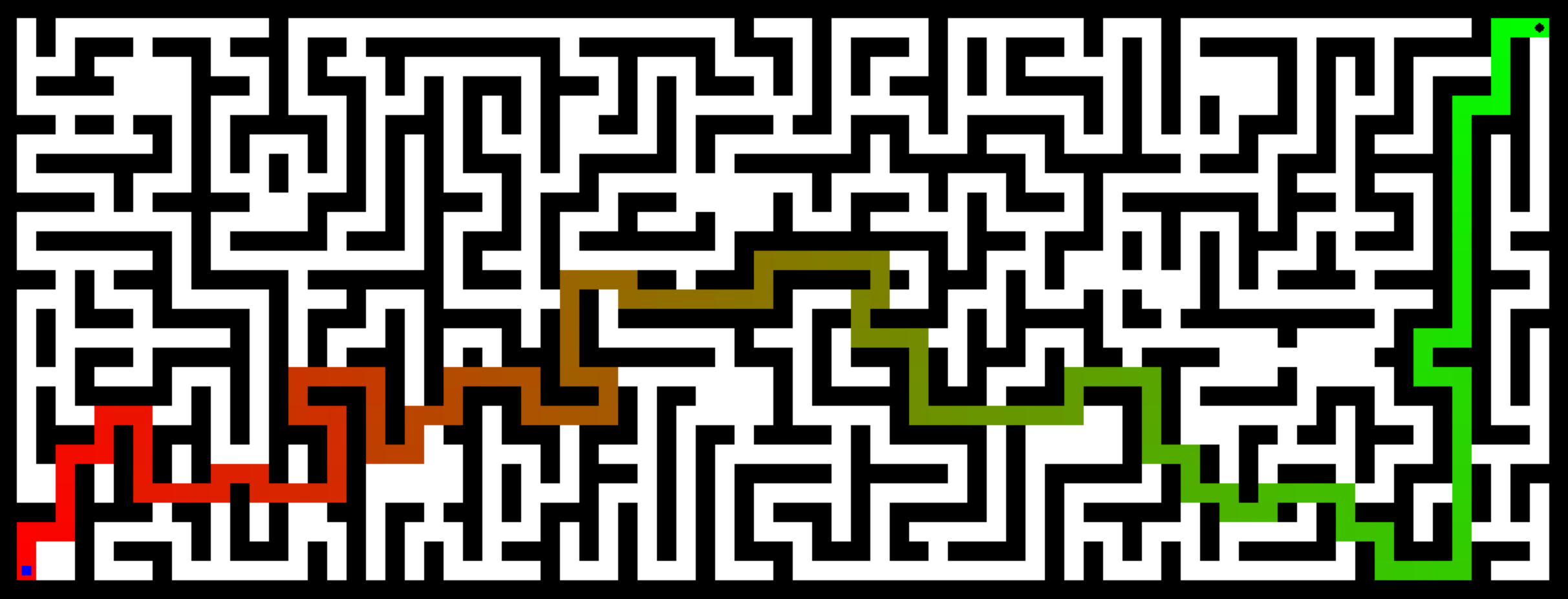
**Figure 8. Greedy on big maze**

****

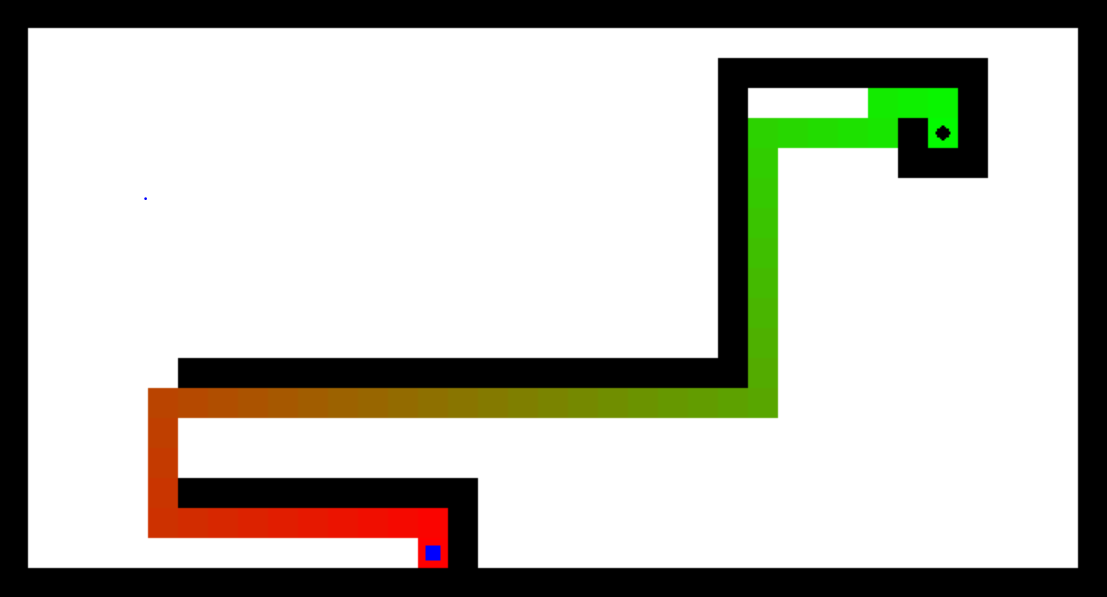
**Figure 9. Greedy on open maze**

****

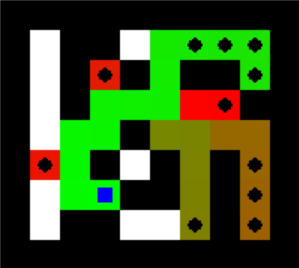
**Figure 10. A\* on medium maze**



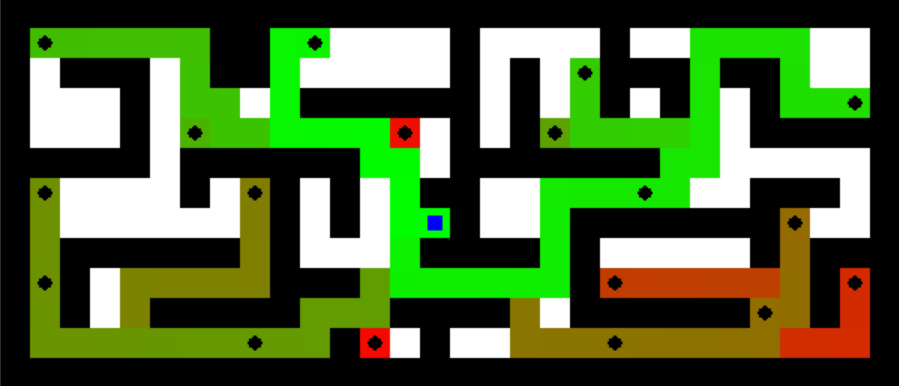
