2022FA - INTRO TO ARTIF INTEL

Assignment 1

Fast Trajectory Replanning

Submitted By:

Kanchu Kiran

NetID: KK1219

Shraddha Sanjeev Pattanshetti

NetID: SP2304

PART 0 – SETUP YOUR ENVIRONMENT

Grid/Maze Implementation – PYGAME UTILITY IN PYTHON

Specifications:

- Dimension = 5 x 5
- Agent: Purple cell on the grid
- Target: Turquoise cell on the grid
- Barrier: Black cells
- Open list in the traversal: Dark grey
- Closed list/Visited cells: Light grey

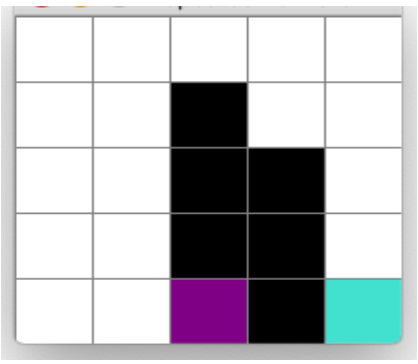
Functions used for grid creation and manipulation:

- pygame.display.setmode() – used to create the desired window size
- pygame.set.caption()- to provide the title to the maze
- pygame.draw() – used to draw the grid of the specified width.
- pygame.event.get() – to get all the events taking place at background
- pygame.draw.line() – to draw horizontal and vertical grid lines in the maze.
- pygame.mouse.get_pos() – to get the coordinates of the spot clicked.
- pygame.k_space – to perform an action when space is being pressed.

Shortest presumed unblocked path

- Console outputs the co-ordinates (row, col) of the shorted path by Agent to reach Target in A* and vice-versa in backward A*
- Using draw function of pygame utility the shortest path is constructed and displays in orange coloR

Grid Snapshot



Part 1 - Understanding the methods [10 points]: Read the chapter in your textbook on uninformed and informed (heuristic) search and then read the project description again. Make sure that you understand A* and the concepts of admissible and consistent h-values.

a) Explain in your report why the first move of the agent for the example search problem from Figure 8 is to the east rather than the north given that the agent does not know initially which cells are blocked.

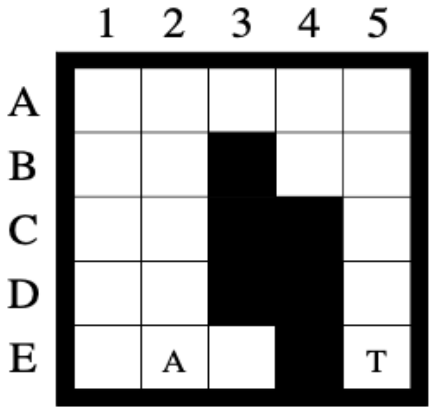


Figure 8: Second Example Search Problem

b) b) This project argues that the agent is guaranteed to reach the target if it is not separated from it by blocked cells. Give a convincing argument that the agent in finite grid-worlds indeed either reaches the target or discovers that this is impossible in finite time. Prove that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.

[PART 1 - A] The first move of the agent for the example search problem from Figure 8 is to the east rather than the north given that the agent does not know initially which cells are blocked because:

- The algorithm used by the agent to find the path from source to destination is A*. A* algorithm finds the optimal and shortest path between the initial and target node. It is an informed cost search algorithm, using heuristics and path cost to find the way to the goal. Regarding heuristics, A* uses Straight Line Distance heuristics (SLD).

- SLD heuristics choose a straight line from the initial node to the goal. Furthermore, it does not consider obstacles while selecting the path. SLD is also admissible as it does not overestimate the path cost.
- In Figure 8, Our initial position is (E,2), three cells away from the target cell. In A*, taking SLD as heuristics into consideration, instead of moving to the north, that is, (D,2), it will move to (E,3) as the f(n) (path cost) associated with (E,3) is less compared to (D,2) cell provided regardless of the obstacle as agent is not aware of the barrier.

