Names
Bishoy Nader (21)
Marc Magdi (55)
Moustafa Mahmoud (76)

Reinforcement Learning

27th January 2019

OVERVIEW

For this assignment you will be making a maze solver. Your program will generate a maze of size NxN. Also you should generate barriers at random grid locations. Then you will try to learn the path out of the grid using policy and value iteration.

ASSUMPTIONS

1. Terminal cell in the maze is the cell with index[N-1][N-1].
2. Barriers are generated randomly on cells in the maze, the terminal cell cannot have a barrier.
3. The inputs to the program are:
    a. The Maze Dimension [N}
    b. The Number of Barriers to be generated.
    c. The value of Gamma.
4. After Entering Inputs the program will solve the maze using both Policy Evaluation and Value Iteration to get optimal Policy in each of them.
5. In case of ties between actions, the program only selects one of them so that the policy is deterministic.

CODE ORGANIZATION

We divided the code into three packages:

1. Models: contains the model classes of the maze.
2. Solvers: contains the logic for both policy iteration and value iteration solvers.
3. Utils: contains the class responsible for printing values and policy in each iteration.

Model Classes

1. Maze: Contains reference to an NxN array of cells,
2. Cell: Contains a flag indicating whether this cell is a barrier or not, 2 values representing the value at a specific state and a backup from the previous state, array of probabilities representing the probability of each action [left, right, up, down].
3. Pair: Inner class used in generating barriers at random indices.

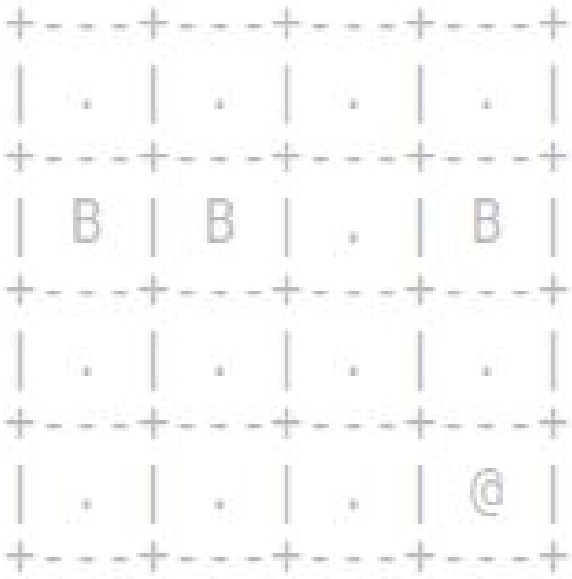
Important Methods

1. List<Pair> getBarrierIndices(int dimension, int numberOfBarriers): generates the random indices for the barriers in the maze,
2. void solve(): solves the maze and is implemented by PolicyIterationSolver and ValueIterationSolver.
3. void initializeRandomPI(): initializes the actions randomly in policy iteration.
4. double performPolicyEvaluation(): evaluates the policy for one iteration.
5. double calculateNewValue(int row, int column): calculates the new value of a specific cell.
6. int getImmediateReward(int row, int column): gets the immediate reward for moving to a specific cell.
7. boolean isValid(int row, int column): check where these indices are within the bounds of the maze.
8. void updateValues(): updates old values of each cell with the new calculated values.
9. double greedyPolicyImprovement(): improves the policy greedly and re-evaluate it.
10. double[] getActionValues(int row, int column): get the gain for each action.
11. int getNewActionGreedy(double[] actionValues): choose the best action from its gain.
12. oid performIteration(): performs one value iteration to the maze.
13. boolean updateValues():: update old values with new values and check whether the policy converged or not in value iteration.

How Algorithms Work

For simplicity: To show how algorithms work we will print the values of each cell and policy in each iteration on a small 4x4 Maze with only 3 barriers and gamma = 1.

