

Artificial Intelligence : Assignment #1

German language and literature 2016130927 Park Jun Yeong

I. The result of *autograder.py* in terminal:

```
Anaconda Prompt
Question q8
=====
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** Method not implemented: findPathToClosestDot at line 502 of searchAgents.py
*** FAIL: Terminated with a string exception.

### Question q8: 0/3 ###

Finished at 15:53:06

Provisional grades
=====
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
-----
Total: 12/25

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

<Pic1: Provisional grades of q1-q4 from *autograder.py*>

II. Discussions on 3 algorithms (DFS, BFS, A*) for playing Pacman.

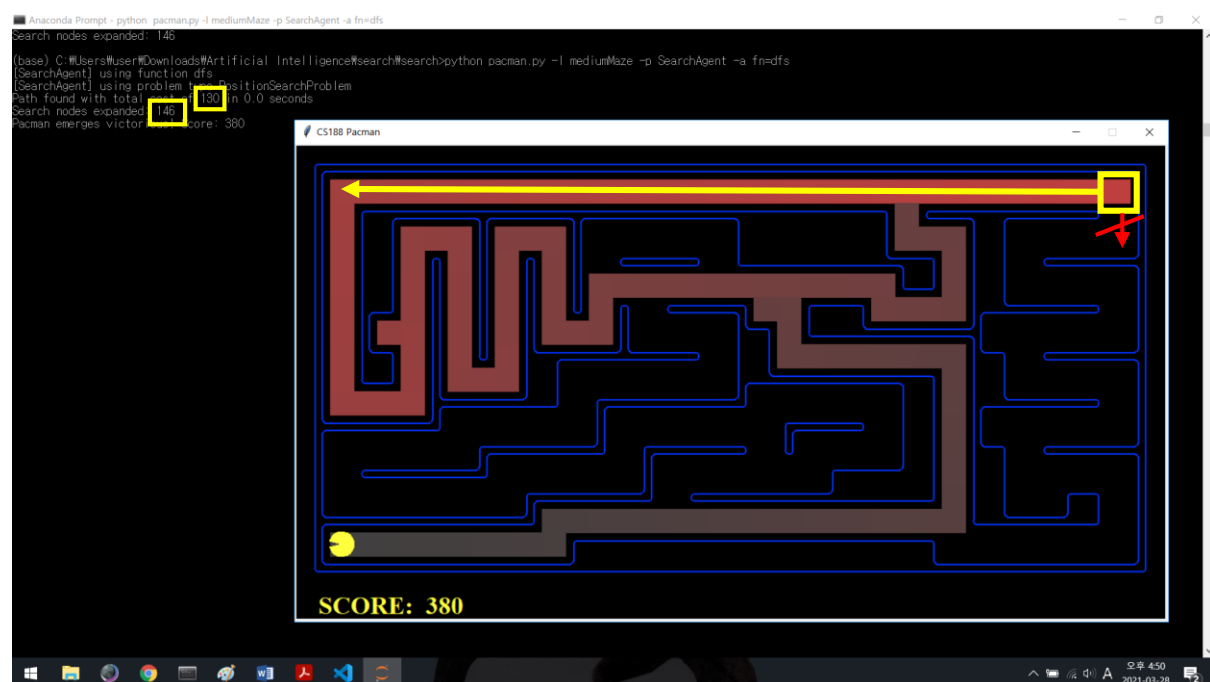
Q1. Which algorithm is better between DFS and BFS in mediumMaze?

A1. **BFS** is much better in terms of the shortest path: when you start to searching the mediumMaze, score begins with 510. The cost for each single Pacman move is added up and subtracted from the total score 510 when it ends. Since cost is not the case in terms of BFS, there is no need to care about path cost counting (using UCS, we will take this). That is, you should take into account how to find out 'the shortest' path among several options if you hope to win this game with the highest scoring. Here comes one issue relative to this observation: then, which algorithm guarantees the shortest path?