[PART 1 - B] A* is a combination of 2 algorithms, one being Dijkstra's and the other being Breadth's first search algorithm. In Dijkstra's algorithm, we get the shortest path in the graph, but this is not always the case in the Greedy Best first search.

There can be two arguments that prove the fact that the pathfinding algorithm will either reach the path if available or provide no path if not available.

- First, we maintain two lists, Open and Closed, to store the potential best path nodes yet to be visited and the already traversed nodes. As said before, A* is a combination of Dijkstra's algo, which uses the distance from the start node to find the next node, and Greedy best-first search, which uses the current node's distance to the target node. In A* we use the g(n) distance of the current node from the start node and the h(n) distance of the current node from the target giving f(n) = g(n) + h(n). The start node is inserted into the empty list. The algorithm chooses a node from the open list with the least f cost to reach the target. If the selected node is not the destination node, it puts all the neighboring nodes of the current one into the open list, and the process continues. Note that we store information about the parent nodes of neighbors, which helps in backtracking and path creation. If the open list is empty at some point, the algorithm can't find the path to the target. This strategy helps avoid traversing unnecessary nodes, which increases execution time, and helps avoid infinite loops by visiting the same node repeatedly.

- The second reason is that A* can find the shortest path provided that the estimated heuristic cost should always be less than the actual cost.

Part 2 - The Effects of Ties [15 points]: Repeated Forward A* needs to break ties to decide which cell to expand next if several cells have the same smallest f-value. It can either break ties in favor of cells with smaller g-values or in favor of cells with larger g-values. Implement and compare both versions of Repeated Forward A* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation.

[Hint: For the implementation part, priorities can be integers rather than pairs of integers. For example, you can use $c \times f(s) - g(s)$ as priorities to break ties in favor of cells with larger g-values, where c is a constant larger than the largest g-value of any generated cell. For the explanation part, consider which cells both versions of Repeated Forward A* expand for the example search problem from Figure 9.]

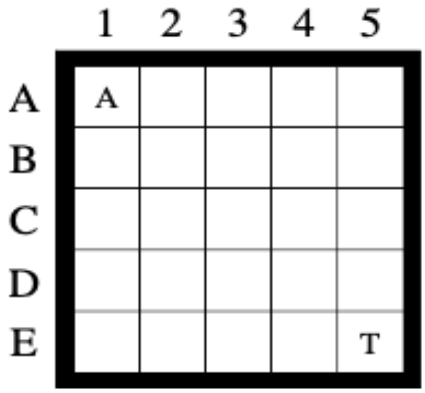
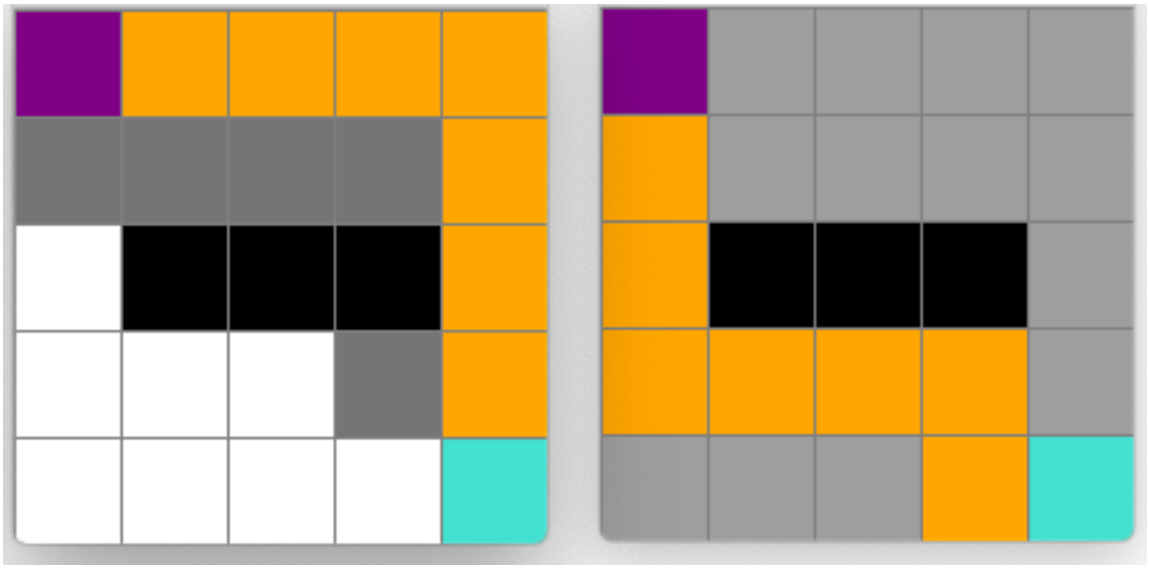


