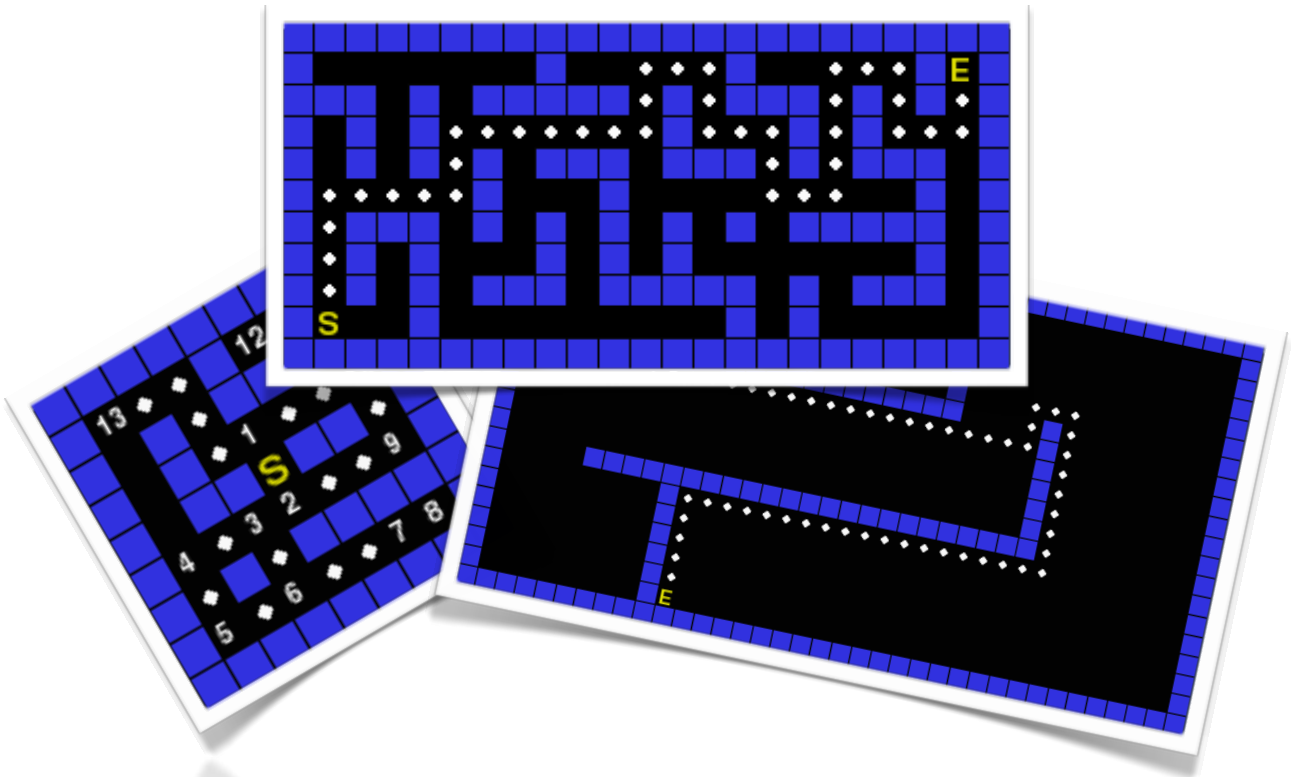
Maze Search

Assignment 1

Shashank Bharadwaj (bharadw3)
John Fritzel (jfritz4)

3 units

September 30, 2013



Foreword

For bonus points, we utilized the Pygame framework to draw different types of mazes. Although these images should be intuitive, I will briefly describe them: Start is always labeled with a yellow "S", and for mazes with a single goal, the end is always labeled with a yellow "E." The final path of the maze is labeled with white dots, and this path always includes the start and end points. For mazes with multiple goals, the number refers to the instance when the maze first found this goal. All of the drawing functionality currently resides in Maze.py.

Our entire project was done in Python (2.7), with Pygame (2.7). Each searching strategy has its own class, which should make inquiries into our implementation easier to find.

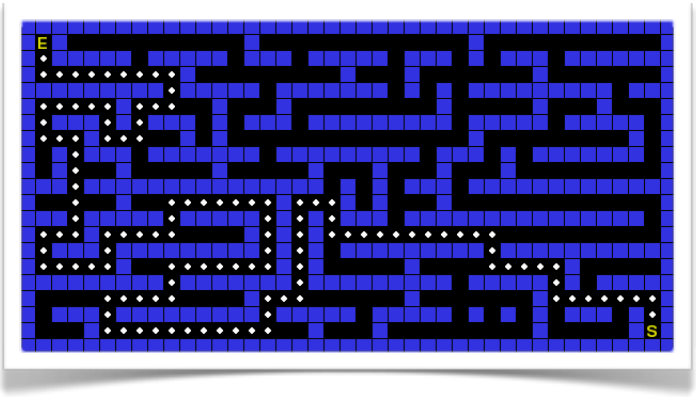
A statement of contribution: Both parties agree that there was equal contribution for this project. This is because nearly all of the code was developed together, in the same room, with one person typing and the other frantically trying to find bugs. For individual spotlights, the drawing functionality was originally programmed by John Fritzel who also programmed the Breadth First search. Shashank Bharadwaj programmed the Depth first Search and originally developed the idea for the heuristic in the informed searches.

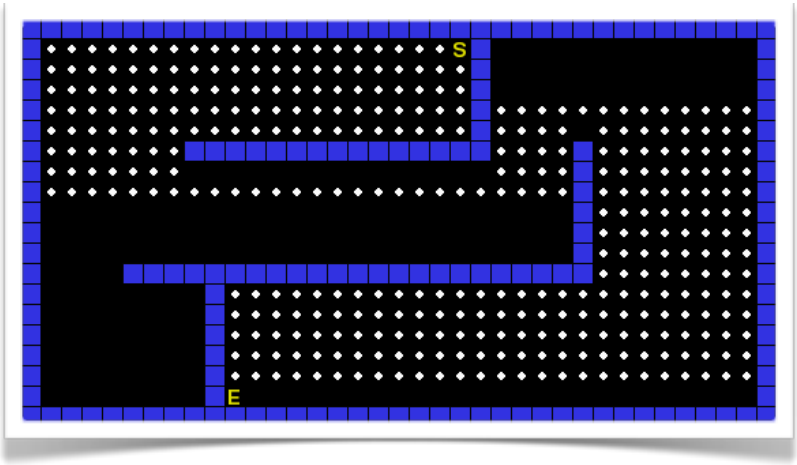
1.1 Basic pathfinding

Depth-first search:

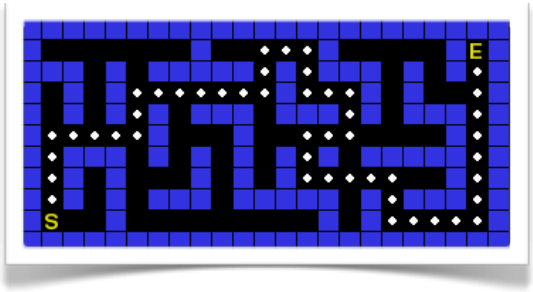
For Part 1.1, we implemented a typical depth first search. While programming this function, we noticed that the order of receiving children on the frontier resulted in different solutions. If, in our getChildren function (*see Maze.py*), we receive children in the order [left, right, down, up] (the default), we observe a path cost of 294 on Big Maze. Other arrangements resulted in different path costs, for example, [down, up, right, left] produces a solution path cost of 266, the optimal solution for this maze.