(1). Try to figure out this issue in terms of concrete instance - mediumMaze. It is a medium size of maze where its starting position is located at the top right corner and an ending position at the bottom left (please take a look at the below Pacman playing picture) and you can reach goal with some different paths (e.g. 2 ones for below DSF and BSF, and another

version of DSF's path in terms of children node push-ordering issue, which is described in DSF part as below). You should consider optimal path among those.

Run DFS: you must keep going down this tree (as given search graph) until you can no longer move at each leaf node. If that node is not a goal, you should turn back to the last unvisited successor node, step by step. Considering this mechanism in mediumMaze, Pacman has two options at the starting position of tiny place, move eastward or southward. Depending on which direction (n, w, s, e) is returned from the getSuccessors function, the order of node explore differs and then DSF-path could print another version of path; at this mediumMaze, your path has taken yellow westward direction, not the unchosen red southward one, which only remains in the stack. To sum up DSF, it is as follows and its rules:



<Pic2: Pacman using DFS. *square – start zone, arrow – chosen direction, red zone – searched nodes>

i. several paths are possible (following programmed order of visiting successors).

ii. given DFS path is not necessarily the shortest one.

It is some of report after running DSF algorithm in this maze. Please take a note of total cost (i.e. counted path length) and search nodes expanded:

[SearchAgent] using function dfs

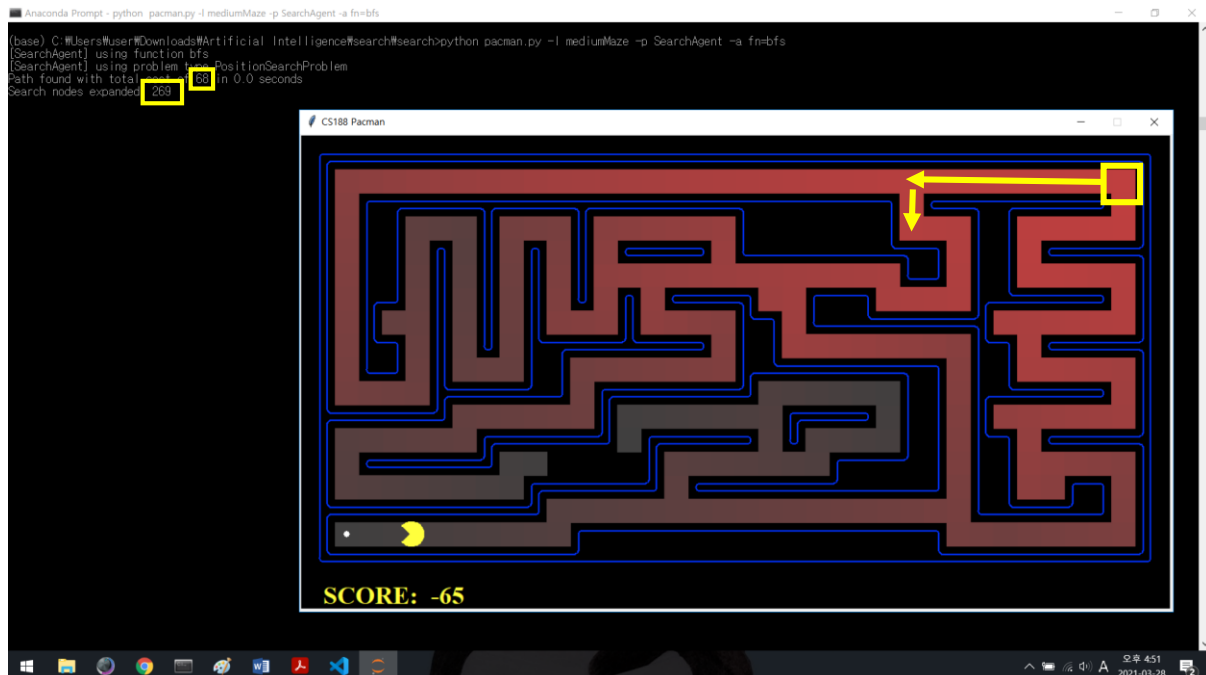
Path found with **total cost of 130** in 0.0 seconds # roughly twice of length using BFS

Search nodes expanded: **146** # roughly half of searched nodes using BFS

Pacman emerges victorious! Score: 380

(2). Then, what about BSF? Since your queue would push(=enqueue) its visited node's successors in sequence, it is surely guaranteed, if there remain unvisited ones, to visit all of nodes at the same level. Among those paths, you have the shortest one. Even if your BFS Pacman has searched roughly twice in terms of search nodes expanded in this

mediumMaze(269 nodes): indeed, what matters is the fact that you obtained a least cost path(68). Here is result of running BFS and its report.