Figure 9: Third Example Search Problem

Comparison between tie-breaking with larger g_value and smaller g_value in Repetitive Forward A*:

Test 1: 5x5 Grid size, shortest path possible between Agent and Target		
Basis for Comparison	larger g_val	smaller g_val
Execution Time (in sec)	0.031284	0.080834
Number of expanded cells	8	21
Cell count in shortest path	8	8

Larger g_value **Smaller g_value**



Repetitive Forward A* with tie-breaking using larger g_value:

```
Shortest Path from Agent to Target: 8 steps

[ 0 0 ] : [ 1 0 ] : [ 2 0 ] : [ 3 0 ] : [ 4 0 ] : [ 4 1 ] : [ 4 2 ] : [ 4 3 ] : [ 4 4 ]
```

Repetitive Forward A* with tie-breaking using smaller g_value:

```
Shortest Path from Agent to Target: 8 steps

[ 0 0 ] : [ 0 1 ] : [ 0 2 ] : [ 0 3 ] : [ 1 3 ] : [ 2 3 ] : [ 3 3 ] : [ 3 4 ] : [ 4 4 ]
```

We know that neighboring nodes with the least f_value in the open list are traversed first. The Tiebreaker situation arises when two nodes of the same f value are in the open list to be expanded. There are two ways in which we can break ties, using h and g values.

- Tiebreakers can help in improved performances, which avoids exploring the nodes of different paths that are equal and optimal. When compared, G costs are more reliable than H costs as the prediction of G value (the price we need to get from the source to the current node) is almost accurate, unlike the prediction made in the case of H values. taking the h-value first has no influence on the number of nodes expanded unless it is the case that h(x)=0.
- **Larger g_values perform better as a more reliable g_value of the f_value dominates the less reliable h value resulting in a more reliable f_value. In this case, the algorithm picks a cell having a greater g value.**

The explanation for the above-implemented algorithm

- The open list of the algorithm is a priority queue. The g_value for the start node is initialized to zero.
- The f value of the start node is the h value of the start node, as the g value is zero initially. Then, open_set.put() function accepts 3 parameters (f_value , tie-breaking parameter, node)
- Here g_value of the start node is included to break the tie in situations in the case with smaller g_val tie breaking situation, whereas h_value[node] – g_value[node] is used in the case of larger g_value tie breaking where the f_value of two nodes is the same. Below are the code snippets for the same:

Tie breaking with smaller g_value

```
open_set.put((f_value[adjacent], g_value[adjacent], adjacent))
```

