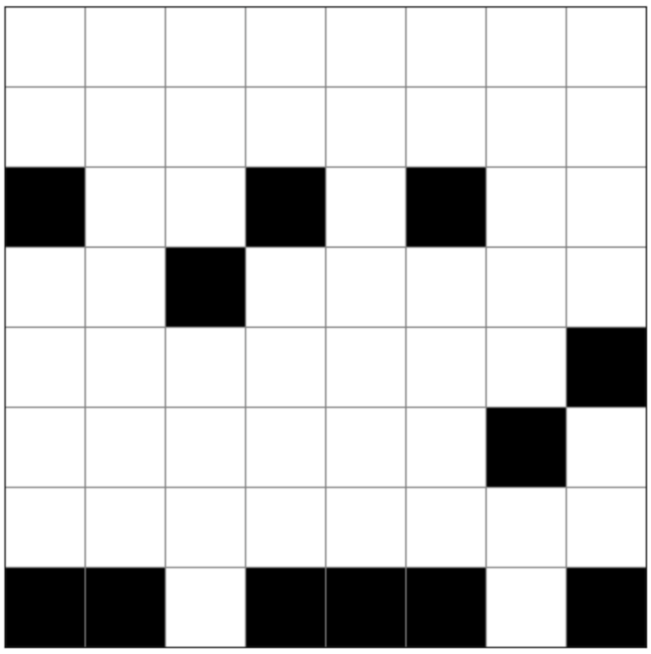
Implement in Python (or reuse existing open source) maze generator, capable of generating mazes of varying sizes

I have not used any open source maze creation algorithm instead I have coded my own. There are many maze creation algorithms like Kruskal's, Prim's, Recursive Backtracking, Aldous Border, Binary Tree but I have used **Recursive backtracking** because it was simple to understand and easy to code. The code can generate mazes of different rows and columns with randomized walls. Below is the screenshot of the sample maze created on 8x8 rows and columns.

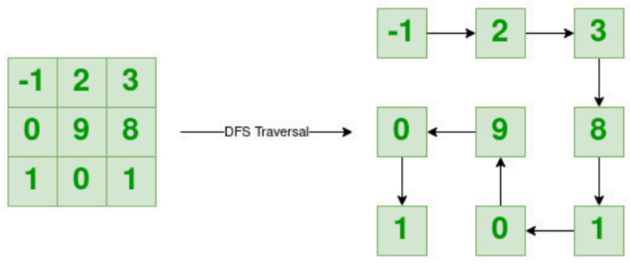


Where black boxes are walls and white boxes are free path to move around.

Implement search algorithms DFS, BFS, and A* for maze solving

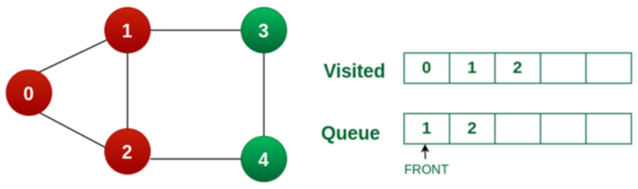
Here I will give the short description of all the algorithms I have implemented.

DFS



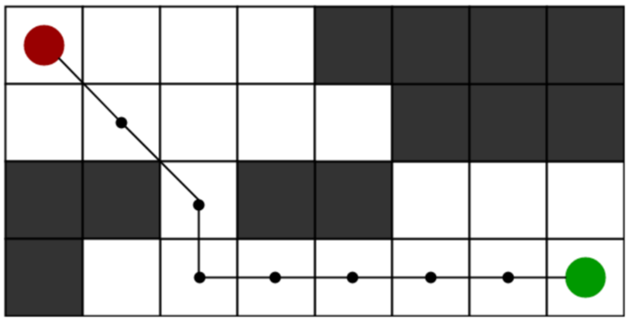
Stack data structure is used in DFS. Initially we push the (0,0) coordinate in the stack. Then check at every iteration for the valid moves and push the moves into the stack. When there is no moves available pop the stack until next move is not available. Repeat all of this until stack is not empty.

BFS



Queue data structure is used in BFS. It can also be termed as level order traversal. We perform the enqueue for the first cell (0,0). Also initialize Boolean array to keep track of visited nodes/moves. Now iterate till queue is not empty and perform Dequeue if cell is present in the front of queue. Check the next move for the given cell and move there and again enqueue then queue and mark them as visited.

A*



It is a heuristic search algorithm. It is faster and smarter than DFS and BFS. Heuristic cost is defined by the estimated cost of travel from current node to goal node. In each step of A*, it chooses all the nodes accordingly, then calculates f(g + h) for all nodes and choose the minimum.

Implement MDP value iteration and policy iteration algorithms for maze solving

MDP

MDP provides the mathematical framework for performing decision making in a stochastic ENV.

Value Iteration

It uses bellman equation to give us optimal value and it is defined as follows.

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

Here we start with $V_0(s) = 0$ for all states S and iterate it until convergence

Policy Iteration

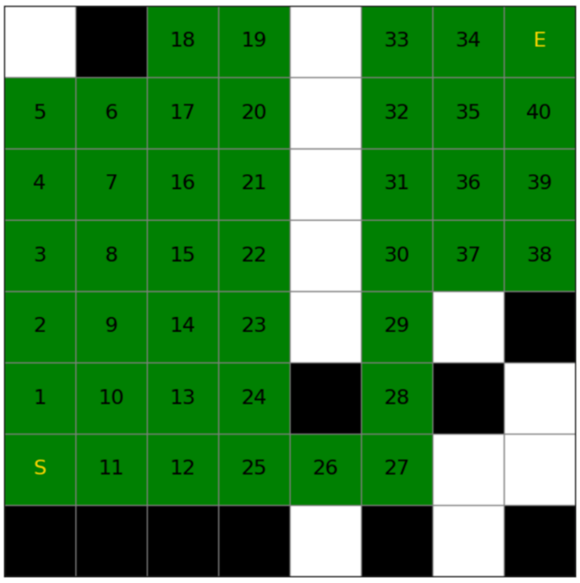
In the beginning it starts with random policy. It then calculate policy evaluation and policy improvement starting with generated random policy. It uses slightly modified but simple version of bellman equation where we don't select on the bases on MAX.

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

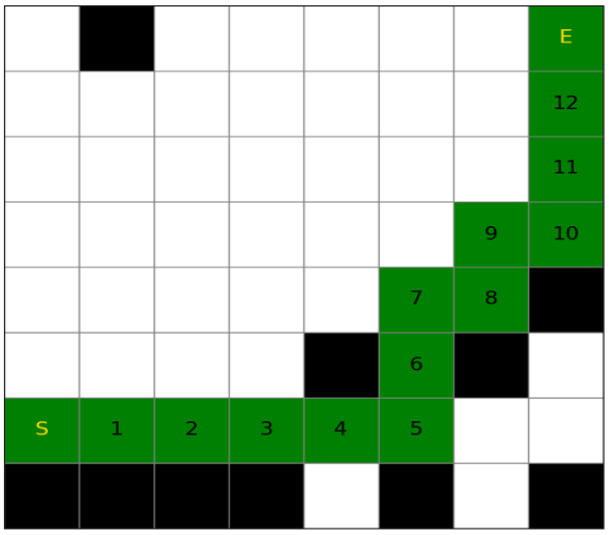
Analyse and discuss (1) the comparison between different search algorithms to each other

S in yellow colour represents starting cell and *E in yellow* colour represents ending cell. All *black cell* represents walls. *Green cells* represents traversed path. *Number* represents order of traversal. I ran all the algorithms on **8x8, 12x12 and 15x15** rows and columns. I didn't take huge number of cell because it would be difficult to visualize it here due to very small cell size.

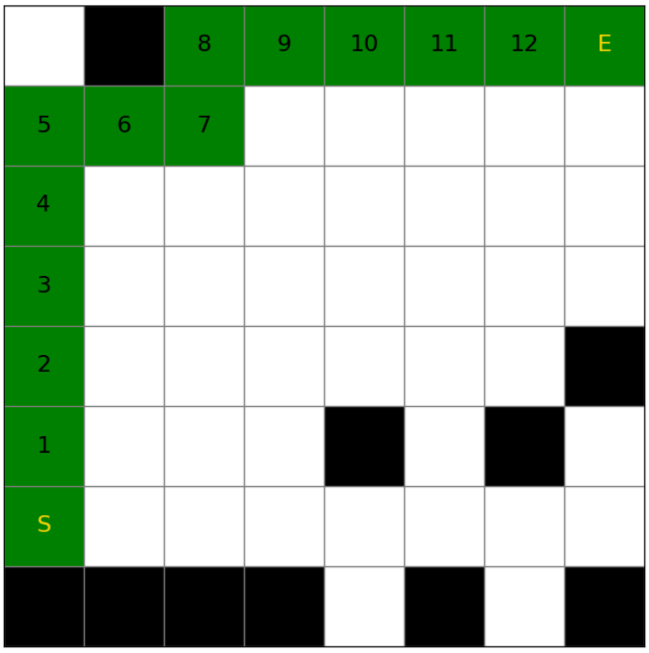
Comparison on 8x8 maze



8x8, starting cell: (6,0), ending cell: (0,7), **DFS**.

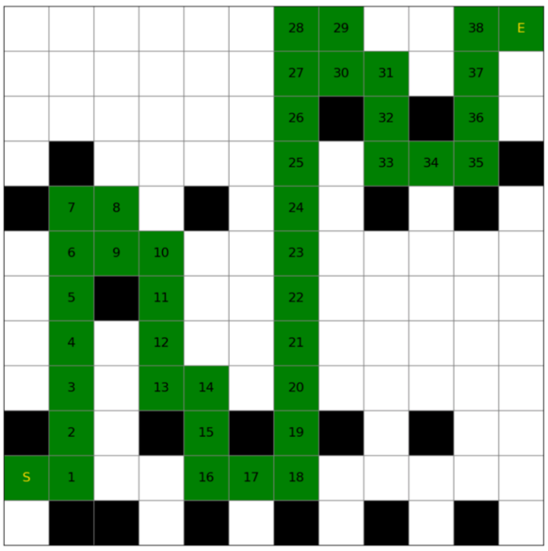


8x8, starting cell: (6,0), ending cell: (0,7), **BFS**.