<Pic2: Pacman using BFS. *square – start zone, arrow – chosen direction, red zone – searched nodes>

The given BFS path is always the shortest path.

The shortest path using BFS reports, moreover, a roughly half of cost and wins the DFS.

[SearchAgent] using function bfs

Path found with **total cost of 68** in 0.0 seconds # roughly half of length using DFS

Search nodes expanded: **269** # roughly twice of cost using DFS

Pacman emerges victorious! Score: 442

Note that DFS-path has a total cost of 130 and BFS almost twice. In conclusion, BFS surely gives you a least cost path from start to the goal and it takes advantage of high scoring relative to the path-length over DSF in the maze path-finding problem.

Q2. What is more efficient heuristic algorithm than Manhattan? Discuss specific cases where these algorithms works effectively.

A2. Depending on how Pacman moves in the maze (i.e. 4 direction, 8 direction or diagonal allowed or not), each heuristic algorithm works differently. That is why you have searched more efficiently in Manhattan than in Euclidean or Null here. In this game, Pacman is allowed to take only 4 kinds of move (north, south, east and west) and not to go diagonally. Since then, it seems better to use Manhattan than others. It means, however, there is no other algorithm which is more efficient than Manhattan in this model. You might get various answers to this problem:

(1). Exact heuristic value using pre-computed path length

Note that Manhattan distance is just seemingly-good path length, not the actual one: you have no idea of how that maze is constructed and where blocks are set. Then, you can get trapped into wrong way and spend lots of time searching nodes which are actually unused in the solution. What if you have accurate value of the shortest path length as heuristic $h(n)$ for all of given nodes? Using the dataset - where path length values between goal and all of other ones in the maze are calculated in advance, Pacman would take 'the optimal path' with even less expanded nodes. Even when starting and goal position varies at each time, you can calculate all of pairs (for size $N \times M$ maze, # of computation = NM) and record them in dataset. This technique gives you an accurate information, but seems too unfeasible in many cases, considering memory or time-consuming issues.

(1.1) All of pairs computed in advance

In case your goal and starting position is predetermined in maze size of $N \times M$: your dataset records the path length value for all of pairs between existing nodes and the goal position, as said above. It might call some search algorithms (e.g. BFS) where the shortest path is always guaranteed. These are to be taken as the exact heuristic value, $h(n)$ and returned when you encounter that position. What if our starting or goal position differs? In this case you can easily have their path length using some tabula where each pair of nodes is checked in terms of connectivity as in Dijkstra's algorithm. After huge amount of computation, your another dataset is completed and then start all of pairs search as like before. However, this method takes too much time or memory complexity.

(1.2) Waypoints algorithm using Manhattan

This is the modified all of pairs heuristic - some specific main points at the maze, say waypoints, are computed in advance (all of pair of waypoints + goal position) and their own path length is used as bridge between start and goal. In this algorithm, there could be some issues for locating waypoints of maze: the number of waypoints, where to locate them, and memory efficiency problem as well.

So you have two kinds of nodes: waypoint or non-waypoint. If you encounter the waypoint, just use the each heuristic value which lies in precomputed dataset. If not, get that $h(n) = h'(n, w1) + \text{path length}(w1, w2) + h'(w2, \text{goal})$ where path length is from dataset and h' function can be thought as Manhattan algorithm. Using exact heuristic values for some waypoints, you end up getting much more efficient algorithm than Manhattan.