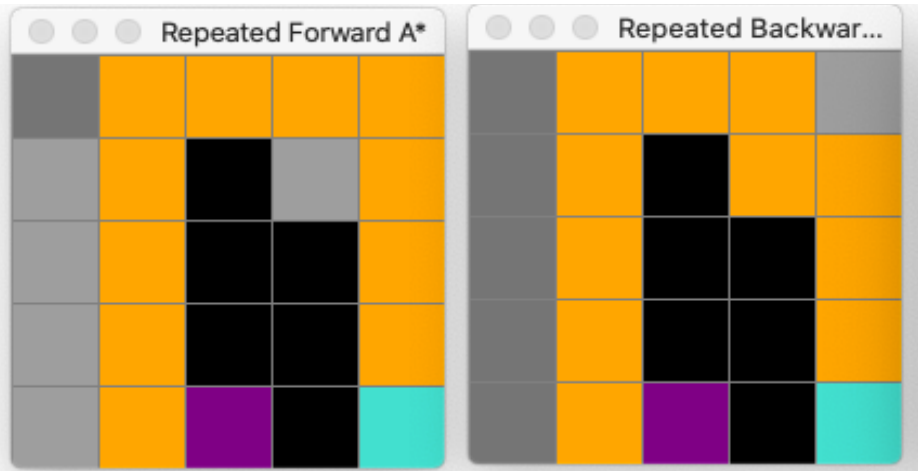
Tie breaking with larger g_value

```
open_set.put((f_value[adjacent], h(adjacent.get_pos(), end.get_pos()) - g_value[adjacent], adjacent))
```

Part 3 - Forward vs. Backward [20 points]: Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both versions of Repeated A* should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

Comparison between Repeated Forward A* and Repeated Backward A*

Test 1: 5x5 Grid size, shortest path possible between Agent and Target		
Basis for Comparison	Forward A*	Backward A*
Execution Time (in sec)	0.154223	0.040268
Number of expanded cells	17	13
Cell count in shortest path	12	12



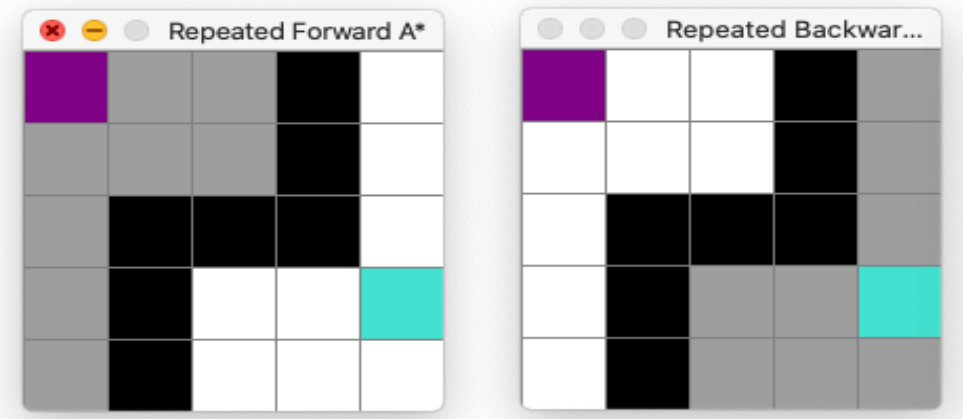
Forward A*: Shortest Path from Agent to Target: 12 steps

[2 4] : [1 4] : [1 3] : [1 2] : [1 1] : [1 0] : [2 0] :
[3 0] : [4 0] : [4 1] : [4 2] : [4 3] : [4 4]

Backward A*: Shortest Path from Target to Agent: 12 steps

[4 4] : [4 3] : [4 2] : [4 1] : [3 1] : [3 0] : [2 0] :
[1 0] : [1 1] : [1 2] : [1 3] : [1 4] : [2 4]

Test 2: 5x5 Grid size, shortest path **not** possible between Agent and Target		
Basis for Comparison	Forward A*	Backward A*
Execution Time (in sec)	0.006428	0.025586
Number of expanded cells	9	9
Cell count in shortest path	NA	NA



Observation and Comparison:

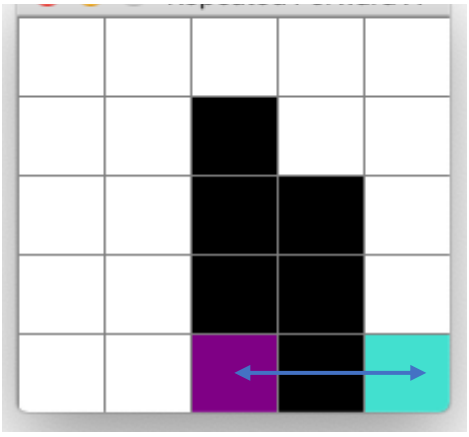
We can see that the Backward A* performs better in terms of both number of expanded cells and execution time than Forward A* in Case 1 but Forward outperforms Backward A* in Test 2.