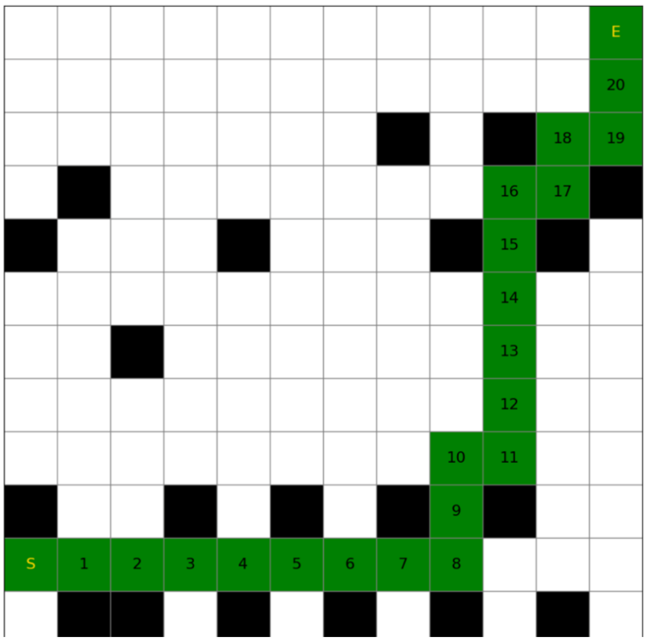


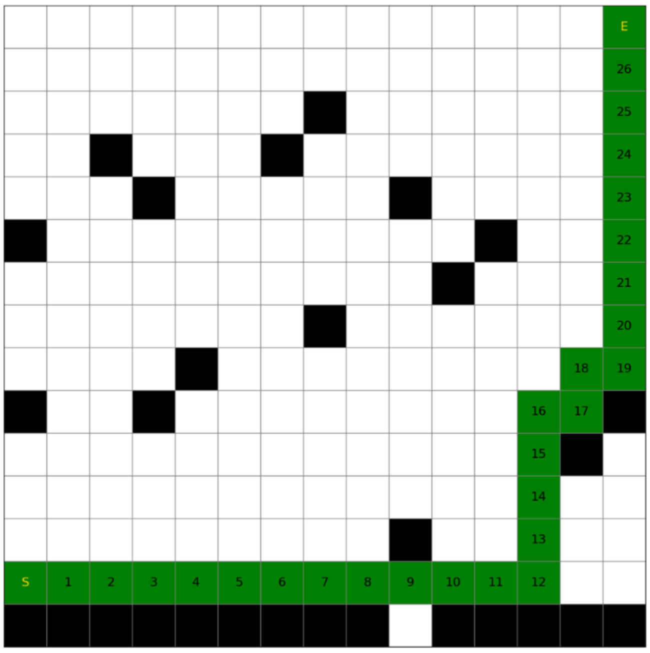
8x8, starting cell: (6,0), ending cell: (0,7), **A***.

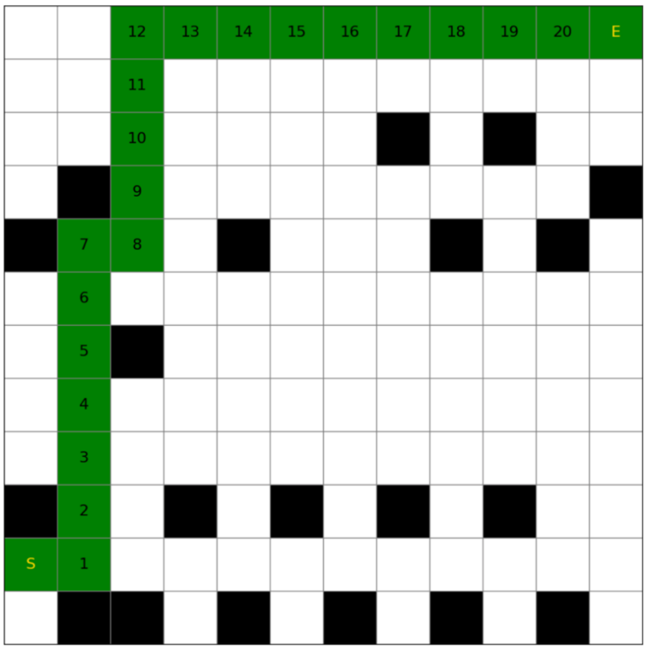
Comparison on 12x12 maze

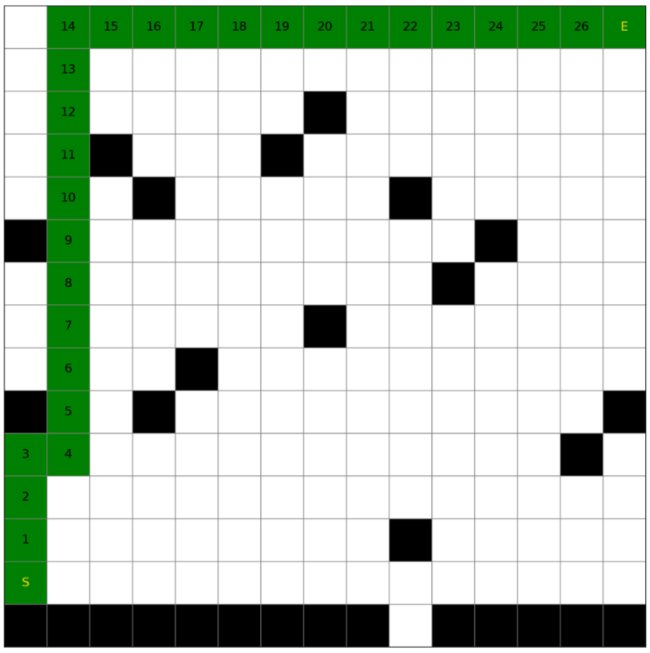


12x12, starting cell: (10,0), ending cell: (0,11), **DFS**.

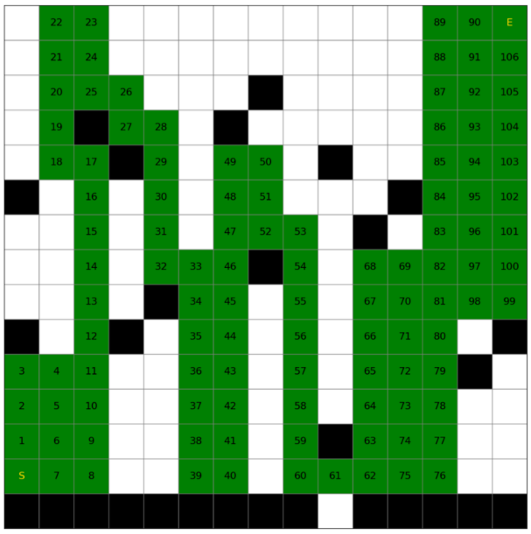

12x12, starting cell: (10,0), ending cell: (0,11), **BFS**.


15x15, starting cell: (13,0), ending cell: (0,14), **BFS**.


12x12, starting cell: (10,0), ending cell: (0,11), **A***.


15x15, starting cell: (13,0), ending cell: (0,14), **A***.

<u>Comparison on 15x15 maze</u>


15x15, starting cell: (13,0), ending cell: (0,14), **DFS**.

By analysing all the figures it is clear than DFS (Depth First Search) performed worst (not giving shortest path) in every case (8x8, 12x12 and 15x15) Whereas BFS performs better and A* performs best.

DFS uses stack data structure. The size of the stack will depend on the depth of the searching tree. That means size of the stack will be total number of nodes in the tree.

BFS is guaranteed to give shortest path in unweighted graph However DFS can perform same as BFS in some cases for example if goal node is just next to start node. The time complexity of BFS and DFS are same O(N).

where N is total number of cells visited in the maze. The space complexity of DFS is O(D) where D is the maximum depth of search tree and space complexity of BFS is O(N).

Performance of A* depends on the heuristic function and structure of the graph. It is important to take good heuristic function because sometimes it can overestimates the cost of reaching to the goal state in the worst case. There are two main heuristic functions: Manhattan and Euclidean. I took Manhattan because Manhattan is computationally inexpensive and best in pathfinding and route optimization problems. The time complexity of A* is O(b^d). where b is the branching factor (maximum number of neighbouring cells explored on given cell) and d is depth of graph till the goal state. The space complexity is same as the time complexity.

Search Algorithm	Maze Length	Path Length	Search Length	Time (sec)
DFS	8x8	40	44	0.000487
BFS	8x8	12	108	0.000381
A*	8x8	12	25	0.000219
DFS	12x12	38	91	0.000895
BFS	12x12	20	248	0.000745
A*	12x12	20	46	0.000368
DFS	15x15	106	119	0.001204
BFS	15x15	26	392	0.001470
A*	15x15	26	59	0.000488

The table above is the recorded performance of each algorithm during analysis. I have taken 3 metrics to compare the performance i.e. ***Path Length, Search Length and time taken*** to find goal state. As we can see DFS performed worst by giving the maximum path length. BFS and A* is giving the shortest path but BFS is slower than A*. In the beginning BFS and A* doesn't show differentiable change in time even though the complexity of both algorithms are different because of small input size i.e. maze length. The difference is significant in 15x15 maze size, A* performing much better on high input size.

DFS has lower search length than BFS because the DFS I coded will choose first direction Up and then left so it is easy to perform traversal from bottom left to upper right just like our starting cell and ending cell. But if we change the goal and starting cell the results will change.

Verdict – A* is generally the best algorithm to solve maze like problems. BFS is good choice to get shortest path if you don't care about the search length. DFS is good to explore all possible paths.