(2). Set Manhattan distance's weight adaptively

Putting aside the 'exact' heuristic issue, you might redesign the Manhattan distance: in this maze problem you have one simple Manhattan algorithm, which is multiplied D is 1 where your $h(n) = D * \text{abs}(dx + dy)$: with your cost function and given min cost, Pacman could have higher heuristic value, if any, than cost = 1. Even though this $D = \text{cost}$ 1 satisfies the condition to be admissible and consistent, Pacman runs too scared even in case it doesn't need to slow down that much; Figure out the highest possible heuristic value at each position.

If you just want to run faster than any algorithm, you can use some overestimating algorithm such as squared Euclidean algorithm. If not, then, balancing between overestimating metrics and too relaxed algorithm, does matter. For instance, let's assign D min value among costs of

next nodes (Set aside the fact that each next cost would be same as 1 in this maze like bigMaze). This balancing technique is the most important issue when you code Manhattan algorithm's variable D. Or just handle your D ratio, scaling them into the size of maze, blocked structures, etc. Likewise, there exist heuristic algorithms which are more efficient than Manhattan in case you are willing to handle some memory, computation, or scaling issues in this maze.

Q3. Discuss how to redesign search algorithm in this specific case using DSF algorithm and then observe optimality and efficiency issues.

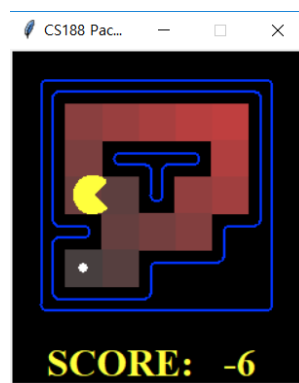
A3. On Q1, We have addressed advantages and disadvantages of DSF and BSF for this case: even though DSF runs faster, it does not guarantee the optimal solution, say 'the shortest path', which can be obtained from running BSF. Then, how about keeping high speed of DSF's advantage while finding optimality? It's observed that there would be several redesigned algorithms using DSF mainly in path-finding: IDS, iterative deepening DSF algorithm is one important algorithm among them and implemented here.

Following this concept, let's discuss how this algorithm works theoretically - in practical instance of this maze problem as well: for more detail, please take a note of IDS source code in *search.py* as submitted. And then, think about this matter in terms of optimality and efficiency issues.

(1). IDS: Run DSF with its accessible node depth limited, increased by 1 iteratively

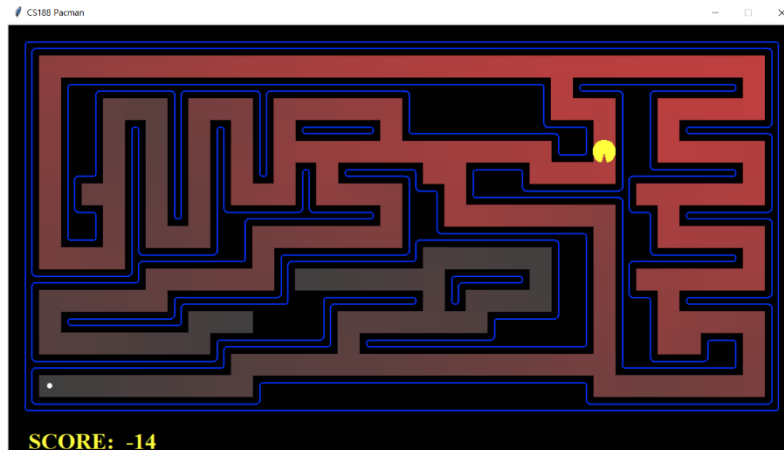
Since DSF has not guaranteed path optimality for its abrupt property such that you cannot stop or reconsider other possible paths, IDS is based on DLS, Depth Limited Search where you are not allowed to go beyond the limit depth level. Using this DLS, all of nodes should be searched fairly, which means that Pacman can think of another version of route: It is actually used to manage some time complexity issues but would not ensure such optimality and even completeness, if goal node lies beyond the scope of limited depth. Since then, you may redesign this algorithm in which its preset limit depth is increased by 1 iteratively. At that moment you reach the goal, your search ends up and that path is returned. Since your obtained path is the shortest version among those possibly accessible depth level, it has optimality as well. Then, think about how this algorithm works in given maze game?