Medium maze:
Solution path cost: 124
Nodes expanded: 177
Max search tree depth: 124
Max size of the frontier: 16



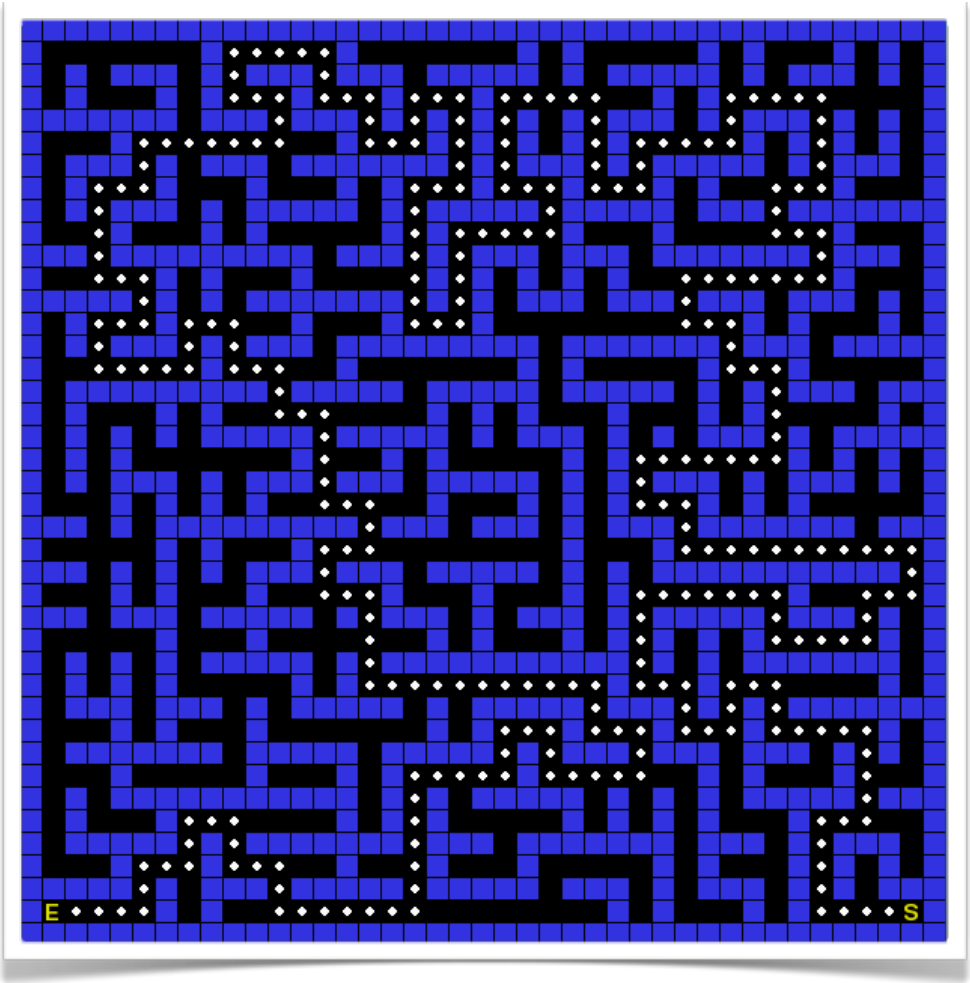


Open maze:
Solution path cost: 364
Nodes expanded: 420
Max search tree depth: 364
Max size of the frontier: 379



Small maze:
Solution path cost: 48
Nodes expanded: 77
Max search tree depth: 59
Max size of the frontier: 13

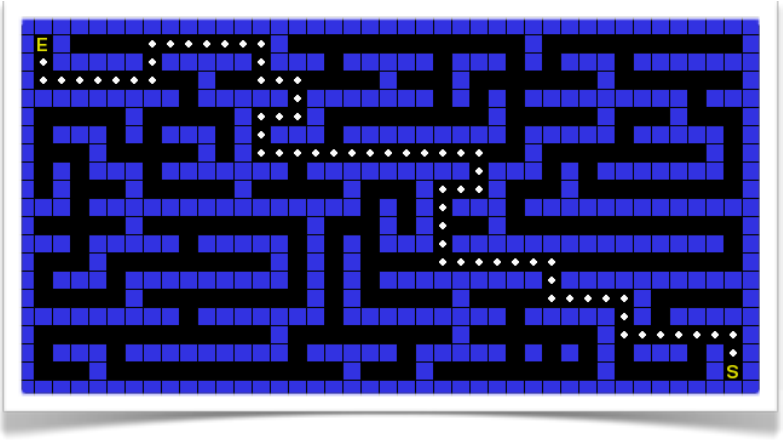
Big maze:
Solution path cost: 294
Nodes expanded: 537
Max search tree depth: 294
Max size of the frontier: 41

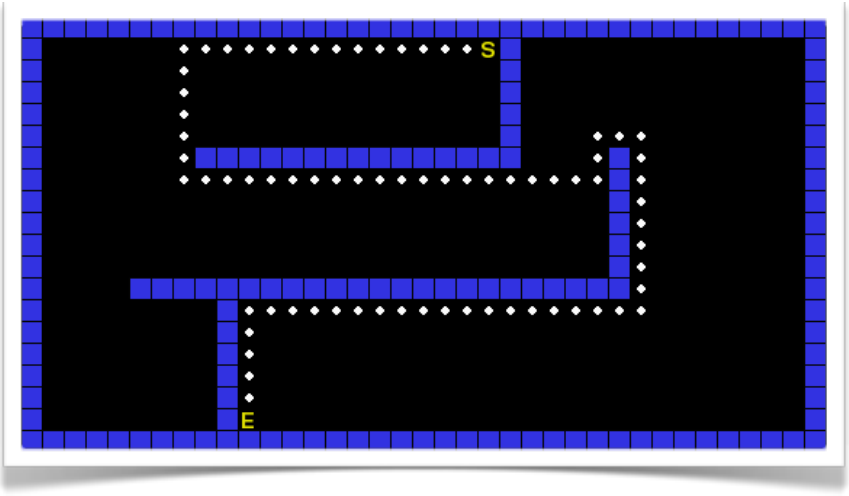


Breadth-first search:

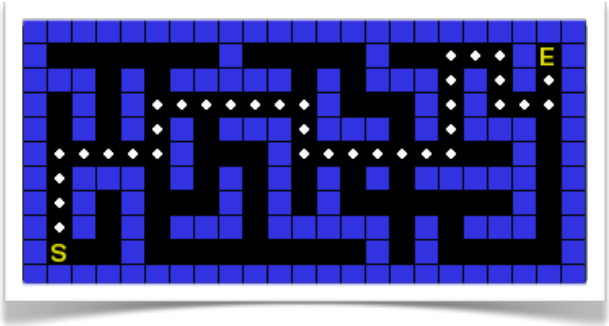
Our implementation of breadth first search found the optimal solution through the mazes, which was expected. The order of receiving children did not result in different final solutions, as it did in depth first. We found that the easiest way for us to implement this search in Python was to create a simple Node class (*see BreadthFirst.py*) that had its coordinate location and parent node. When the end was found, we simply followed a linked list from the the goal node through the parent node pointers to the start position, and drew the solution.