**Figure 11. A\* on big maze**



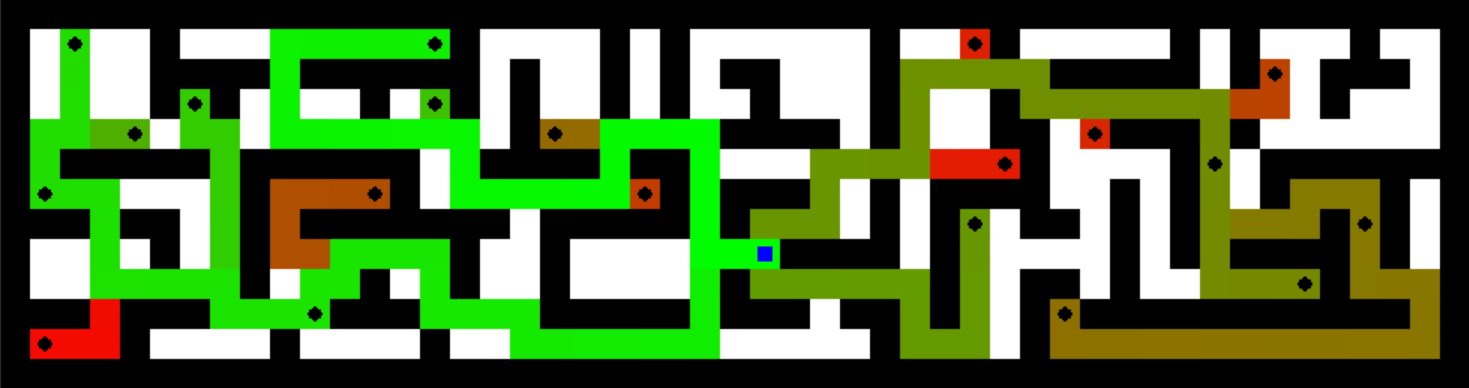
**Figure 12. A\* on open maze**

****

**Figure 13. A\* on tiny search**

****

**Figure 14. A\* on small search**

****

**Figure 15. A\* on medium search**