(2). Run IDS and its result report



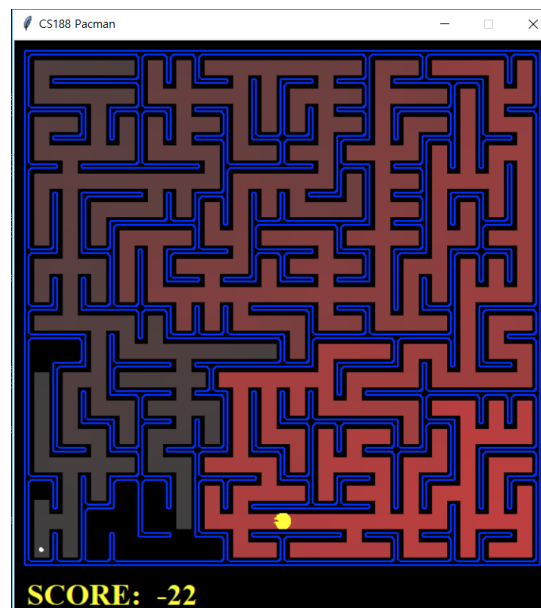
```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l tinyMaze -p SearchAgent -a fn=ids
[SearchAgent] using function ids
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 104
```

<Pic3-4: Pacman using IDS in tinyMaze and its report>



```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=ids
[SearchAgent] using function ids
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.4 seconds
Search nodes expanded: 8450
```

<Pic5-6: Pacman using IDS in mediumMaze and its report>



```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l bigMaze -p SearchAgent -a fn=ids -z .5
[SearchAgent] using function ids
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 2.8 seconds
Search nodes expanded: 60831
```


<Pic7-8: Pacman using IDS in bigMaze and its report>

(3). Optimal but inefficient in given case using IDS

As seen in above pictures, IDS gives you the shortest path (total cost of 10, 68, 210 in tiny, medium, bigMaze), which has shown poor efficiency (search nodes expanded is 104, 8450, 60831 in sequence), compared to other optimal-guaranteed search algorithms: it stems from

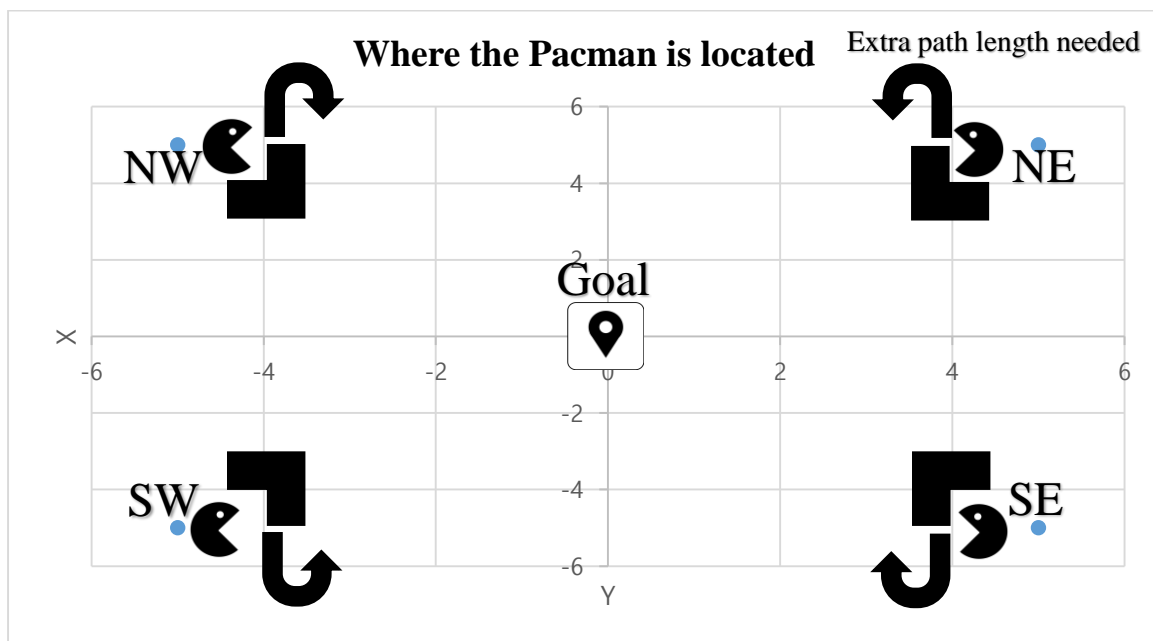