Medium maze:
Solution path cost: 68
Nodes expanded: 343
Max search tree depth: 68
Max size of the frontier: 12



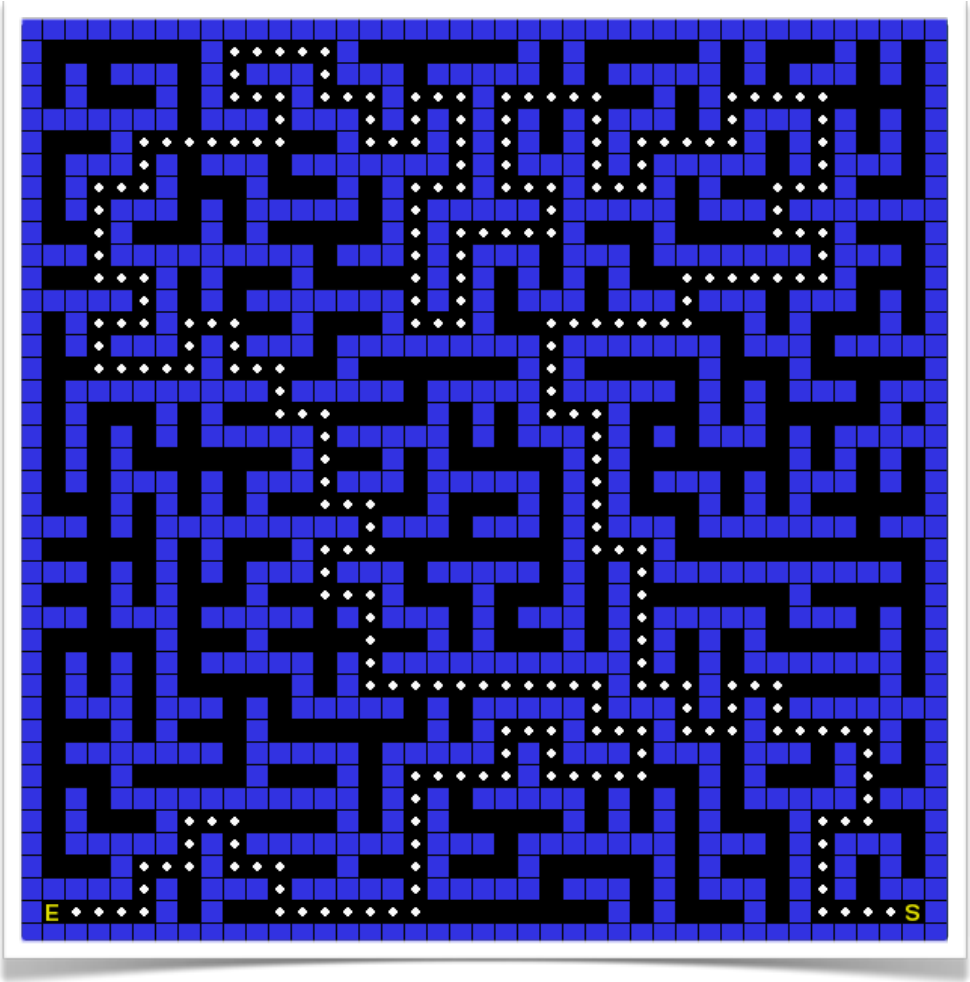


Open maze:
Solution path cost: 74
Nodes expanded: 574
Max search tree depth: 74
Max size of the frontier: 14



Small maze:
Solution path cost: 36
Nodes expanded: 113
Max search tree depth: 36
Max size of the frontier: 6

Big maze:
Solution path cost: 266
Nodes expanded: 795
Max search tree depth: 266
Max size of the frontier: 9



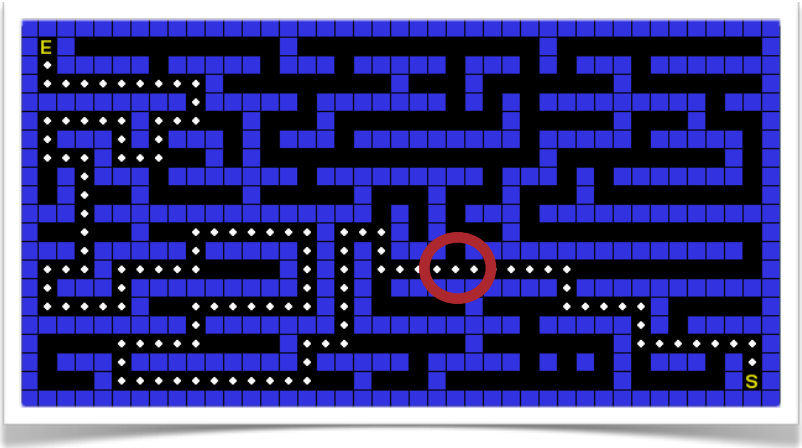
Greedy best-first search:

Our implementation of greedy best first search used the Manhattan distance for the heuristic h(n). The flaws of GBFS and the reason for its lack of optimality can be clearly seen in the images our program generated.

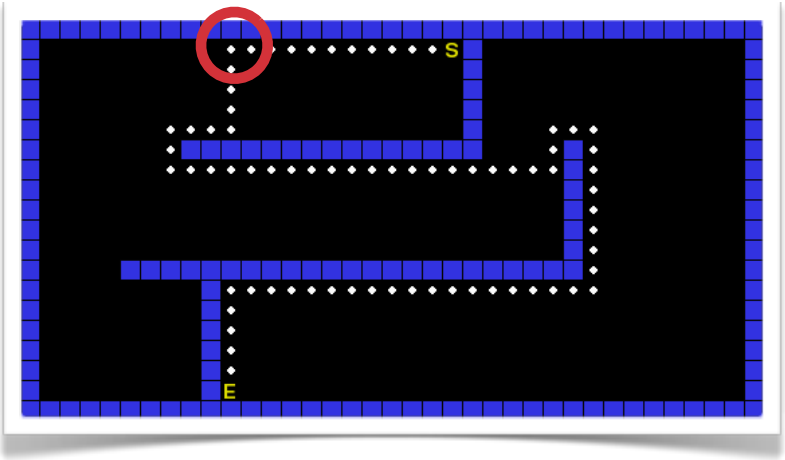
Observe the intersection highlighted in Medium Maze. Continuing left, or going upward at this intersection carry the same weight in our GBFS heuristic, and GBFS chooses to continue left (because of the ordering of its child nodes). However, going upward at this intersection would result in the optimal path. A similar effect can be observed in the intersection highlighted in Small Maze. The optimal choice at this intersection is to go down, despite temporarily moving further from the goal. GBFS, with it's greediness, is forced to move upward at this point, eliminating the possibility of finding an optimal path.