---

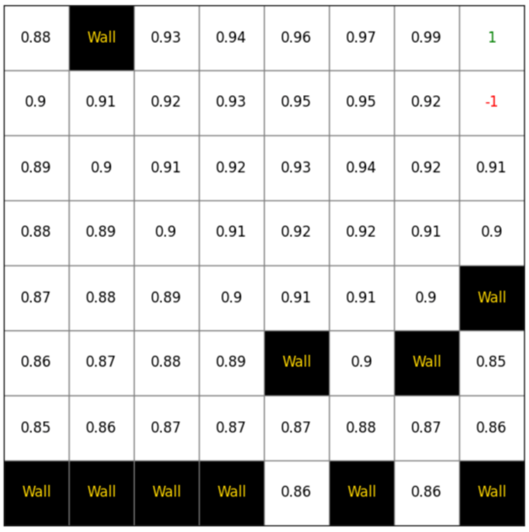
(2) comparison between different MDP algorithms to each other

Both algorithms are used in reinforcement learning. Value iteration updates the value function until it reaches the convergence. Policy Iteration focuses on improving the policies. We first start will random policy then we calculate its value function. Policy improvement involves selecting the maximum value based on current value.

Value iteration is faster but converges later than Policy iteration. So if there is computational limitation, use value iteration but if not, policy iteration is faster. This shows that we cannot minimise the space and time complexity at the same time. We have to compromise either in space or time to achieve one.

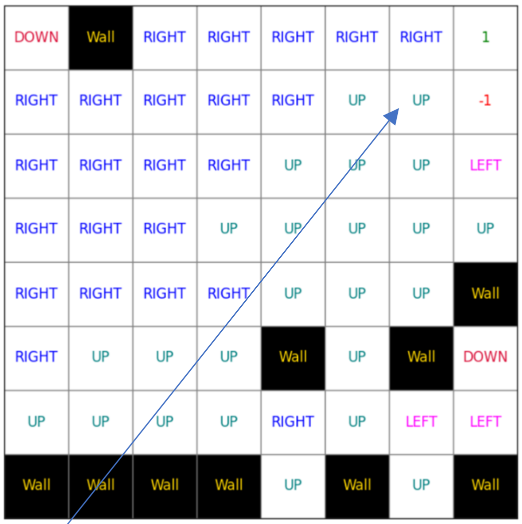
Following plots are the results of **value iteration** on different living rewards (negative and zero only).

For all the plots discount = 0.99 and max_error/epsilon = 0.001. 1 represents the goal state.



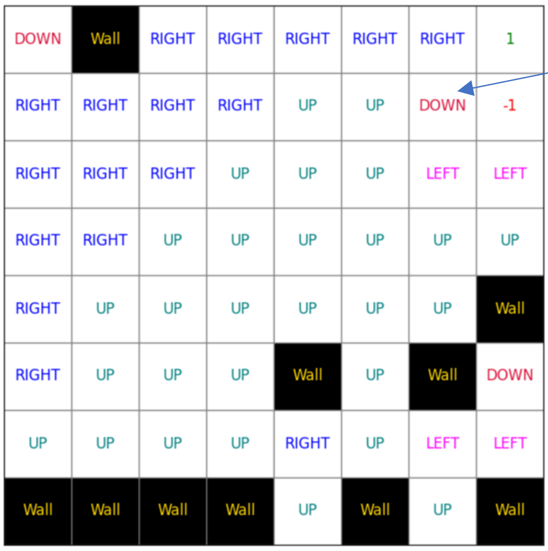
0.88	Wall	0.93	0.94	0.96	0.97	0.99	1
0.9	0.91	0.92	0.93	0.95	0.95	0.92	-1
0.89	0.9	0.91	0.92	0.93	0.94	0.92	0.91
0.88	0.89	0.9	0.91	0.92	0.92	0.91	0.9
0.87	0.88	0.89	0.9	0.91	0.91	0.9	Wall
0.86	0.87	0.88	0.89	Wall	0.9	Wall	0.85
0.85	0.86	0.87	0.87	0.87	0.88	0.87	0.86
Wall	Wall	Wall	Wall	0.86	Wall	0.86	Wall

Values after value iteration 8x8
reward = 0, discount = 0.99, epsilon = 0.001



DOWN	Wall	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	-1
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	LEFT
RIGHT	RIGHT	RIGHT	UP	UP	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall
RIGHT	UP	UP	UP	Wall	UP	Wall	DOWN
UP	UP	UP	UP	RIGHT	UP	LEFT	LEFT
Wall	Wall	Wall	Wall	UP	Wall	UP	Wall

Optimal Policy after value iteration 8x8
reward = -0.2, discount = 0.99, epsilon = 0.001

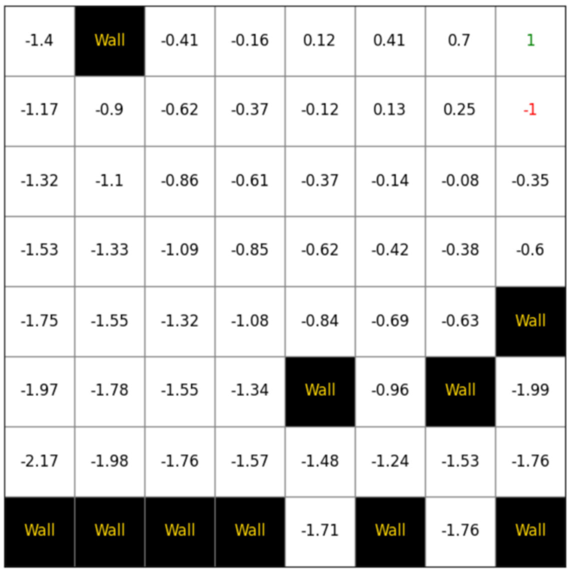


DOWN	Wall	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	DOWN	-1
RIGHT	RIGHT	RIGHT	UP	UP	UP	LEFT	LEFT
RIGHT	RIGHT	UP	UP	UP	UP	UP	UP
RIGHT	UP	UP	UP	UP	UP	UP	Wall
RIGHT	UP	UP	UP	Wall	UP	Wall	DOWN
UP	UP	UP	UP	RIGHT	UP	LEFT	LEFT
Wall	Wall	Wall	Wall	UP	Wall	UP	Wall

Optimal Policy after value iteration 8x8
reward = 0, discount = 0.99, epsilon = 0.001

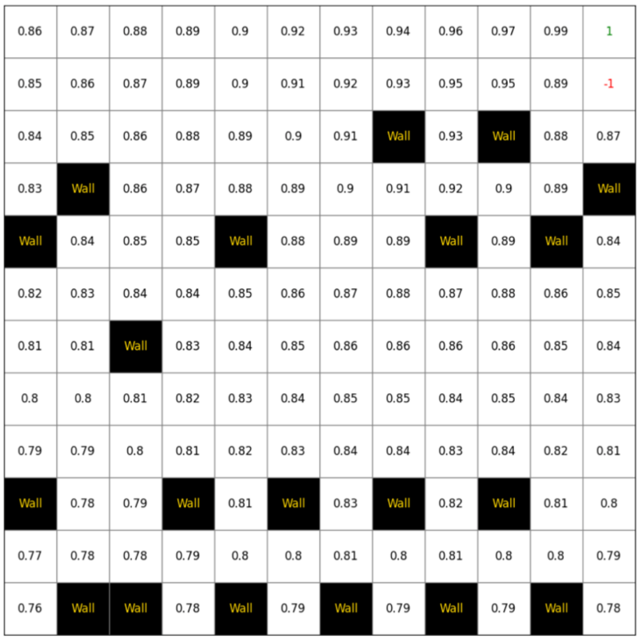
Here we can see there is change in policy after introducing the negative reward. Agent will less number of steps in negative reward. This is because MDP focuses on maximizing reward. So if we introduce negative reward, agent will try to spend less time in the environment by reaching to goal state faster.

Below are the some plots of different maze sizes of value iteration.



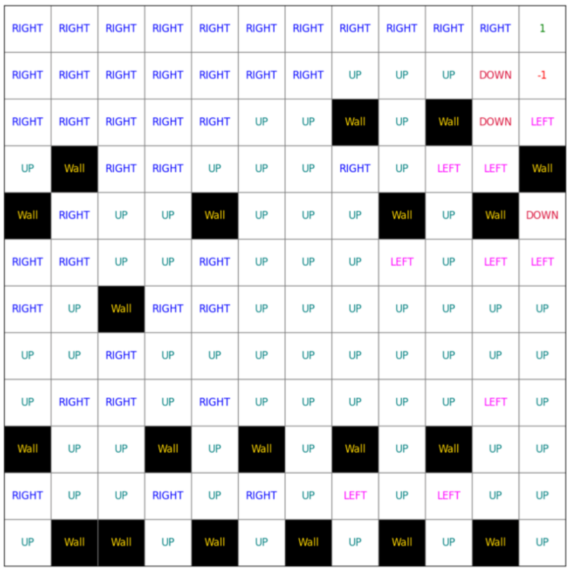
-1.4	Wall	-0.41	-0.16	0.12	0.41	0.7	1
-1.17	-0.9	-0.62	-0.37	-0.12	0.13	0.25	-1
-1.32	-1.1	-0.86	-0.61	-0.37	-0.14	-0.08	-0.35
-1.53	-1.33	-1.09	-0.85	-0.62	-0.42	-0.38	-0.6
-1.75	-1.55	-1.32	-1.08	-0.84	-0.69	-0.63	Wall
-1.97	-1.78	-1.55	-1.34	Wall	-0.96	Wall	-1.99
-2.17	-1.98	-1.76	-1.57	-1.48	-1.24	-1.53	-1.76
Wall	Wall	Wall	Wall	-1.71	Wall	-1.76	Wall