the fact that your Pacman has lots of choices. Plus, Pacman should travel all around in the maze where the goal node is located in the corner. In order to travel efficiently from one corner to another corner as here, you should not look around all of nodes at each moment encountering some nodes – optimal, but too inefficient in given maze. If your node is located at non-max depth (i.e. closer to the starting node), however, you can obtain the optimal solution more efficiently. Likewise, you may consider how your maze is designed (size, start or goal node's position, allowed move direction, etc.) before running algorithms.

III. The result of A* algorithm using myHeuristic – Blocked_distance:

#1	Up #2	#3
Left #4		Right #6
#7	Down #8	#9

<Pic9: 3X3 square where Pacman stands>

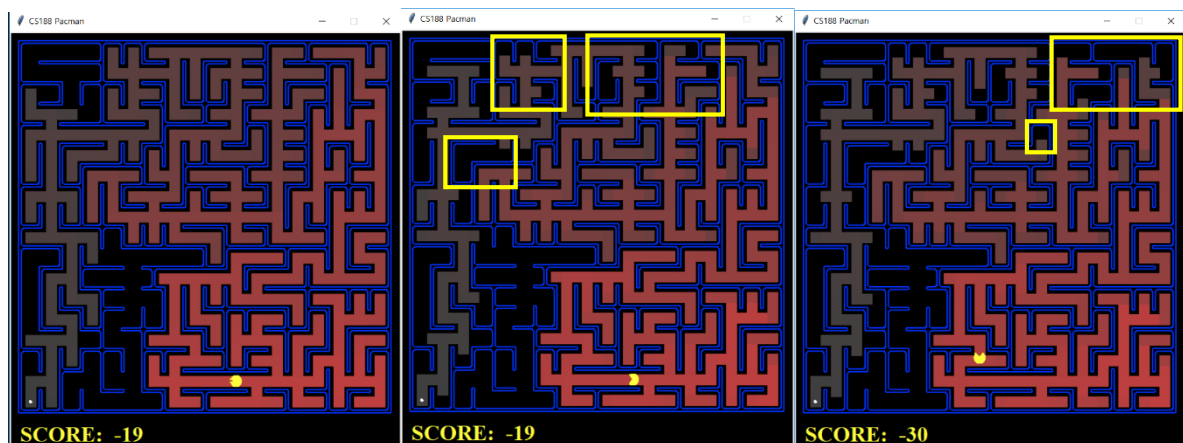
This Blocked_distance algorithm comes from the Manhattan distance, which is effective heuristic in 4-way maze runner and computes feasible $h(n)$, assuming that there are no obstacles between current and goal position: you might spend much time expanding your nodes, however, in certain cases since you cannot know whether you'll get stuck or not. that means, your Pacman senses 1X1 as it sees only itself. Then, how about making your scope bigger? In 3X3 your Pacman gets information about your surrounding area. You might be surrounded by blocks (in this code, these are called as walls). Using 3X3 square, your runner asks whether up-, down-, right-, or left side way is accessible or not. Plus, where your Pacman stands from the viewpoint of goal is also important. Please see this little map.