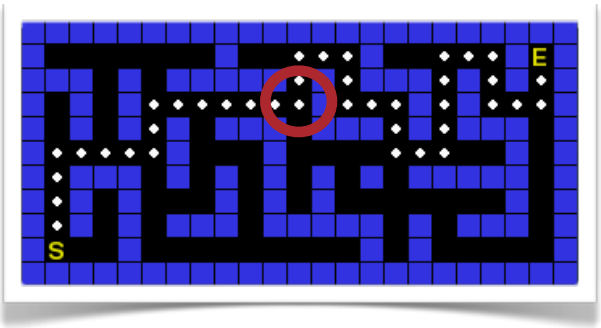
It is interesting to note: GBFS outperformed every other algorithm in terms of Solution path cost and Nodes expanded on Open Maze. Intuitively, this makes sense: GBFS is most problematic in mazes with dozens of intersections and possibilities; the chance of choosing a non-optimal intersection as it did in Small Maze magnifies.

Medium maze:
Solution path cost: 124
Nodes expanded: 158
Max search tree depth: 124
Max size of the frontier: 18



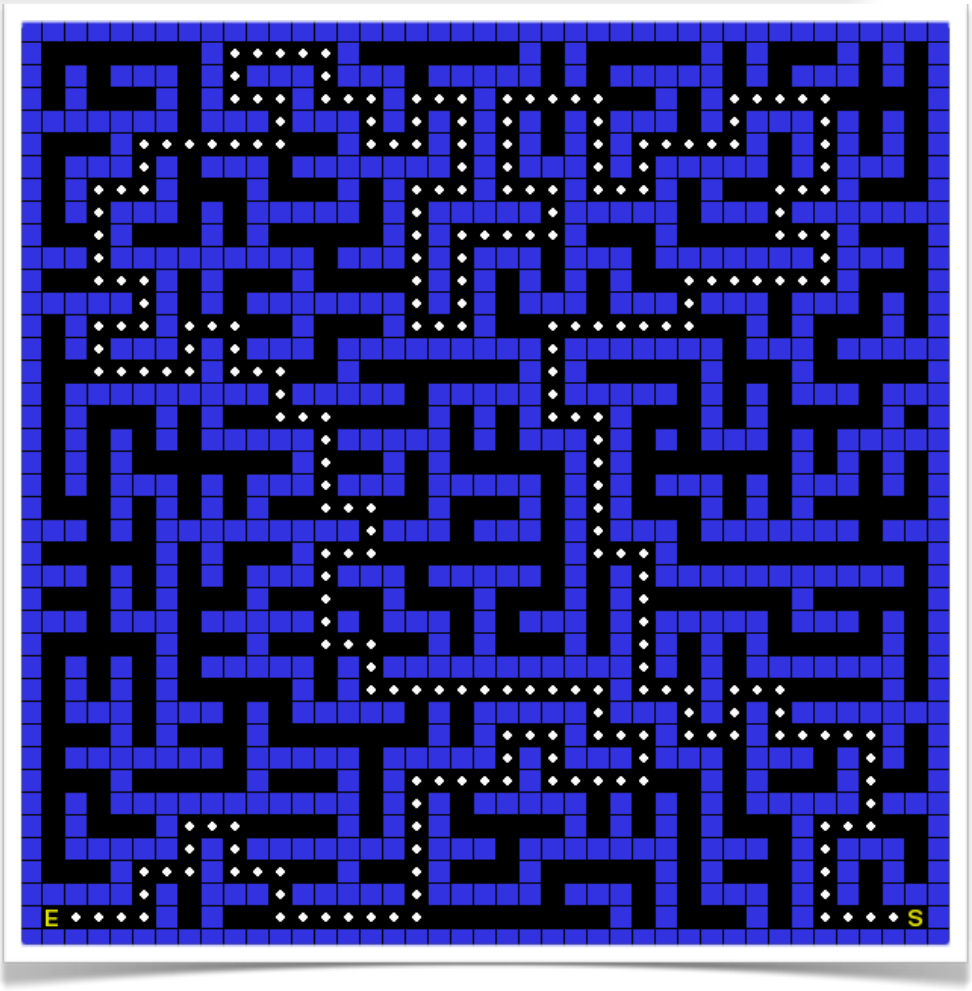
Open maze:
Solution path cost: 74
Nodes expanded: 346
Max search tree depth: 74
Max size of the frontier: 42





Small maze:
Solution path cost: 40
Nodes expanded: 46
Max search tree depth: 40
Max size of the frontier: 14

Big maze:
Solution path cost: 266
Nodes expanded: 623
Max search tree depth: 266
Max size of the frontier: 30

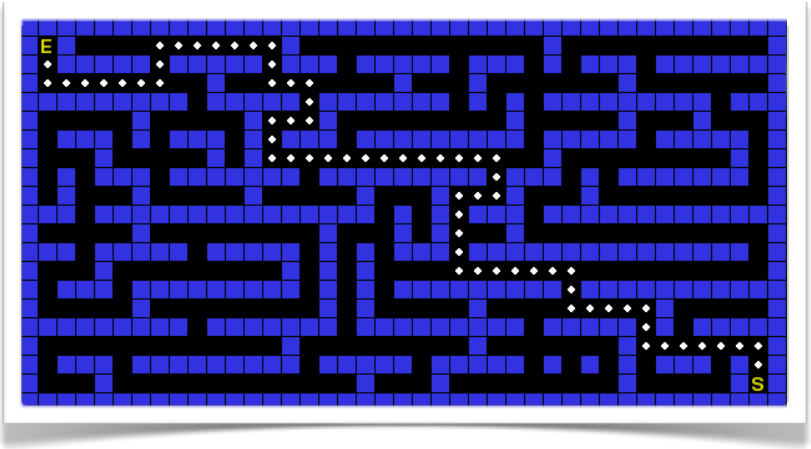


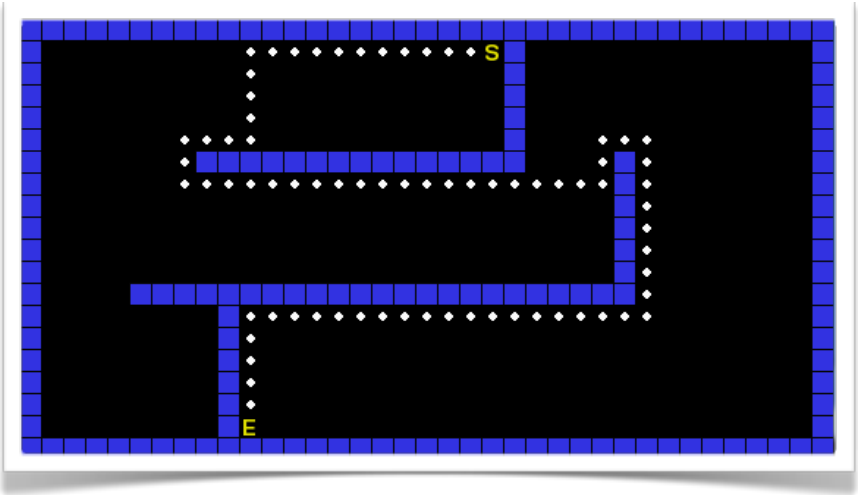
A-Star Search:

Our implementation of A* found the optimal solution while reducing the number of nodes expanded compared to BFS, which was expected. A* used much of the same functionality that GBFS used, with the additional heuristic g(n). To accomplish this heuristic, the pathCost function inside of the AStar class (*see AStar.py*) iterated through the current node to its parent (start), summing the path cost at each step and then returning it.