Values after value iteration 8x8
reward = -0.2, discount = 0.99, epsilon = 0.001



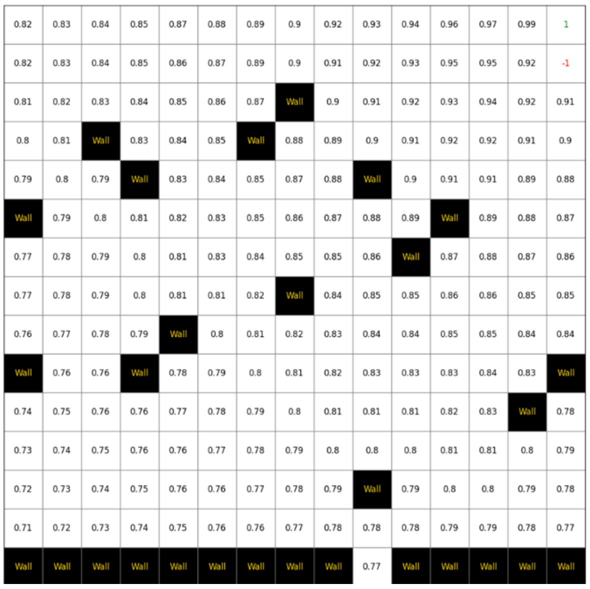
0.86	0.87	0.88	0.89	0.9	0.92	0.93	0.94	0.96	0.97	0.99	1
0.85	0.86	0.87	0.89	0.9	0.91	0.92	0.93	0.95	0.95	0.89	-1
0.84	0.85	0.86	0.88	0.89	0.9	0.91	Wall	0.93	Wall	0.88	0.87
0.83	Wall	0.86	0.87	0.88	0.89	0.9	0.91	0.92	0.9	0.89	Wall
Wall	0.84	0.85	0.85	Wall	0.88	0.89	0.89	Wall	0.89	Wall	0.84
0.82	0.83	0.84	0.84	0.85	0.86	0.87	0.88	0.87	0.88	0.86	0.85
0.81	0.81	Wall	0.83	0.84	0.85	0.86	0.86	0.86	0.86	0.85	0.84
0.8	0.8	0.81	0.82	0.83	0.84	0.85	0.85	0.84	0.85	0.84	0.83
0.79	0.79	0.8	0.81	0.82	0.83	0.84	0.84	0.83	0.84	0.82	0.81
Wall	0.78	0.79	Wall	0.81	Wall	0.83	Wall	0.82	Wall	0.81	0.8
0.77	0.78	0.78	0.79	0.8	0.8	0.81	0.8	0.81	0.8	0.8	0.79
0.76	Wall	Wall	0.78	Wall	0.79	Wall	0.79	Wall	0.79	Wall	0.78

Values after value iteration 12x12
reward = 0, discount = 0.99, epsilon = 0.001



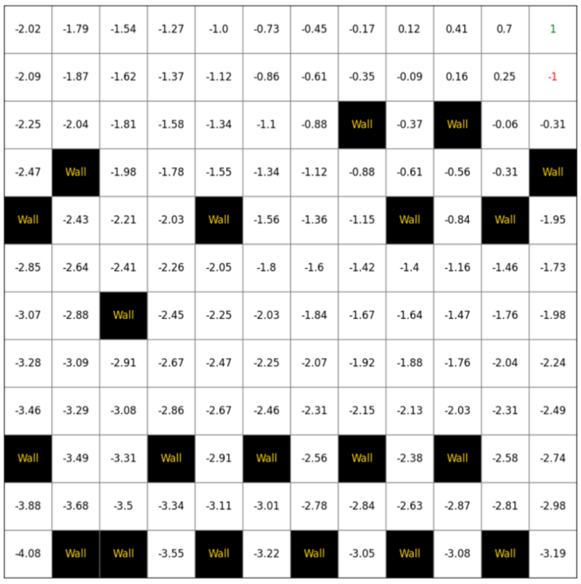
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	DOWN	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	Wall	UP	Wall	DOWN	LEFT
UP	Wall	RIGHT	RIGHT	UP	UP	UP	RIGHT	UP	LEFT	LEFT	Wall
Wall	RIGHT	UP	UP	Wall	UP	UP	UP	Wall	UP	Wall	DOWN
RIGHT	RIGHT	UP	UP	RIGHT	UP	UP	UP	LEFT	UP	LEFT	LEFT
RIGHT	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP
UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	RIGHT	UP	UP	UP	UP	UP	LEFT	UP
Wall	UP	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP	UP
RIGHT	UP	UP	RIGHT	UP	RIGHT	UP	LEFT	UP	LEFT	UP	UP
UP	Wall	Wall	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP

Optimal Policy after value iteration 12x12
reward = 0, discount = 0.99, epsilon = 0.001



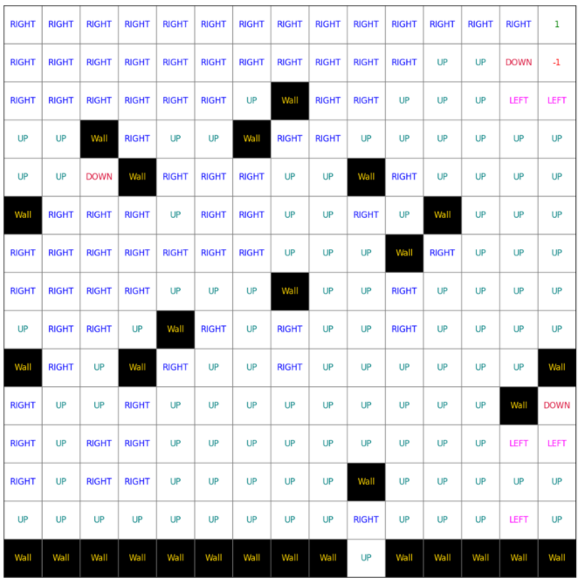
0.82	0.83	0.84	0.85	0.87	0.88	0.89	0.9	0.92	0.93	0.94	0.96	0.97	0.99	1
0.82	0.83	0.84	0.85	0.86	0.87	0.89	0.9	0.91	0.92	0.93	0.95	0.95	0.92	-1
0.81	0.82	0.83	0.84	0.85	0.86	0.87	Wall	0.9	0.91	0.92	0.93	0.94	0.92	0.91
0.8	0.81	Wall	0.83	0.84	0.85	Wall	0.88	0.89	0.9	0.91	0.92	0.92	0.91	0.9
0.79	0.8	0.79	Wall	0.83	0.84	0.85	0.87	0.88	Wall	0.9	0.91	0.91	0.89	0.88
Wall	0.79	0.8	0.81	0.82	0.83	0.85	0.86	0.87	0.88	0.89	Wall	0.89	0.88	0.87
0.77	0.78	0.79	0.8	0.81	0.83	0.84	0.85	0.85	0.86	Wall	0.87	0.88	0.87	0.86
0.77	0.78	0.79	0.8	0.81	0.81	0.82	Wall	0.84	0.85	0.85	0.86	0.86	0.85	0.85
0.76	0.77	0.78	0.79	Wall	0.8	0.81	0.82	0.83	0.84	0.84	0.85	0.85	0.84	0.84
Wall	0.76	0.76	Wall	0.78	0.79	0.8	0.81	0.82	0.83	0.83	0.83	0.84	0.83	Wall
0.74	0.75	0.76	0.76	0.77	0.78	0.79	0.8	0.81	0.81	0.81	0.82	0.83	Wall	0.78
0.73	0.74	0.75	0.76	0.76	0.77	0.78	0.79	0.8	0.8	0.8	0.81	0.81	0.8	0.79
0.72	0.73	0.74	0.75	0.76	0.76	0.77	0.78	0.79	Wall	0.79	0.8	0.8	0.79	0.78
0.71	0.72	0.73	0.74	0.75	0.76	0.76	0.77	0.78	0.78	0.78	0.79	0.79	0.78	0.77
Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	0.77	Wall	Wall	Wall	Wall	Wall

Values after value iteration 15x15
reward = 0, discount = 0.99, epsilon = 0.001



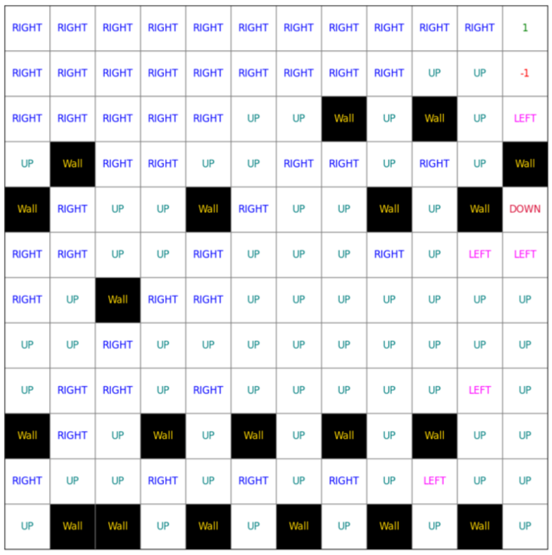
-2.02	-1.79	-1.54	-1.27	-1.0	-0.73	-0.45	-0.17	0.12	0.41	0.7	1
-2.09	-1.87	-1.62	-1.37	-1.12	-0.86	-0.61	-0.35	-0.09	0.16	0.25	-1
-2.25	-2.04	-1.81	-1.58	-1.34	-1.1	-0.88	Wall	-0.37	Wall	-0.06	-0.31
-2.47	Wall	-1.98	-1.78	-1.55	-1.34	-1.12	-0.88	-0.61	-0.56	-0.31	Wall
Wall	-2.43	-2.21	-2.03	Wall	-1.56	-1.36	-1.15	Wall	-0.84	Wall	-1.95
-2.85	-2.64	-2.41	-2.26	-2.05	-1.8	-1.6	-1.42	-1.4	-1.16	-1.46	-1.73
-3.07	-2.88	Wall	-2.45	-2.25	-2.03	-1.84	-1.67	-1.64	-1.47	-1.76	-1.98
-3.28	-3.09	-2.91	-2.67	-2.47	-2.25	-2.07	-1.92	-1.88	-1.76	-2.04	-2.24
-3.46	-3.29	-3.08	-2.86	-2.67	-2.46	-2.31	-2.15	-2.13	-2.03	-2.31	-2.49
Wall	-3.49	-3.31	Wall	-2.91	Wall	-2.56	Wall	-2.38	Wall	-2.58	-2.74
-3.88	-3.68	-3.5	-3.34	-3.11	-3.01	-2.78	-2.84	-2.63	-2.87	-2.81	-2.98
-4.08	Wall	Wall	-3.55	Wall	-3.22	Wall	-3.05	Wall	-3.08	Wall	-3.19

Values after value iteration 12x12
reward = -0.2, discount = 0.99, epsilon = 0.001