Initial Maze Shape



Policy Iteration

1. i=0; Initialize $\pi_0(s)$ randomly for all states s
2. While i == 0 or $|\pi_i - \pi_{i-1}| > 0$ ←
   - Policy evaluation: Compute value of π_i
   - i=i+1
   - Policy improvement:

$$Q^{\pi_i}(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^{\pi_i}(s')$$

$$\pi_{i+1}(s) = \arg\max_a Q^{\pi_i}(s,a)$$

Use a L1 norm: measures if the policy changed for any state


```
+---+---+---+---+
| ↓ | ↓ | ↑ | → |
+---+---+---+---+
| # | # | → | # |
+---+---+---+---+
| ← | ↑ | ↓ | ↓ |
+---+---+---+---+
| ↓ | → | → | @ |
+---+---+---+---+
```

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	84709.39572712817	99984.0
0.0	84709.39572712817	99984.0	0.0

↓	↓	↑	→
#	#	↓	#
←	→	→	↓
→	→	→	@

0.0	0.0	0.0	0.0
0.0	0.0	84708.39572712817	0.0
0.0	84708.39572712817	99983.0	99984.0
84708.39572712817	99983.0	99984.0	0.0

↓	↓	↓	→
#	#	↓	#
→	→	→	↓
→	→	→	@

0.0	0.0	84707.39572712817	0.0
0.0	0.0	99982.0	0.0
84707.39572712817	99982.0	99983.0	99984.0
99982.0	99983.0	99984.0	0.0

```
+---+---+---+---+
| ↓ | → | ↓ | ← |
+---+---+---+---+
| # | # | ↓ | # |
+---+---+---+---+
| → | → | → | ↓ |
+---+---+---+---+
| → | → | → | @ |
+---+---+---+---+
```

0.0	84706.39572712817	99981.0	84706.39572712817
0.0	0.0	99982.0	0.0
99981.0	99982.0	99983.0	99984.0
99982.0	99983.0	99984.0	0.0

```
+---+---+---+---+
| → | → | ↓ | ← |
+---+---+---+---+
| # | # | ↓ | # |
+---+---+---+---+
| → | → | → | ↓ |
+---+---+---+---+
| → | → | → | @ |
+---+---+---+---+
```

84705.39572712817	99980.0	99981.0	99980.0
0.0	0.0	99982.0	0.0
99981.0	99982.0	99983.0	99984.0
99982.0	99983.0	99984.0	0.0

```
+---+---+---+---+
| → | → | ↓ | ← |
+---+---+---+---+
| # | # | ↓ | # |
+---+---+---+---+
| → | → | → | ↓ |
+---+---+---+---+
| → | → | → | @ |
+---+---+---+---+
```

99979.0	99980.0	99981.0	99980.0
0.0	0.0	99982.0	0.0
99981.0	99982.0	99983.0	99984.0
99982.0	99983.0	99984.0	0.0