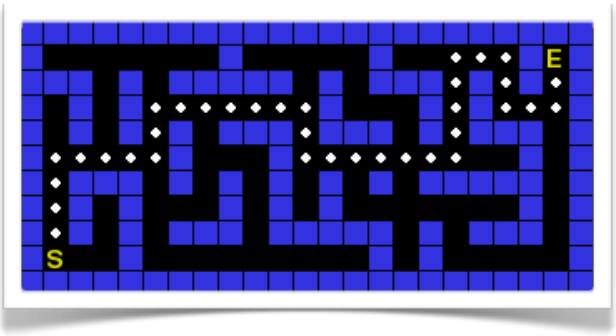
A* performed best overall in Small Maze, Medium Maze, and Big Maze, losing to GBFS in Open Maze due to Nodes Expanded.

Medium maze:
Solution path cost: 68
Nodes expanded: 198
Max search tree depth: 68
Max size of the frontier: 17



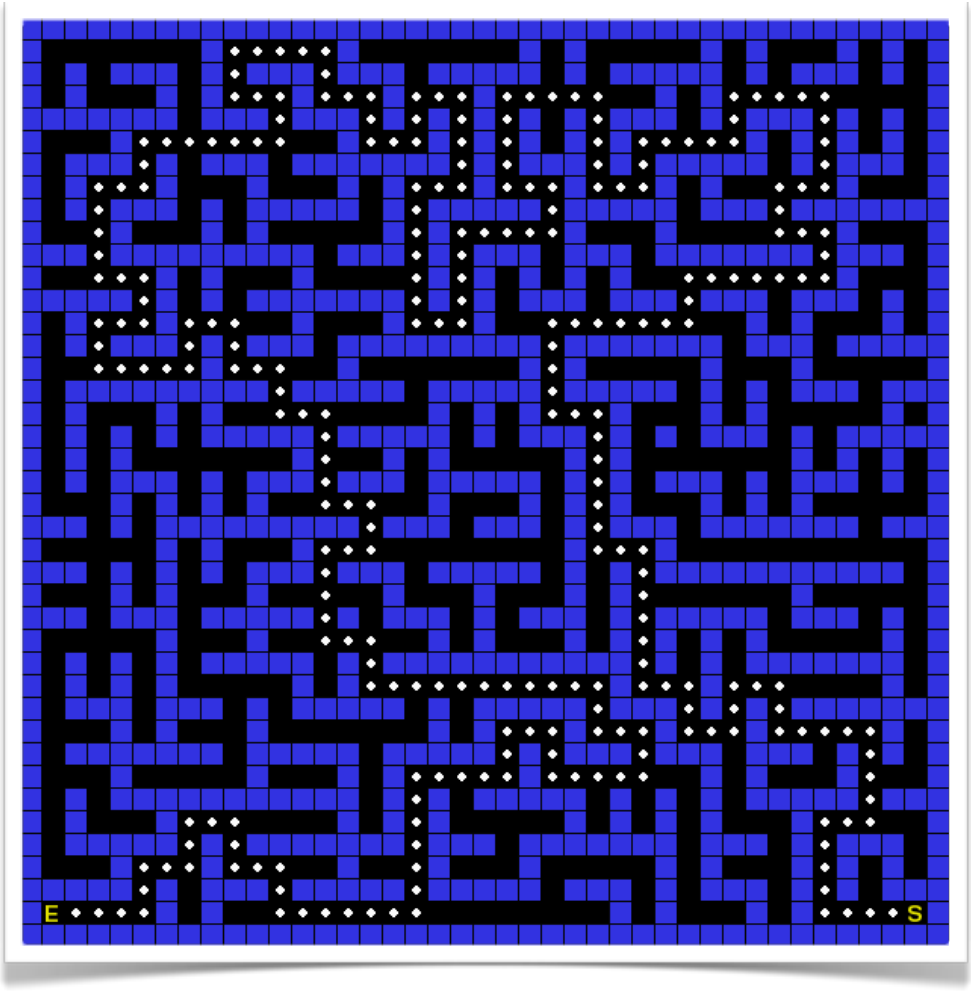


Open maze:
Solution path cost: 74
Nodes expanded: 440
Max search tree depth: 74
Max size of the frontier: 27



Small maze:
Solution path cost: 36
Nodes expanded: 98
Max search tree depth: 36
Max size of the frontier: 9

Big maze:
Solution path cost: 266
Nodes expanded: 793
Max search tree depth: 266
Max size of the frontier: 9



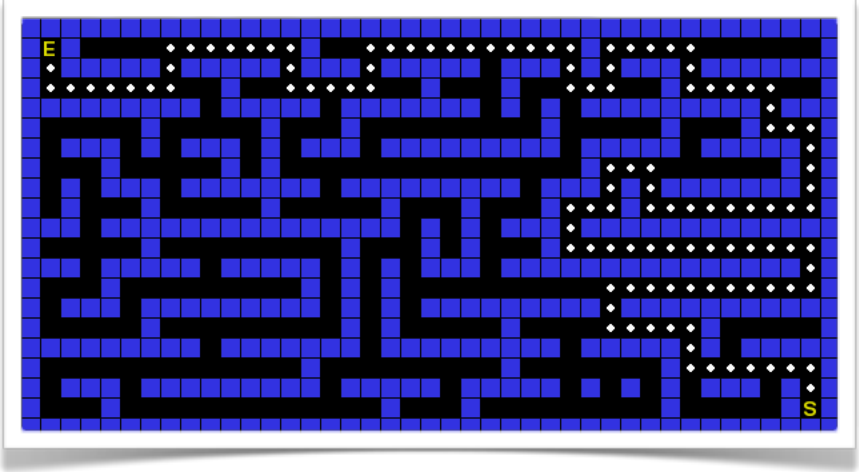
1.2 Search with different cost functions