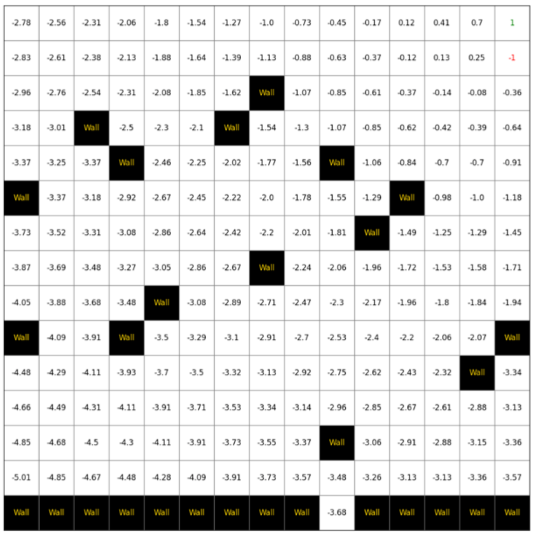
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	DOWN	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	Wall	RIGHT	RIGHT	UP	UP	UP	LEFT	LEFT
UP	UP	Wall	RIGHT	UP	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP
UP	UP	DOWN	Wall	RIGHT	RIGHT	RIGHT	UP	UP	Wall	RIGHT	UP	UP	UP	UP
Wall	RIGHT	RIGHT	RIGHT	UP	RIGHT	RIGHT	UP	UP	RIGHT	UP	Wall	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	RIGHT	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	UP	UP	RIGHT	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	Wall	RIGHT	UP	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP
Wall	RIGHT	UP	Wall	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	Wall
RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	Wall	DOWN
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	LEFT	LEFT
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	Wall	UP	UP	UP	UP	UP
UP	UP	UP	UP	UP	UP	UP	UP	UP	RIGHT	UP	UP	UP	LEFT	UP
Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	UP	Wall	Wall	Wall	Wall	Wall

Optimal Policy after value iteration 15x15
reward = 0, discount = 0.99, epsilon = 0.001



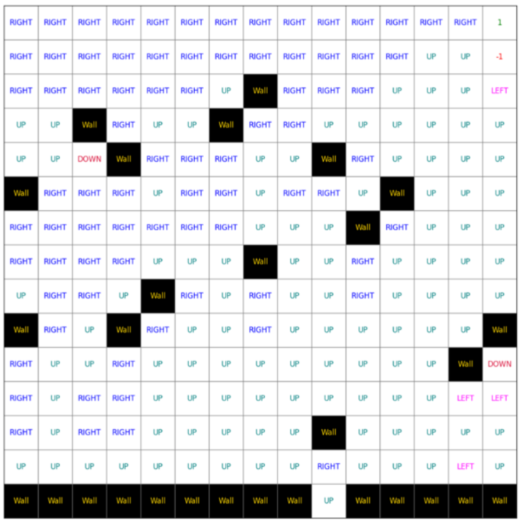
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	Wall	UP	Wall	UP	LEFT
UP	Wall	RIGHT	RIGHT	UP	UP	RIGHT	RIGHT	UP	RIGHT	UP	Wall
Wall	RIGHT	UP	UP	Wall	RIGHT	UP	UP	Wall	UP	Wall	DOWN
RIGHT	RIGHT	UP	UP	RIGHT	UP	UP	UP	RIGHT	UP	LEFT	LEFT
RIGHT	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP
UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	RIGHT	UP	UP	UP	UP	UP	LEFT	UP
Wall	RIGHT	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP	UP
RIGHT	UP	UP	RIGHT	UP	RIGHT	UP	RIGHT	UP	LEFT	UP	UP
UP	Wall	Wall	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP

Optimal Policy after value iteration 12x12
reward = -0.2, discount = 0.99, epsilon = 0.001



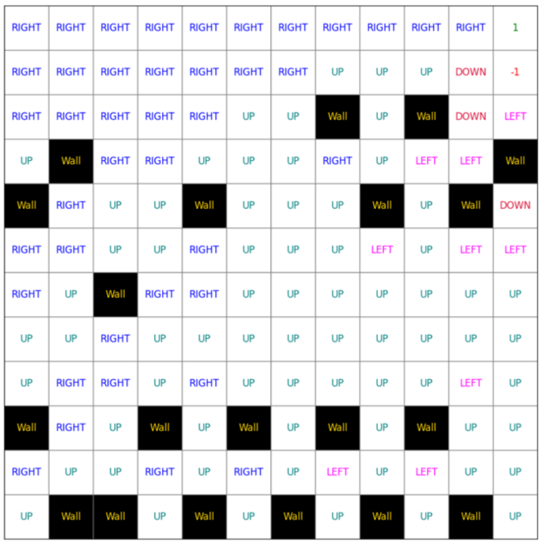
-2.78	-2.56	-2.31	-2.06	-1.8	-1.54	-1.27	-1.0	-0.73	-0.45	-0.17	0.12	0.41	0.7	1
-2.83	-2.61	-2.38	-2.13	-1.88	-1.64	-1.39	-1.13	-0.88	-0.63	-0.37	-0.12	0.13	0.25	-1
-2.96	-2.76	-2.54	-2.31	-2.08	-1.85	-1.62	Wall	-1.07	-0.85	-0.61	-0.37	-0.14	-0.08	-0.36
-3.18	-3.01	Wall	-2.5	-2.3	-2.1	Wall	-1.54	-1.3	-1.07	-0.85	-0.62	-0.42	-0.39	-0.64
-3.37	-3.25	-3.37	Wall	-2.46	-2.25	-2.02	-1.77	-1.56	Wall	-1.06	-0.84	-0.7	-0.7	-0.91
Wall	-3.37	-3.18	-2.92	-2.67	-2.45	-2.22	-2.0	-1.78	-1.55	-1.29	Wall	-0.98	-1.0	-1.18
-3.73	-3.52	-3.31	-3.08	-2.86	-2.64	-2.42	-2.2	-2.01	-1.81	Wall	-1.49	-1.25	-1.29	-1.45
-3.87	-3.69	-3.48	-3.27	-3.05	-2.86	-2.67	Wall	-2.24	-2.06	-1.96	-1.72	-1.53	-1.58	-1.71
-4.05	-3.88	-3.68	-3.48	Wall	-3.08	-2.89	-2.71	-2.47	-2.3	-2.17	-1.96	-1.8	-1.84	-1.94
Wall	-4.09	-3.91	Wall	-3.5	-3.29	-3.1	-2.91	-2.7	-2.53	-2.4	-2.2	-2.06	-2.07	Wall
-4.48	-4.29	-4.11	-3.93	-3.7	-3.5	-3.32	-3.13	-2.92	-2.75	-2.62	-2.43	-2.32	Wall	-3.34
-4.66	-4.49	-4.31	-4.11	-3.91	-3.71	-3.53	-3.34	-3.14	-2.96	-2.85	-2.67	-2.61	-2.88	-3.13
-4.85	-4.68	-4.5	-4.3	-4.11	-3.91	-3.73	-3.55	-3.37	Wall	-3.06	-2.91	-2.88	-3.15	-3.36
-5.01	-4.85	-4.67	-4.48	-4.28	-4.09	-3.91	-3.73	-3.57	-3.48	-3.26	-3.13	-3.13	-3.36	-3.57
Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	-3.68	Wall	Wall	Wall	Wall	Wall

Values after value iteration 15x15
reward = -0.2, discount = 0.99, epsilon = 0.001



RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	Wall	RIGHT	RIGHT	RIGHT	UP	UP	UP	LEFT
UP	UP	Wall	RIGHT	UP	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP
UP	UP	DOWN	Wall	RIGHT	RIGHT	RIGHT	UP	UP	Wall	RIGHT	UP	UP	UP	UP
Wall	RIGHT	RIGHT	RIGHT	UP	RIGHT	RIGHT	UP	RIGHT	RIGHT	UP	Wall	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	RIGHT	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	UP	UP	RIGHT	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	Wall	RIGHT	UP	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP
Wall	RIGHT	UP	Wall	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	Wall
RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	Wall	DOWN
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	LEFT	LEFT
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	Wall	UP	UP	UP	UP	UP
UP	UP	UP	UP	UP	UP	UP	UP	UP	RIGHT	UP	UP	UP	LEFT	UP
Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	UP	Wall	Wall	Wall	Wall	Wall

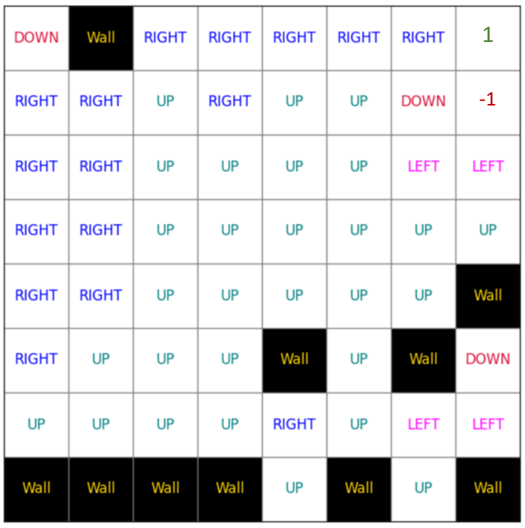
Optimal Policy after value iteration 15x15
reward = -0.2, discount = 0.99, epsilon = 0.001



RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	DOWN	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	Wall	UP	Wall	DOWN	LEFT
UP	Wall	RIGHT	RIGHT	UP	UP	UP	RIGHT	UP	LEFT	LEFT	Wall
Wall	RIGHT	UP	UP	Wall	UP	UP	UP	Wall	UP	Wall	DOWN
RIGHT	RIGHT	UP	UP	RIGHT	UP	UP	UP	LEFT	UP	LEFT	LEFT
RIGHT	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP
UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	RIGHT	UP	UP	UP	UP	UP	LEFT	UP
Wall	RIGHT	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP	UP
RIGHT	UP	UP	RIGHT	UP	RIGHT	UP	LEFT	UP	LEFT	UP	UP
UP	Wall	Wall	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP

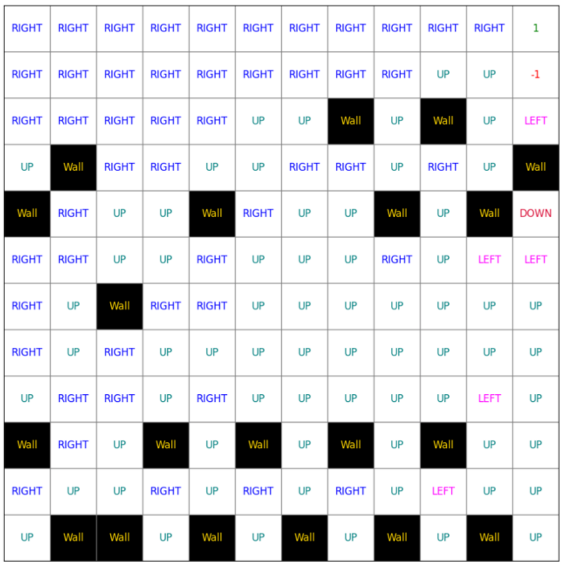
Optimal Policy after policy iteration 12x12
reward = 0, discount = 0.99, epsilon = 0.001

Now below are the plots of Policy Iteration on different maze sizes and living reward



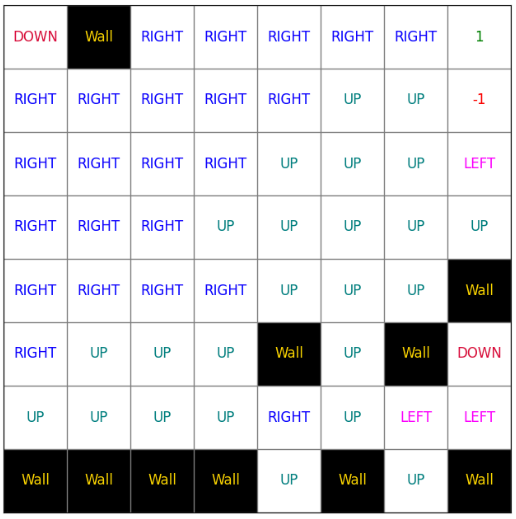
DOWN	Wall	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	UP	RIGHT	UP	UP	DOWN	-1
RIGHT	RIGHT	UP	UP	UP	UP	LEFT	LEFT
RIGHT	RIGHT	UP	UP	UP	UP	UP	UP
RIGHT	RIGHT	UP	UP	UP	UP	UP	Wall
RIGHT	UP	UP	UP	Wall	UP	Wall	DOWN
UP	UP	UP	UP	RIGHT	UP	LEFT	LEFT
Wall	Wall	Wall	Wall	UP	Wall	UP	Wall

Optimal Policy after policy iteration 8x8
reward = 0, discount = 0.99, epsilon = 0.001



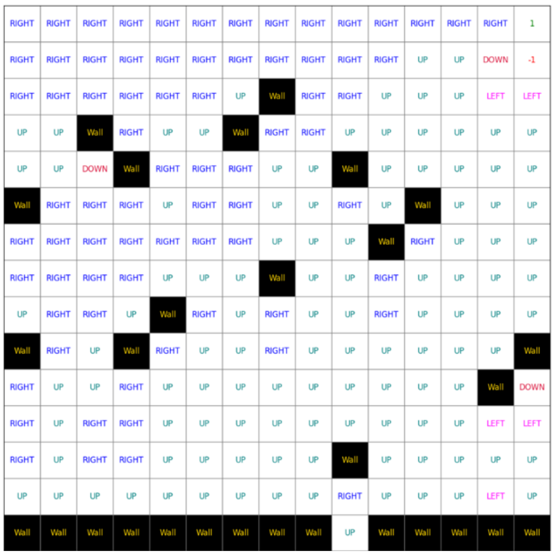
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	Wall	UP	Wall	UP	LEFT
UP	Wall	RIGHT	RIGHT	UP	UP	RIGHT	RIGHT	UP	RIGHT	UP	Wall
Wall	RIGHT	UP	UP	Wall	RIGHT	UP	UP	Wall	UP	Wall	DOWN
RIGHT	RIGHT	UP	UP	RIGHT	UP	UP	UP	RIGHT	UP	LEFT	LEFT
RIGHT	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP
RIGHT	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	RIGHT	UP	UP	UP	UP	UP	LEFT	UP
Wall	RIGHT	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP	UP
RIGHT	UP	UP	RIGHT	UP	RIGHT	UP	RIGHT	UP	LEFT	UP	UP
UP	Wall	Wall	UP	Wall	UP	Wall	UP	Wall	UP	Wall	UP

Optimal Policy after policy iteration 12x12
reward = -0.2, discount = 0.99, epsilon = 0.001



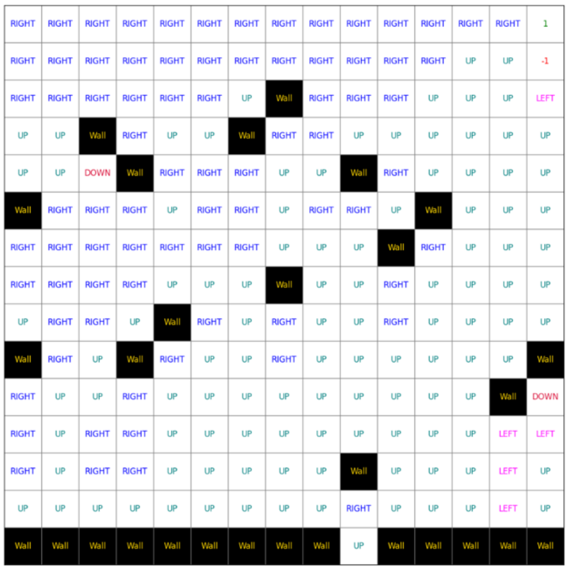
DOWN	Wall	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	-1
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	LEFT
RIGHT	RIGHT	RIGHT	UP	UP	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall
RIGHT	UP	UP	UP	Wall	UP	Wall	DOWN
UP	UP	UP	UP	RIGHT	UP	LEFT	LEFT
Wall	Wall	Wall	Wall	UP	Wall	UP	Wall

Optimal Policy after policy iteration 8x8
reward = -0.2, discount = 0.99, epsilon = 0.001



RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	DOWN	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	Wall	RIGHT	RIGHT	UP	UP	UP	LEFT	LEFT
UP	UP	Wall	RIGHT	UP	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP
UP	UP	DOWN	Wall	RIGHT	RIGHT	RIGHT	UP	UP	Wall	UP	UP	UP	UP	UP
Wall	RIGHT	RIGHT	RIGHT	UP	RIGHT	RIGHT	UP	UP	RIGHT	UP	Wall	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	RIGHT	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	UP	UP	RIGHT	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	Wall	RIGHT	UP	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP
Wall	RIGHT	UP	Wall	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	Wall
RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	Wall	DOWN
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	LEFT	LEFT
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	Wall	UP	UP	UP	UP	UP
UP	UP	UP	UP	UP	UP	UP	UP	UP	RIGHT	UP	UP	UP	LEFT	UP
Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	UP	Wall	Wall	Wall	Wall	Wall

Optimal Policy after policy iteration 15x15
reward = 0, discount = 0.99, epsilon = 0.001



RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	-1
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	Wall	RIGHT	RIGHT	RIGHT	UP	UP	UP	LEFT
UP	UP	Wall	RIGHT	UP	UP	Wall	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP
UP	UP	DOWN	Wall	RIGHT	RIGHT	RIGHT	UP	UP	Wall	RIGHT	UP	UP	UP	UP
Wall	RIGHT	RIGHT	RIGHT	UP	RIGHT	RIGHT	UP	RIGHT	RIGHT	UP	Wall	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	RIGHT	UP	UP	UP
RIGHT	RIGHT	RIGHT	RIGHT	UP	UP	UP	Wall	UP	UP	RIGHT	UP	UP	UP	UP
UP	RIGHT	RIGHT	UP	Wall	RIGHT	UP	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP
Wall	RIGHT	UP	Wall	RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	Wall
RIGHT	UP	UP	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	Wall	DOWN
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	UP	UP	UP	UP	LEFT	LEFT
RIGHT	UP	RIGHT	RIGHT	UP	UP	UP	UP	UP	Wall	UP	UP	UP	LEFT	UP
UP	UP	UP	UP	UP	UP	UP	UP	UP	RIGHT	UP	UP	UP	LEFT	UP
Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	UP	Wall	Wall	Wall	Wall	Wall

Optimal Policy after policy iteration 15x15
reward = -0.2, discount = 0.99, epsilon = 0.001

MDP Algorithms	Living Reward	Maze Length	Time consumption	Iterations
Value	0	8x8	0.054	55
Policy	0	8x8	0.024	15
Value	0	12x12	0.157	63
Policy	0	12x12	0.087	24
Value	0	15x15	0.266	69
Policy	0	15x15	0.200	32
Value	-0.2	8x8	0.046	42
Policy	-0.2	8x8	0.019	12
Value	-0.2	12x12	0.138	55
Policy	-0.2	12x12	0.088	24
Value	-0.2	15x15	0.255	62
Policy	-0.2	15x15	0.171	29

The above table is the recorded metrics to measure the performance of the MDP algorithms. The Metrics are time consumption and total iteration performed to reach convergence. As we can see Value iteration is taking more time and iteration to reach convergence whereas policy iteration is taking less time and iterations to reach convergence. Also for both algorithms convergence time is increasing as number of maze size increases.

My choice of max_error = 0.001 because convergence is obtained when maximum difference between current and previous state is below this threshold. That's why this number should be less.

My discount factor = 0.99. It effects the future rewards in next iterations from previous iterations. The value should be less then 1 to prevent infinite loop problem.

(3) comparison between search and MDP algorithms.

After analysing all these different algorithms, I came to verdict that if shortest path finding is the only goal them go for A* because it performs better than BFS and DFS. But if the environment Is big, unpredictable and dynamic then use MDP algorithms. Value iteration is resources are limited or policy iteration if speed is what matters.

