**Part 0 - Setup your Environments**

Run the Generate\_Maze.py to create the 50 grid worlds. This file generates the 101x101 grid world on cell at a time stating from (1,1) and ending at (101,101). Also it marks a cell as locked with the Probability of 30%.

**Part 1 – Understanding the Method**

a) A\* uses a priority queue as its open list. The priority queue is based on the f value of nodes such that the node with the minimum f value gets expanded first. The f value is in turn dependant on the g value and h value (f=g+h) where h is the heuristic function and g is the distance travelled from the start node till now. Every time the agent moves 1 step the g value will increase by 1. The h value on the other hand is based on the Manhattan distance between the current node and the goal. In this example, the g value of each of the nodes a, b, c when it enters the open list is equal to 1. At the same time the h value for node a, c is 4 whereas for node b(towards the east) it is 2. Hence the node b has the minimum h value and with the g value remaining the same for all three nodes, the f value for node b is the minimum. So the node that will expand first is Node b. Hence the agent moves to east initially.



Node b: H=2; G=1; F=3

Node c: H=4; G=1; F=5

Node a: H=4; G=1; F=5

b) A\* always expands nodes with the lowest f value. It terminates when it expands the goal node G with cost c\*, which is the optimal cost. Therefore, it does not expand nodes with f value greater than c\*. Moreover it expands all nodes with f value less than c\* because it will expand G only after expanding all nodes with f value less than c\*. This is true unless there is an infinite state space. Hence A\* will always reach a target in a finite graph or inform if there is no path.   
A\* only expands nodes that have lower cost than the optimal cost. Hence it expand all n such that f(n) < c\*. The optimal cost c\* is bounded by m^2 in a gridworld where m is the number of unblocked nodes. Hence the number of moves that the agent makes before reaching the target is bounded by the square of the number of unblocked cells.

Part 2 –

|  |  |
| --- | --- |
| Repeated Forward A\* | Repeated Backward A\* |
|  |  |
| Running Time 6.9377 ms | Running Time : 20.0387 ms |
| Number of Cells Expanded : 488 | Number of Cells Expanded : 1239 |

**Part 3 - Forward vs. Backward**

Observed Implementaoin:

|  |  |
| --- | --- |
| Repeated Forward A\* | Repeated Backward A\* |
|  |  |
| Running Time 6.9377 ms | Running Time : 20.0387 ms |
| Number of Cells Expanded : 488 | Number of Cells Expanded : 1239 |

Observations :

* Repeated Forward A\* has smaller number of expanded Cells as compared to Repeated Backward A\*. Based on our observation this is because Repeated Forward A\* has a lower branching factor as compared to repeated Backward A\*.

**Part 4 – Heuristics in the Adaptive A\***

A heuristic is consistent if the heuristic function follow the triangular inequality h(n) <= c(n, a, n') + h(n'). Manhattan distance always gives the shortest path between two nodes in a grid. H(n) is the shortest distance between the node n and the Goal node. c(n, a, n') + h(n’) along any ‘a’ will either be equal to or greater than h(n) depending upon the amount of deviation due to ‘a’ from the Manhattan path of n. Hence c(n, a, n') + h(n’) will either be greater than or equal to h(n) and the triangular inequality will hold.

The Heuristic of adaptive A\* algorithm is the goal distance gd(n) of a node n. Also, gd(n) <= gd(n’) + c(n, a, n’). This is because any c(n, a, n’) is adding an incremental cost towards the goal. Now if c(n, a, n’) increases then also gd(n) <= gd(n’) + c(n, a, n’) will hold true. Therefore the gd(n) is a consistent heuristic even if the action cost increases.

**Part 5 - Heuristics in the Adaptive A\***

|  |  |
| --- | --- |
| Repeated Forward A\* | Adaptive A\* |
|  |  |
| Running Time : 10.3441 ms | Running Time: 0.009184308841843089  Running Time: 0.010610394172037019  Running Time: 0.004066139340110908  Running Time: 0.15140909318991547  Running Time: 0.017897627486668455  Running Time: 0.18183414758757266  Running Time: 0.1642718012581028  Running Time: 0.06281788987268655  Running Time: 0.61568967562118  Running Time: 0.005797284906876143 |

* As can be observed the first iteration of adaptive forward A\* is almost same as that for repeated Forward A\*
* However before every consecutive iteration the heuristics for the visited cells is updated hence as and when a new start for the same Goal state starts from a State in the vicinity of a previous start state it is easier for the agent to choose a path to follow.
* This can be observed from the Running Time 1 and Running Time 4

**Part 6 - Memory Issues**

The memory consumption can be reduced by identifying attributes which are unique to every cell and attributes which are a heuristic. Then heuristic attributes need not be part of every cell. These can always be calculated on the fly. However if these heuristics were to change that is not be same as that given by the heuristic function these can be stored into a data structure which is a part of the environment. In this was the information stored per cell can be kept at a minimal.

**Part 7 – D\* Lite**

We are following the implementation D\* Lite in Java. This implementation runs a D\* Lite algorithm on a grid of 101x101 and 1001x1001. Following are the timing recorded timings for the same.

D\* Lite Adaptive A\*

101 x 101: 82ms

101 x 101: 79ms

101 x 101: 14ms

101 x 101: 78ms

1001 x 1001: 1205ms

1001 x 1001: 1406ms

1001 x 1001: 2718ms

1001 x 1001: 862ms