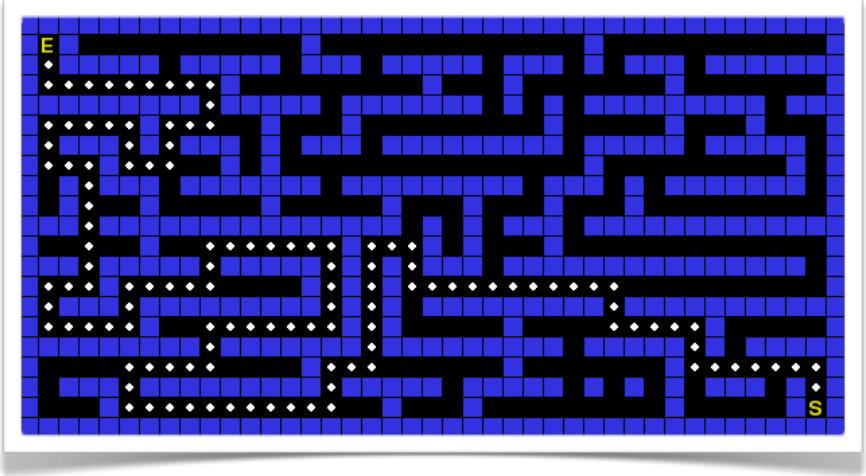
Uniform Cost Search:

The intuition that the cost function c1 will result in a solution that prefers the right side, and the cost function c2 the left side, is clearly visible in both Medium Maze drawings. Our UniformCost class (*see UniformCost.py*) contains both of these cost functions. In this particular maze, the cost function c1 results in a shorter solution path. We tried these same cost functions when searching for improvements in part 1.4. This did not yield a better solution as it did in Medium Maze. Intuitively, these cost functions would be most useful if it was known that the left or right side of the maze would generally require a longer path.

Path cost **c1:** $\frac{1}{(2^x)}$

Medium maze:
Solution length: 116
Solution path cost: 2.015884
Nodes expanded: 390
Max search tree depth: 116
Max size of the frontier: 8





Path cost **c2:** 2^x

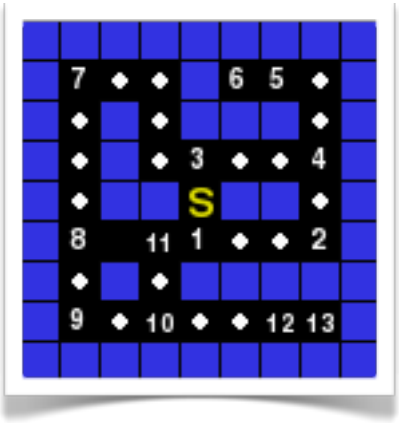
Medium maze:
Solution length: 124
Solution path cost: 2.21727e+12
Nodes expanded: 192
Max search tree depth: 124
Max size of the frontier: 16

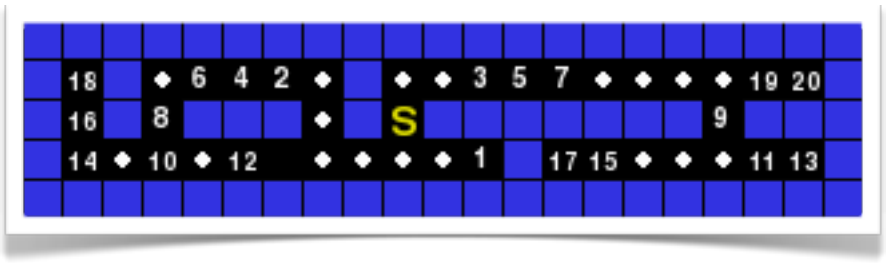
1.3 Search with multiple dots

Depth First:

In all of the drawings that follow, the goals have been numbered by the order in which they were found. For part 1.3, the presence of multiple goals caused us to modify our depth first search class to contain an array of goal states (named goalStates, *see DepthFirst.py*). The original functionality created for DFS in 1.1 is called repeatedly until the array is empty. For these mazes, the performance of DFS was worse than any other search. In the solution drawings, it can be seen that DFS often finds two adjacent goals in non-consecutive steps. For example, in Tricky Search, DFS discovers the 8th goal before discovering every goal in the upper chamber of the maze. As such, it must retreat back toward the start, increasing it's path cost.

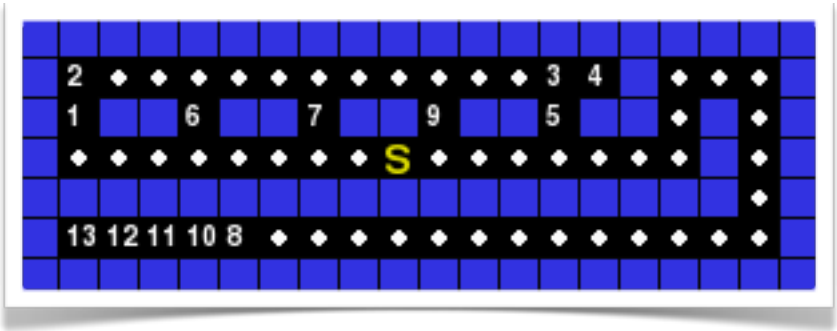
Tiny search:
Solution path cost: 138
Nodes expanded: 194





Small search:
Solution path cost: 213
Nodes expanded: 265

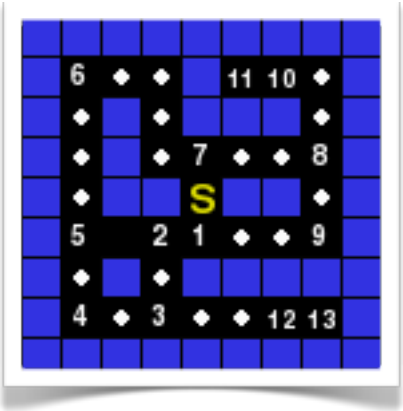
Tricky search:
Solution path cost: 358
Nodes expanded: 405

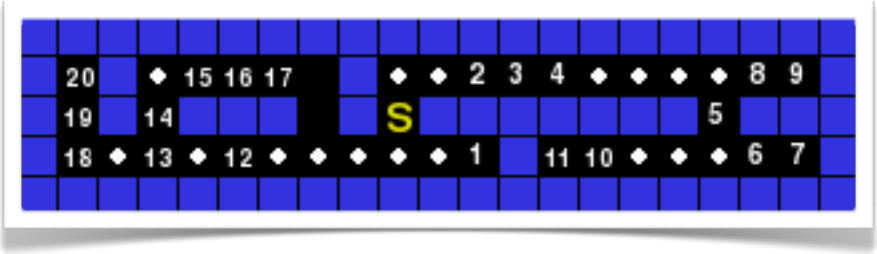


Breadth First:

Breadth First Search outperformed DFS in terms of Solution Path Cost. As with DFS, the modification to our implementation included an array of goal states (named goalStates, *see BreadthFirst.py*). BFS continued to search until this array was empty. Utilizing a Node class, which contains a pointer to its parent, we were able to iterate from the last goal state to start, obtaining the solution path. This method was utilized as it prevented us from making the easy mistake of counting each goal twice (once because it is a "start", and once because it is an "end"). BFS performed better than DFS in terms of Solution path cost. However, because it is an uninformed algorithm, it is only guaranteed to find the optimal path to the first goal.