Submitted By : Chirag Saxena, Student Id : 22312293

Appendix:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:



import pandas as pd import
numpy as np import
matplotlib.pyplot as plt import
matplotlib.colors import cv2
import random
import time



#
# 1. Implement in Python (or reuse existing open source) maze generator, capable of
generating mazes of varying sizes

# In[2]:
```

#This following code is for maze visualization.

```
def viz(matrix,start,end,fs,path=False):

   n_rows = matrix.shape[0]
n_cols = matrix.shape[1]     matrix
= np.copy(matrix)     for i in
range(matrix.shape[0]):         for j in
range(matrix.shape[1]):            if
matrix[i][j]==1:               matrix[i][j]
= 0            elif matrix[i][j]==0:
matrix[i][j] = 1

   fig, ax = plt.subplots()
fig.set_size_inches(n_cols, n_rows)     if
path:
      cmap = matplotlib.colors.ListedColormap(['white', 'black','green'])
for i in range(len(path)):
         a,b = path[i]            if
i>0 and i<len(path)-1:
```

```
            ax.text(b, a, i , ha='center', va='center', color='black',fontsize = fs)
matrix[a][b] = 2
   else:
       cmap = matplotlib.colors.ListedColormap(['white', 'black'])
ax.imshow(matrix, cmap=cmap, origin='upper')     ax.set_xticks(np.arange(-0.5,
n_cols, 1), minor=True)     ax.set_yticks(np.arange(-0.5, n_rows, 1),
minor=True)     ax.grid(which='minor', color='gray', linestyle='-', linewidth=1)
ax.set_yticks(np.arange(-0.5, n_rows, 1))
   ax.grid(which='major', axis='y', color='grey', linestyle='-', linewidth=1)
ax.tick_params(axis='both', which='both', length=0)     a,b = start
   ax.text(b, a, 'S', ha='center', va='center', color='gold',fontsize = fs)
a,b = end
   ax.text(b, a, 'E', ha='center', va='center', color='gold',fontsize = fs)
ax.set_xticklabels([])     ax.set_yticklabels([])

   plt.show()


# In[3]:

#Following is the Maze Generation code.

#20,30,50
rows = 5 cols
= 5
starti = 0
startj = 0


# In[4]:



class Cell():     def
__init__(self,i,j):
self.i = i        self.j = j
self.visited = False
      self.walls = [False,False,False,False] #top,bottom,left,right
if i==0:
         self.walls[0] = True
if j==0:
         self.walls[2] = True
if i==rows-1:
         self.walls[1] = True
if j == cols-1:
```

```
            self.walls[3] = True




# In[5]:


maze = np.empty((rows, cols), dtype=object)
for i in range(rows):     for j in range(cols):
maze[i][j] = Cell(i,j)


# In[6]:


def helper(starti,startj):
    c_i = starti
c_j = startj
    walls = maze[starti][startj].walls

    neighbour = []     k = -1     if not walls[0] and
not maze[c_i-1][c_j].visited:
        k = 0
        neighbour.append((c_i-1,c_j))     if not
walls[3] and not maze[c_i][c_j+1].visited:
        k = 3
        neighbour.append((c_i,c_j+1))     if not
walls[1] and not maze[c_i+1][c_j].visited:
        k = 1
        neighbour.append((c_i+1,c_j))     if not
walls[2] and not maze[c_i][c_j-1].visited:
        k = 3
        neighbour.append((c_i,c_j-1))     if
len(neighbour)>0:         r =
random.randint(0, len(neighbour)-1)
a,b = neighbour[r]         maze[a][b].visited =
True         maze[a][b].walls[k] = True
return neighbour[r]     else:
        return -1,-1


# In[7]:
```

```
mazebool = np.zeros((rows, cols))


# In[8]:


def reset(n):
    mazebool1 = [[1., 0., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 0.],
      [1., 1., 1., 1., 0., 1., 0., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1.],
[0., 0., 0., 0., 1., 0., 1., 0.]]
    mazebool1 = np.array(mazebool1)

    mazebool2 = [[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 0., 1., 0., 1., 1.],
      [1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0.],
      [0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 0., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [0., 1., 1., 0., 1., 0., 1., 0., 1., 0., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 0., 0., 1., 0., 1., 0., 1., 0., 1., 0., 1.]]

    mazebool2 = np.array(mazebool2)

    mazebool3 = [[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 0., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1.],
      [0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [0., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
      [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]]
   mazebool3 = np.array(mazebool3)

   if n==1:        return
mazebool1     if n==2:
return mazebool2     if
n==3:
       return mazebool3



# In[9]:



starti= 0 startj
= 0
stack     =     []     stack.append(maze[0][0])
maze[0][0].visited = True while(len(stack)):
starti = stack[-1].i        startj = stack[-1].j
maze[starti][startj].visited     =     True     #
print("current ",starti,startj)     starti,startj =
helper(starti, startj)  #        print("selected
neighbour ",starti,startj)     if starti==-1 and
startj==-1: #      print('pop')      stack.pop()
else:
       stack.append(maze[starti][startj])




# In[12]:



k = 1 for i in range(rows):
for j in range(cols):
arr = maze[i][j].walls
if arr[0]==False: #top
mazebool[i-1][j] = k

       elif arr[1]==False:#bottom
           mazebool[i+1][j] = k
```

```
        elif arr[2]==False:#left
           mazebool[i][j-1] = k

        elif arr[3]==False:#right
           mazebool[i][j+1] = k




# In[13]:


viz(mazebool,(rows-2,0),(0,cols-1),16)


# DFS

# In[14]:


starti = 2
startj = 2 endi
= rows-2
endj = cols-2


# # SEARCH PROBLEMS

# In[15]:
```

#1. Code For DFS

```
dfs_visited_count = []


# In[16]:


def dfs_path(matrix, start, end):
   directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]  #up, down, left, right
stack = [(start, [start])]     visited = set()     visited_count = 0
```

```
while stack:        node, path = stack.pop()        if node == end:
dfs_visited_count.append(visited_count)
            return path
        visited.add(node)
visited_count += 1        for
direction in directions:
            neighbor = (node[0] + direction[0], node[1] + direction[1])
            if (0 <= neighbor[0] < len(matrix) and 0 <= neighbor[1] < len(matrix[0]) and
matrix[neighbor[0]][neighbor[1]] == 1):                if neighbor not in visited:
                    stack.append((neighbor, path + [neighbor]))
return None


# In[28]:


mazebool1 = reset(1)


# In[29]:


st = time.time()
path = dfs_path(mazebool1,(mazebool1.shape[0]-2,0),(0,mazebool1.shape[1]-1)) et
= time.time()
print(et-st)


# In[30]:


dfs_visited_count


# In[31]:


print(mazebool1.shape[0]-2,0) print(0,mazebool1.shape[1]-1)


# In[32]:


viz(mazebool1,(mazebool1.shape[0]-2,0),(0,mazebool1.shape[1]-1),16,path)
```

# In[33]:

#2. Code for BFS

```
bfs_visited_cells = []
```

# In[34]:

```
from collections import deque

def bfs_path(matrix, start, end):
    directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]     queue = 
deque([(start, [start])])     visited = set([start])     search_space = 
1     while queue:         search_space = search_space + 1         
node, path = queue.popleft()         if node == end:             
bfs_visited_cells.append(search_space)             return path         
for direction in directions:             neighbor = (node[0] + 
direction[0], node[1] + direction[1])
            if (0 <= neighbor[0] < len(matrix) and 0 <= neighbor[1] < len(matrix[0]) and 
matrix[neighbor[0]][neighbor[1]] == 1):                 if neighbor not in visited:                     
queue.append((neighbor, path + [neighbor]))                     visited.add(neighbor)                     
search_space = search_space + 1     return None
```

# In[38]:

```
mazebool1 = reset(1)
```

# In[39]:

```
st = time.time()
path = bfs_path(mazebool1,(mazebool1.shape[0]-2,0),(0,mazebool1.shape[1]-1)) et 
= time.time()
print(et-st)
```

```
# In[40]:

print(mazebool1.shape[0]-2,0) print(0,mazebool1.shape[1]-1)


# In[41]:


bfs_visited_cells


# In[42]:


viz(mazebool1,(mazebool1.shape[0]-2,0),(0,mazebool1.shape[1]-1),16,path)


# In[43]:
```

#3. Code for A*

```
import heapq

def a_star_path(matrix, start, goal):
   directions = [(0, -1), (0, 1), (-1, 0), (1, 0)] #up, down, left, right     def
heuristic(node):        return abs(node[0] - goal[0]) + abs(node[1] - goal[1])
#manhattan distance     queue = [(heuristic(start), start, [start])]     visited =
set([start])     while queue:
       f_cost, node, path = heapq.heappop(queue)        if node ==
goal:            return path, len(visited)        for direction in
directions:            neighbor = (node[0] + direction[0], node[1] +
direction[1])
           if (0 <= neighbor[0] < len(matrix) and 0 <= neighbor[1] < len(matrix[0]) and
matrix[neighbor[0]][neighbor[1]] == 1):               if neighbor not in visited:
g_cost = len(path) + 1                   h_cost = heuristic(neighbor)
                 f_cost = g_cost + h_cost

                 heapq.heappush(queue, (f_cost, neighbor, path + [neighbor]))
visited.add(neighbor)

   return None, len(visited)
```

```
# In[45]:


mazebool1 = reset(1)


# In[46]:


st = time.time()
path,visited_cells = a_star_path(mazebool1,(mazebool1.shape[0]-
2,0),(0,mazebool1.shape[1]-1))
et = time.time() print(et-st)


# In[47]:


visited_cells


# In[48]:


print(mazebool1.shape[0]-2,0) print(0,mazebool1.shape[1]-1)


# In[49]:


viz(mazebool1,(mazebool1.shape[0]-2,0),(0,mazebool1.shape[1]-1),16,path)


# # MDP PROBLEMS (NON DETERMINISTICS AND CONSISTING PROBABILITY)

# In[50]:
```

#4. Code for Value Iteration

```
def getU(mazee, r, c, action):
   actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]     dr, dc = actions[action]     newr, newc = r+dr,
c+dc #     print(newr,newc)     if newr < 0 or newc < 0 or newr >= rows or newc >= cols or
(mazee[newr][newc] == 9):         return mazee[r][c]     else:
       return mazee[newr][newc]