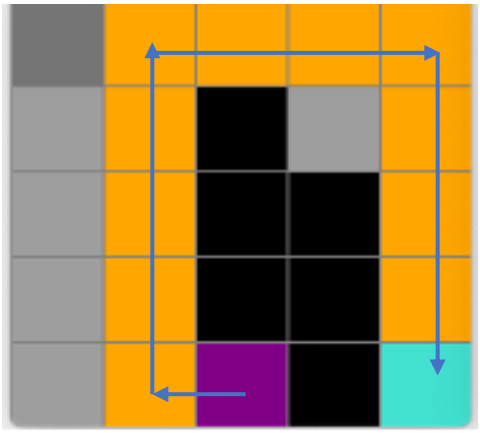
Search direction impact explanation: closeness to the barrier (blocked cell)

- In test 2, The target observes blocked cells close to itself. Thus, the blocked cells are close to the target of the search in the early phases of forward A* searches, but close to the start of the search in the early phases of backward A* searches.
- The above condition in opposite in test 1.
- The closer the blocked cells are to the start of the search, the more cells there are for which the Manhattan distances are perfectly informed and thus the faster A* searches are that break ties between cells with the same f-values in favor of cells with larger g-values since they expand only states along a cost-minimal trajectory from cells for which the Manhattan distances are perfectly informed to the goal of the search.
- And, hence the execution time and number of expanded cells of Backward A* are less than Forward A* in test 1 but opposite in test 2.

Part 4 - Heuristics in the Adaptive A* [20 points]: The project argues that "the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions." Prove that this is indeed the case.

Furthermore, it is argued that "The h-values $h_{new}(s)$ ... are not only admissible but also consistent." Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.





The Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions because:

- There are several heuristics used while calculating the f_value of the node. Manhattan distance is the absolute difference between one point and another. For a given point, the other point at a given Manhattan distance lies in a square.

- The Manhattan distance metric of two nodes **n** (x_n, y_n) and m (x_m, y_m) is defined as $d(n, m) = |x_n - x_m| + |y_n - y_m|$

- To be precise, when any action generates the current node and its successor node, the predicted cost of reaching the target from the current node should be less than the step cost of getting to the successor plus the expected cost of running the goal from the successor. Manhattan distance is the most used heuristic in A* as it is admissible and consistent.

- One of the reasons why Manhattan distance is consistent is estimated cost is less than the actual cost as it does not consider obstacles in the grid world and can only move 1 unit at a time and in 4 directions.
- As the grid consists of obstacles, the algorithm will give a path with no blocks, taking more moves than the Manhattan distance to reach the target, which is admissible and consistent.

Adaptive A* leaves initially consistent h-values consistent even if action costs can increase because:

Let,

- s = non-goal state
- a= actions that can be executed on state s
- c1 = costs before the changes
- c1 = costs after the changes
- h1(s) = h-values before running the consistency procedure
- h2(s) = h-values after its termination

$h1(s_{goal}) = 0$ and $h1(s) \leq c1(s, a) + h1(suc(s, a))$
The h-value of any non-goal state is continuously non-increasing over time since it only gets updated to an h-value that is smaller than its current h-value. So, we can list down 2 different cases for a non-goal state s and all actions a that can be executed in states:

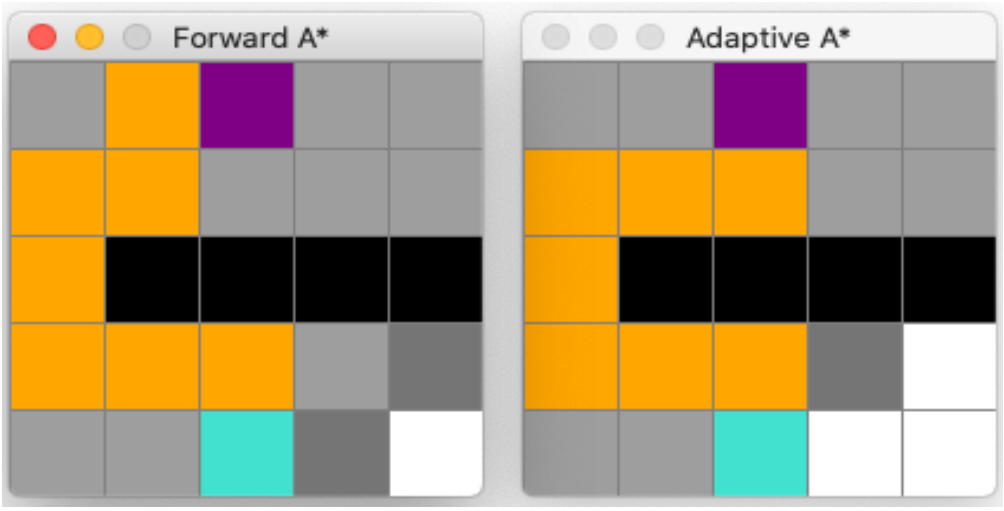
- The h-value of state suc(s,a) never decreased
  - $h1(suc(s, a)) = h2(suc(s, a)))$ and $c1(s, a) \leq c2(s, a)$.
  - Then, $h2(s) \leq h1(s) \leq c1(s, a) + h1(suc(s, a)) = c1(s, a) + h2(suc(s, a)) \leq c1(s, a) + h2(suc(s, a)$.
- The h-value of state suc(s, a) never decreased
  - $h1(suc(s, a)) = h2(suc(s, a)))$ and $c(s, a) > c'(s, a)$
  - Then, c1(s, a) decreased and s not equal to s_{goal} and $h2(s) \leq h1(s) \leq c2(s, a) + h2(suc(s, a)$.
- The h-value of state suc(s,a) decreased
  - $h1(suc(s, a)) > h2(suc(s, a))$.
  - $h2(s) \leq h1(s) \leq c2(s, a) + h2(suc(s, a))$.

Thus, $h2(s) \leq c2(s, a) + h2(suc(s, a))$ in all three cases, and the h-values remain consistent with respect to the goal state.

Part 5 - Heuristics in the Adaptive A* [15 points]: Implement and compare Repeated Forward A* and Adaptive A* with respect to their runtime. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both search algorithms should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

Comparison between Repeated Forward A* and Repeated Backward A*

Test 1: 5x5 Grid size, shortest path possible between Agent and Target		
Basis for Comparison	Forward A*	Adaptive A*
Execution Time (in sec)	0.090954	0.060214
Number of expanded cells	17	16
Cell count in shortest path	8	8