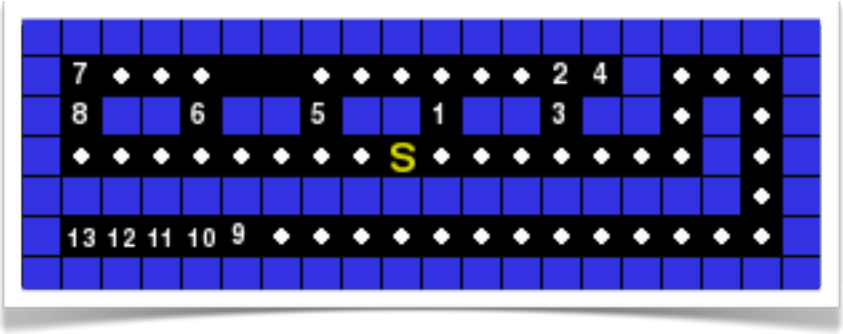
Tiny search:
Solution path cost: 80
Nodes expanded: 124





Small search:
Solution path cost: 206
Nodes expanded: 264

Tricky search:
Solution path cost: 210
Nodes expanded: 231



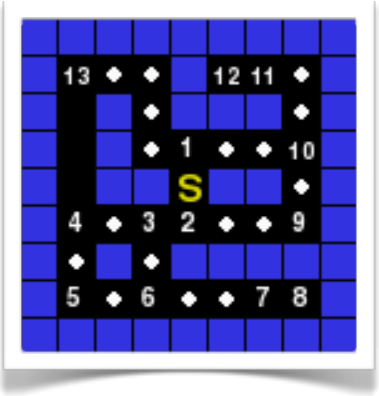
Greedy best-first search:

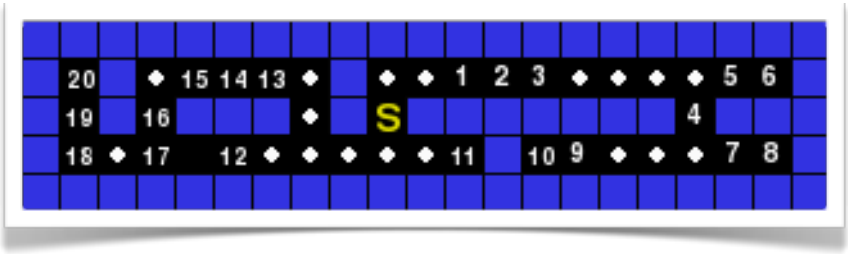
Altering Greedy Best First Search to find multiple goals involved the addition of much more functionality than DFS or BFS. Being an informed search, GBFS has knowledge of the locations of the goals in the maze. After discovering a goal in the maze, GBFS utilizes functionality (*see the priority queue in GreedyBestFirst.py*) to always search for the next closest goal at its location.

A note on admissibility: The distance is still calculated using the standard Manhattan distance functionality originally used in 1.1 (*see Maze.py*). The Manhattan distance will at most match the distance required to get to any goal, never overestimating it. With multiple goals, this situation will occur if each goal is in a vertical or horizontal line to the next goal, with no wall blocking a goal.

The performance of GBFS was far better than DFS or BFS. Observing the solution images, it is clear GBFS always chooses the next closest goal after finding a goal, which is the behavior we aimed for.

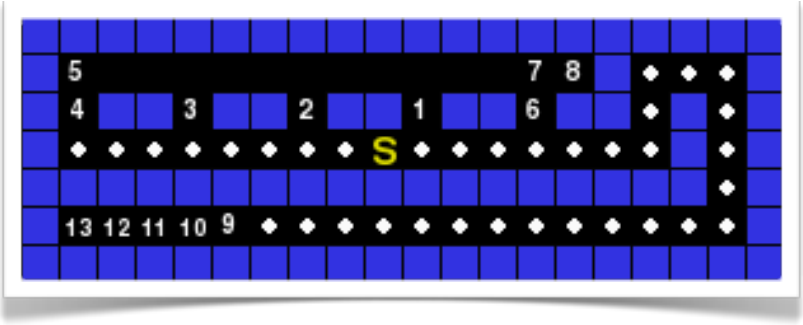
Tiny search:
Solution path cost: 42
Nodes expanded: 50





Small search:
Solution path cost: 79
Nodes expanded: 105

Tricky search:
Solution path cost: 78
Nodes expanded: 127

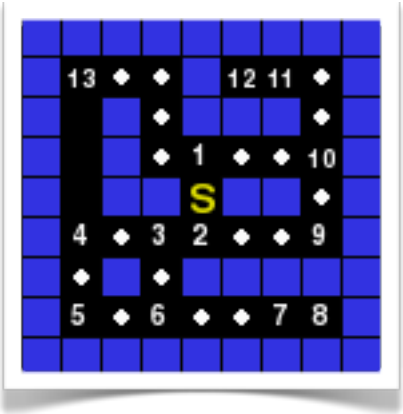


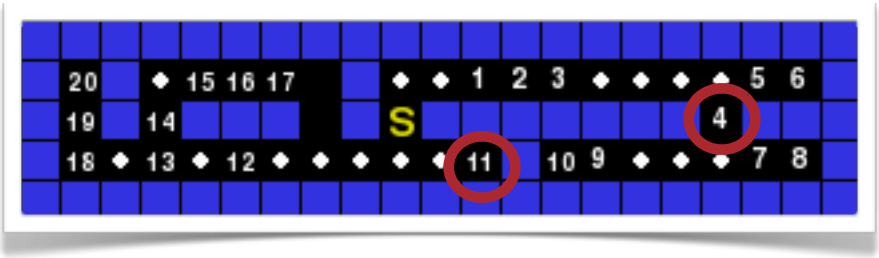
A* search:

Modifying A* search involved much of the same overhaul in functionality that GBFS involved. A priority queue was implemented, as well as the same sorting mechanisms to order goal states by their relative distance (for an explanation on admissibility, *see GBFS*; for code, *see AStar.py*). Some of the solution paths that A* found were nearly optimal, if it were not for a couple miss-steps. For example, observe the highlighted area in *Tricky Search*. Had A* instead chose to reverse its choice of goal 7 and 8, a Solution path cost of 62 would have been found.

A classic improvement of A* over GBFS is visible in the highlighted portion of Small Search. GBFS and A* are identical for the first three goals, until GBFS chooses 11 as its 4th goal (it is closer), while A* continues right. Now GBFS must iterate all the way back to the right side of the maze to find the rest of the goals, while A* simply clears the right chamber and moves on.