# In[51]:


def statemaker(mazebool,rows,cols):
   k = np.zeros((rows, cols))
for i in range(rows):
for j in range(cols):
if mazebool[i][j]==0:
k[i][j] = 9     return k


# In[52]:


def getOptimalPolicy(U):     policy = [[-1]*cols
for i in range(rows)]     for r in range(rows):
for c in range(cols):            if (r <= 1 and c ==
cols-1) or (U[r][c]==9):               continue
           maxAction, maxU = None, -float("inf")
for action in range(4):
              u = reward
              u += 0.1 * discount * getU(U, r, c, (action-1)%4)
u += 0.8 * discount * getU(U, r, c, action)               u +=
0.1 * discount * getU(U, r, c, (action+1)%4)               if u
> maxU:
                 maxAction, maxU = action, u
policy[r][c] = maxAction     return policy


# In[53]:


def printEnvironment(arr, policy=False):
   res = ""     for r in
range(rows):
       res += "|"        for c
in range(cols):           if
```

```
arr[r][c]==-1 and ((r!=0
or r!=1) and c!= cols-1):
                val = "WALL"            elif r
<= 1 and c == cols-1 :                val =
"+1" if r == 0 else "-1"            else:
if policy:
                    val = ["Up", "Down", "Left", "Right"][arr[r][c]]
else:
                    val = str(arr[r][c])
            res += " " + val[:5].ljust(5) + " |" # format
res += "\n"
    print(res)


# In[54]:


def viz2(arr,fs):
    arr = np.array(arr)
n_rows = arr.shape[0]
n_cols = arr.shape[1]
    matrix = np.copy(arr)

    fig, ax = plt.subplots()
    fig.set_size_inches(n_cols, n_rows)


    for i in range(n_rows):
for j in range(n_cols):

            if arr[i][j]==9:                ax.text(j, i, 'Wall', ha='center', va='center',
color='gold',fontsize = fs)            elif arr[i][j]==1:                ax.text(j, i, 1,
ha='center', va='center', color='green',fontsize = fs)            elif arr[i][j]==-1:
                ax.text(j, i, -1, ha='center', va='center', color='red',fontsize = fs)

else:
                ax.text(j, i, round(arr[i][j],2), ha='center', va='center', color='black',fontsize = fs)


    cmap = matplotlib.colors.ListedColormap(['white', 'black'])
    ax.imshow(matrix, cmap=cmap, origin='upper')     ax.set_xticks(np.arange(-
0.5, n_cols, 1), minor=True)     ax.set_yticks(np.arange(-0.5, n_rows, 1),
minor=True)     ax.grid(which='minor', color='gray', linestyle='-', linewidth=1)
ax.set_yticks(np.arange(-0.5, n_rows, 1))     ax.grid(which='major', axis='y',
color='grey', linestyle='-', linewidth=1)     ax.tick_params(axis='both',
which='both', length=0)     ax.set_xticklabels([])     ax.set_yticklabels([])

   plt.show()


# In[55]:


def viz3(policy,arr,fs):
arr = np.array(arr)
n_rows = arr.shape[0]
n_cols = arr.shape[1]
   matrix = np.copy(arr)

   fig, ax = plt.subplots()
   fig.set_size_inches(n_cols, n_rows)


   for i in range(n_rows):
for j in range(n_cols):

         if arr[i][j]==9:
            ax.text(j, i, 'Wall', ha='center', va='center', color='gold',fontsize = fs)
                              elif arr[i][j]==1:                  ax.text(j, i, 1,
ha='center', va='center', color='green',fontsize = fs)            elif arr[i][j]==-
1:
            ax.text(j, i, -1, ha='center', va='center', color='red',fontsize = fs)
elif policy[i][j]==0:

            ax.text(j, i, 'UP', ha='center', va='center', color='teal',fontsize = fs)
elif policy[i][j]==1:                ax.text(j, i, 'DOWN', ha='center', va='center',
color='crimson',fontsize = fs)            elif policy[i][j]==2:
            ax.text(j, i, 'LEFT', ha='center', va='center', color='magenta',fontsize = fs)
elif policy[i][j]==3:
            ax.text(j, i, 'RIGHT', ha='center', va='center', color='blue',fontsize = fs)




   cmap = matplotlib.colors.ListedColormap(['white', 'black'])
ax.imshow(matrix, cmap=cmap, origin='upper')     ax.set_xticks(np.arange(-0.5,
n_cols, 1), minor=True)     ax.set_yticks(np.arange(-0.5, n_rows, 1),
minor=True)     ax.grid(which='minor', color='gray', linestyle='-', linewidth=1)
ax.set_yticks(np.arange(-0.5, n_rows, 1))
   ax.grid(which='major', axis='y', color='grey', linestyle='-', linewidth=1)
ax.tick_params(axis='both', which='both', length=0)
ax.set_xticklabels([])     ax.set_yticklabels([])

   plt.show()




# In[56]:



def value_iteration(maze, reward, discount, epsilon, rows, cols, loosei, loosej, wini, winj):
number_of_actions = 4 #up, down, left, right
   maze[loosei][loosej] = -1     maze[wini][winj] = 1
start_state = np.copy(maze)     iterations = 0
while(True):        next_state = np.copy(start_state)
error = 0        for i in range(rows):            for j in
range(cols):                if (i <= 1 and j == cols-1) or
(maze[i][j] == 9):
               continue                all_states = []
for action in range(number_of_actions):
u = reward
                  u += 0.1 * discount * getU(maze, i, j, (action-1)%4)
u += 0.8 * discount * getU(maze, i, j, action)                      u +=
0.1 * discount * getU(maze, i, j, (action+1)%4)
all_states.append(u)

            next_state[i][j] = max(all_states)

            error = max(error, abs(next_state[i][j]-maze[i][j]))
iterations = iterations + 1        maze = next_state        if
error < epsilon * (1-discount)/discount:            break
print(iterations)
   return maze



# In[58]:



reward = -0.2 discount =
0.99 epsilon = 0.0001
mazebool1 = reset(1) rows
```

```
= mazebool1.shape[0] cols
= mazebool1.shape[1] st =
time.time()
U1 =
value_iteration(statemaker(mazebool1,rows,cols),reward,discount,epsilon,rows,cols,1,cols-
1,0,cols-1) et =
time.time()
print(et-st) #
viz2(U1,12)
U2 = getOptimalPolicy(U1)
# print(U2) viz3(U2,U1,12)
# printEnvironment(U2, policy=True)


# In[ ]:


reward = -0.1 discount =
0.99 epsilon = 0.0001
mazebool1 = reset(2) rows
= mazebool1.shape[0] cols
= mazebool1.shape[1] U =
value_iteration(statemaker(mazebool1,rows,cols),reward,discount,epsilon,rows,cols,1,cols
1,0,cols-1)
U = getOptimalPolicy(U) printEnvironment(U,
policy=True)


# In[ ]:


reward = -0.1 discount =
0.99 epsilon = 0.0001
mazebool1 = reset(3) rows
= mazebool1.shape[0] cols
= mazebool1.shape[1] U =
value_iteration(statemaker(mazebool1,rows,cols),reward,discount,epsilon,rows,cols,1,cols
1,0,cols-1)
U = getOptimalPolicy(U) printEnvironment(U,
policy=True)


# In[59]:
```

```
def calculateU(U, r, c, action):
u = reward
   u += 0.1 * discount * getU(U, r, c, (action-1)%4)
u += 0.8 * discount * getU(U, r, c, action)     u +=
0.1 * discount * getU(U, r, c, (action+1)%4)
return u


# In[60]:
```

#5. Code for policy Iteration

```
def policy_iteration(policy, maze, reward, discount, epsilon, rows, cols, loosei, loosej, wini,
winj):
   number_of_actions = 4 #up, down, left, right
   maze[loosei][loosej] = -1
maze[wini][winj] = 1     start_state
= np.copy(maze)     iterations = 0
   print("During the policy iteration:\n")
interations = 0     while(True):         next_state =
np.copy(start_state)         error = 0         for i in
range(rows):             for j in range(cols):                 if (i
<= 1 and j == cols-1) or (maze[i][j] == 9):
               continue
            next_state[i][j] = calculateU(maze, i, j, policy[i][j])
error = max(error, abs(next_state[i][j]-maze[i][j]))

      maze = next_state

      if error < epsilon * (1-discount)/discount:
break

      unchanged = True
for i in range(rows):
for j in range(cols):
if (i <= 1 and j == cols-1)
or (maze[i][j] == 9):
               continue
            maxAction, maxU = None, -float("inf")
for action in range(4):                   u =
calculateU(maze, i, j, action)                   if u >
maxU:
```

```
                        maxAction, maxU = action, u
if maxU > calculateU(maze, i, j, policy[i][j]):
policy[i][j] = maxAction                    unchanged =
False        if unchanged:
            break
        iterations = iterations + 1 #
printEnvironment(policy)
print('iterations ',iterations)
return policy


# In[61]:



##


reward = 0 discount = 0.99
epsilon = 0.0001
mazebool1 = reset(1) rows
= mazebool1.shape[0]
cols = mazebool1.shape[1]

policy = [[random.randint(0, 3) for j in range(cols)] for i in range(rows)]

# print("The initial random policy is:\n")
# printEnvironment(policy,policy = True)

st = time.time()

U = policy_iteration(policy,
statemaker(mazebool1,rows,cols),reward,discount,epsilon,rows,cols,1,cols-1,0,cols-1) et
= time.time()

print(et-st)
# print(U)
# arr = np.copy(mazebool1)
# arr[1][cols-1] = -1
# arr[0][cols-1]
# viz3(U,statemaker(arr,rows,cols),12)

# print(U)
mazebool1 = statemaker(mazebool1,rows,cols) mazebool1[1][cols-1]
= -1
mazebool1[0][cols-1] = 1
```

```
viz3(U,mazebool1,12)
# print(mazebool1)

# print(U)
# print(statemaker(mazebool1,rows,cols))
# print(arr)
# print("The optimal policy is:\n")
# printEnvironment(U,policy = True)


# In[ ]:




# In[ ]:
```