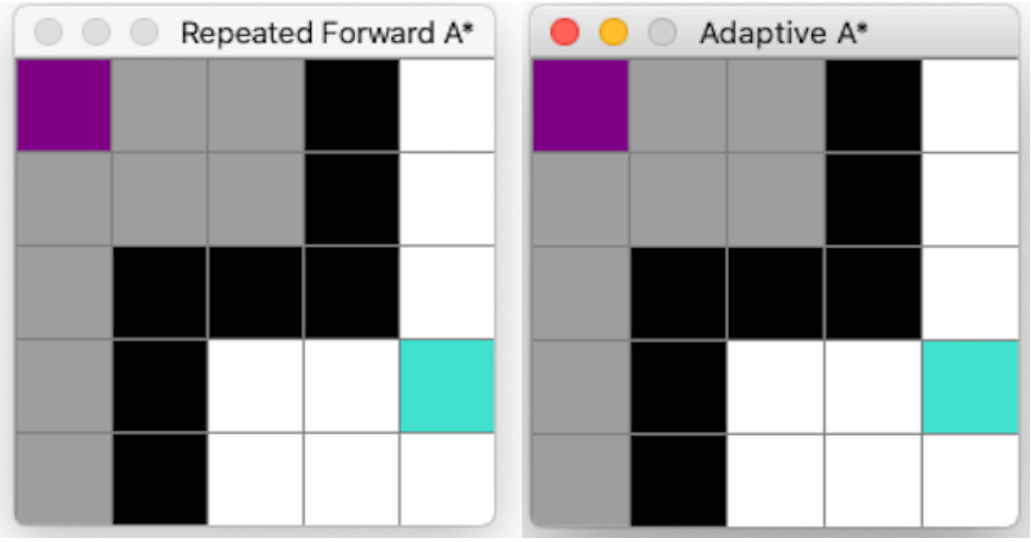
Forward A*: Shortest Path from Agent to Target: 8 steps

[2 0] : [2 1] : [1 1] : [0 1] : [0 2] : [0 3] : [1 3] : [2 3] : [2 4]

Adaptive A*: Shortest Path from Agent to Target: 8 steps

[2 0] : [1 0] : [1 1] : [0 1] : [0 2] : [0 3] : [1 3] : [2 3] : [2 4]

Test 2: 5x5 Grid size, shortest path **not** possible between Agent and Target		
Basis for Comparison	Forward A*	Adpative A*
Execution Time (in sec)	0.009128	0.009057
Number of expanded cells	9	9
Cell count in shortest path	NA	NA



Observation and Comparison:

- We can see that the Adaptive A* performs better in terms of both number of expanded cells and execution time than Forward A* in both the test cases.
- In both test cases that forward A* searches and Adaptive A* follows same trajectories if they break ties in the same way. They differ only in the number of cell expansions, which is larger for forward A* than in Adaptive A*
- Above happens as **Adaptive A* updates the heuristics while forward A* searches does not.** Thus, forward A* searches limits to the local minimum in the heuristic value surface.

Part 6 - Statistical Significance [10 points]: Performance differences between two search algorithms can be systematic in nature or only due to sampling noise (= bias exhibited by the selected test cases since the number of test cases is always limited). One can use statistical hypothesis tests to determine whether they are systematic in nature. Read up on statistical hypothesis tests (for example, in Cohen, Empirical Methods for Artificial Intelligence, MIT Press, 1995) and then describe for one of the experimental questions above exactly how a statistical hypothesis test could be performed. You do not need to implement anything for this question, you should only precisely describe how such a test could be performed.

Statistical hypothesis test for comparison between Adaptive A* and Repetitive A*:

- **Null hypothesis ($H_{o)}$** – adaptive A* algorithm functions better than the repetitive A* Algorithm as it expands fewer cells than A*

- **Explanation of Null hypothesis:**
  Consider a maze of size 10*10. The repetitive A* algorithm and Adaptive A* algorithm is implemented to locate the shortest path from the source to the target. It is seen that Adaptive A* transcends the A* algorithm as it uses a more efficient heuristic function H(s) = g(goal) – g(current) than the heuristic function (Manhattan) utilized by the A* algorithm. In addition, cells expanded by Adaptive A* are less than A*, giving us the optimal path to the target.

- **Alternate hypothesis ($H_{a)}$** – A* algorithm performs well than the Adaptive algorithm as the grid size increases.

- **Explanation of Alternate hypothesis:**
  We know that for any dimension algorithm surpasses the A* algorithm in terms of cell expansion. Assuming we increase the grid size dramatically, we might get the shortest path to the target, but the time complexity is comprised. The reason for this is the calculation of new h(n) for every expanded cell. Hence, there are situations where Adaptive A* fails to perform well than A*.