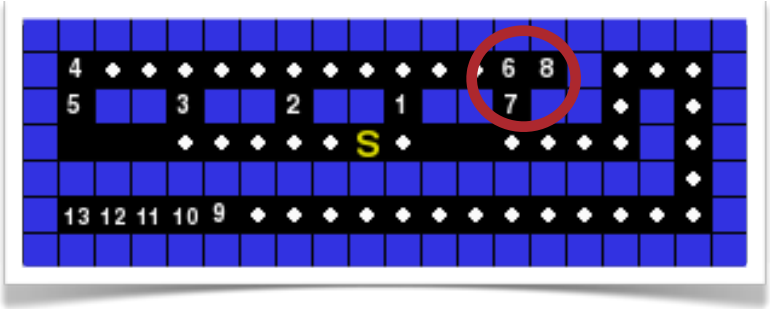
Tiny search:
Solution path cost: 42
Nodes expanded: 63





Small search:
Solution path cost: 65
Nodes expanded: 110

Tricky search:
Solution path cost: 64
Nodes expanded: 153

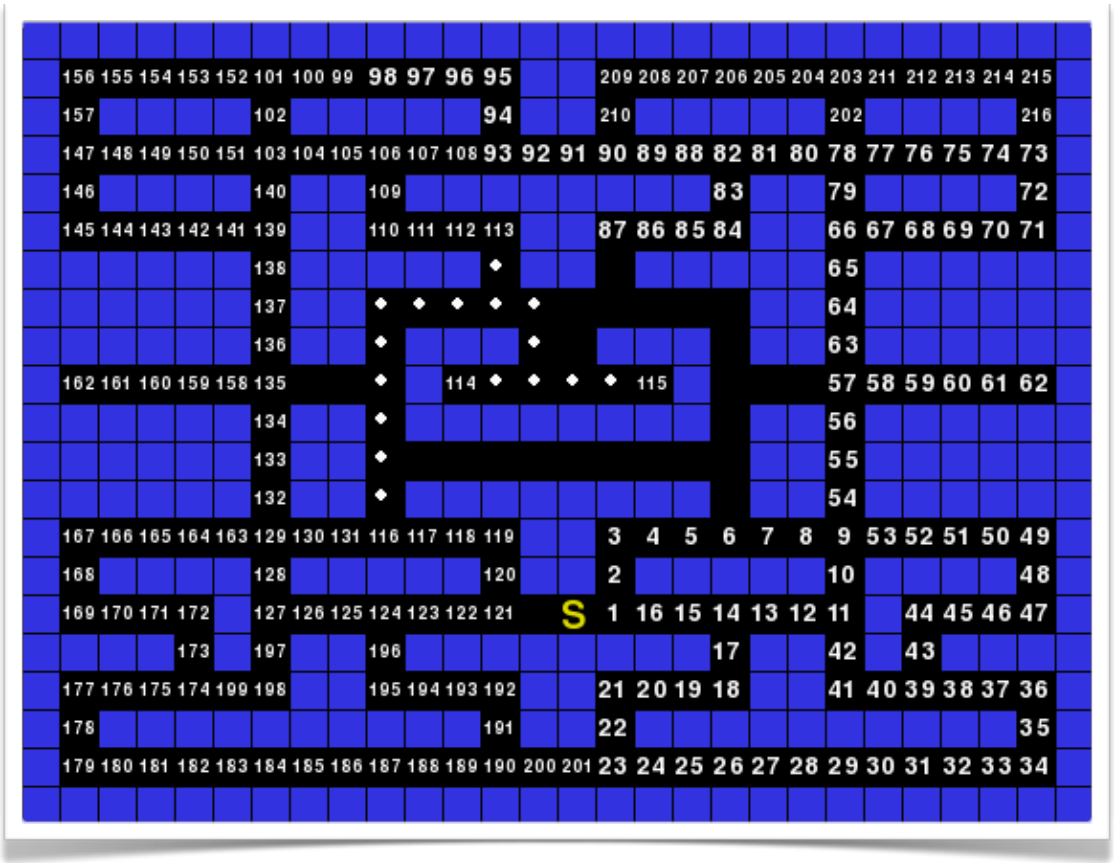


1.4 Suboptimal search

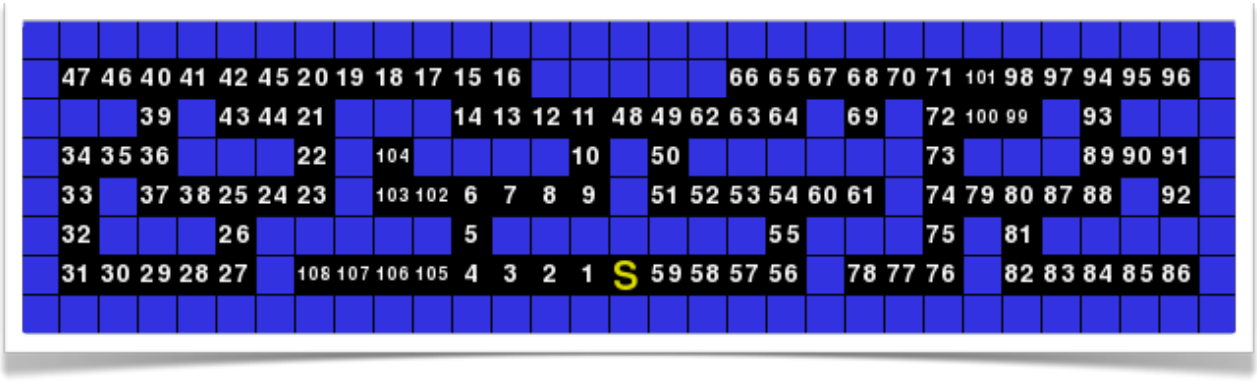
A* search

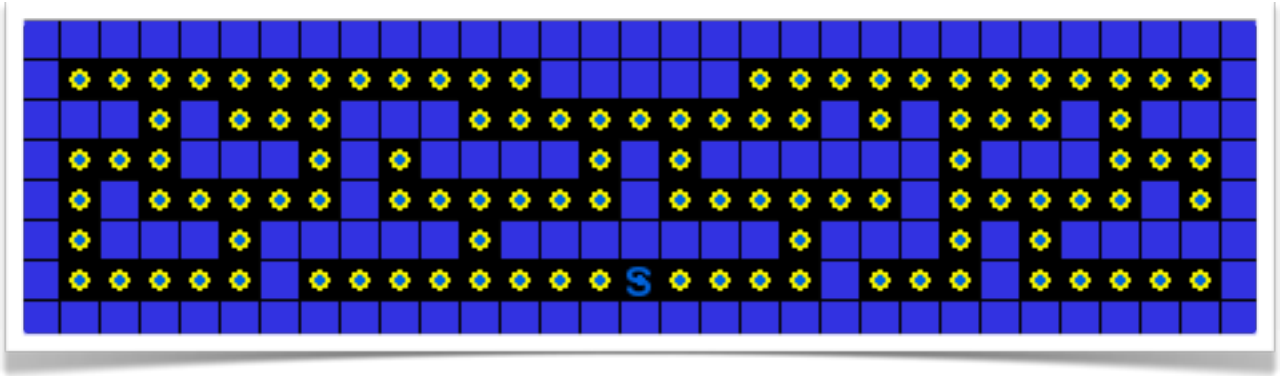
Several strategies were attempted to revise A* to achieve better performance on *Big Search* and *Medium Search.* We tried to implement the same functionality seen in part *1.2,* creating a path cost that we hoped would sweep the maze in one general direction. This did not produce better results. We also tried to implement an increasing path cost for revisiting already explored nodes, which also failed. Eventually, we settled for a priority queue of goals in the order of their Manhattan distance from the current goal, and this functionality was extended to all informed search strategies for part *1.3.*

Big search:
Solution path cost: 333
Nodes expanded: 677



Medium search:
Solution path cost: 178
Nodes expanded: 346





(Not numbered)