<Pic10: Relative position of Pacman with respect to Goal with some assumed blocks>

Calculated each time, your relative position (Northern east, Northern west, Southern west, or Southern East) is taken. Since you might get into trouble of unneeded node searching due to Manhattan algorithm, recognize at which direction you are blocked off. If then, your runner takes another route more efficiently using Blocked_distance algorithm, which counts $h(n)$ of that route even bigger. In this algorithm, D (the factor of Manhattan distance) is assigned 3 ($1 < D \leq 3$, effective range of D) when Pacman encounters blocked case. For more detail, please see its source code, *SearchAgents.py*.

You can easily figure out how it goes with Pacman's pictures and reports. Please check some improved (say, more efficient - unexpanded part) yellow square carefully:



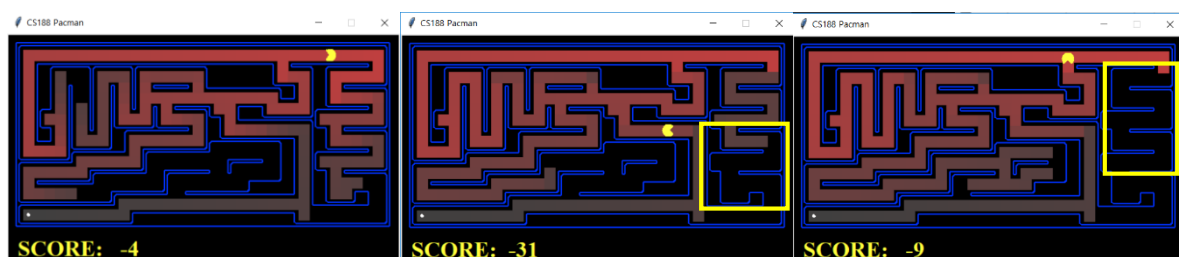
<Pic11-13: Pacman using Manhattan, Blocked of $D = 2$, and 3 in bigMaze as above>

```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l bigMaze -z .5 -p SearchAgent
-a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
```

```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l bigMaze -z .5 -p SearchAgent
-a fn=astar,heuristic=myHeuristic
[SearchAgent] using function astar and heuristic myHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 516
```

```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l bigMaze -z .5 -p SearchAgent
-a fn=astar,heuristic=myHeuristic
[SearchAgent] using function astar and heuristic myHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 499
```

<Pic14-16: Reports for Pacman using Manhattan, Blocked of $D = 2$, and 3 in bigMaze as above>



<Pic17-19: Pacman using Manhattan, Blocked of $D = 2$, and 3 in mediumMaze as above>


```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 221
```

```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=myHeuristic
[SearchAgent] using function astar and heuristic myHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 221
```

```
(base) C:\Users\User\Downloads\CS\Artificial Intelligence\search\search>python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=myHeuristic
[SearchAgent] using function astar and heuristic myHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 220
```

<Pic20-22: Reports for Pacman using Manhattan, Blocked of D = 2, and 3 in mediumMaze as above>

It is observed that some nodes remain unsearched in two maze as seen in yellow-square so that total number of node expanded decreases; that is, this heuristic distance works fine in both mazes.

There are several issues in spite of these improvement, such as longer computational time, how to manage suitable D value, or even such that whether this D value satisfies admissibility or consistency property as well: putting aside these time or space complexity problems, let's discuss matter of D, which has a lot to do with admissibility or consistency. For this matter of D, it is revealed that Pacman have no optimal solution using Blocked_distance in mediumMaze if your D becomes larger than 3 (even 3.1 makes a big mistake while finding path of mediumMaze so that D is set as 3 in submitted source code, showing the highest efficiency and guaranteed optimality), which means that you 'might' have no guaranteed optimal path (Since your $h(n) = D * \text{Manhattan distance}$, your set D might make $h(n)$ larger than h^* , real length in some cases). That is, Blocked_distance algorithm seems to target only some features of those 3 maze models and inadmissible if taken with generalized mazes, unlike those generalized model such as Manhattan or Euclidean algorithm.