Value Iteration

1. Initialize $V_0(s)=0$ for all states s
2. Set k=1
3. Loop until [finite horizon, convergence]
   - For each state s

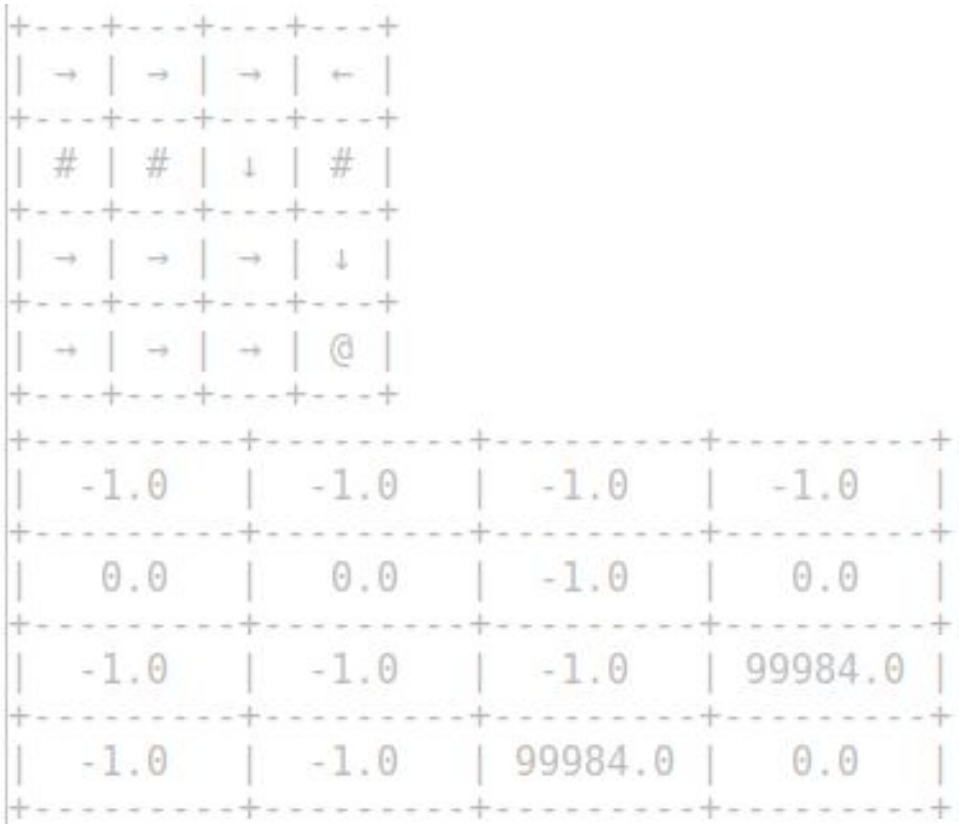
$$V_{k+1}(s) = \max_a R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s')$$

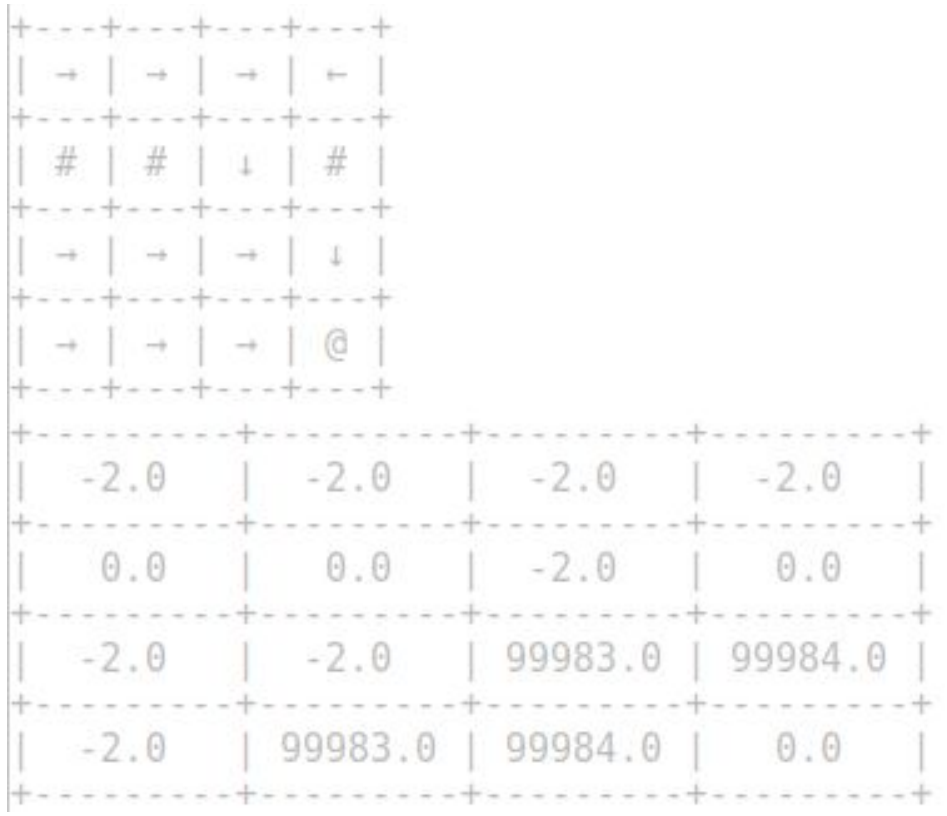
   - View as Bellman backup on value function

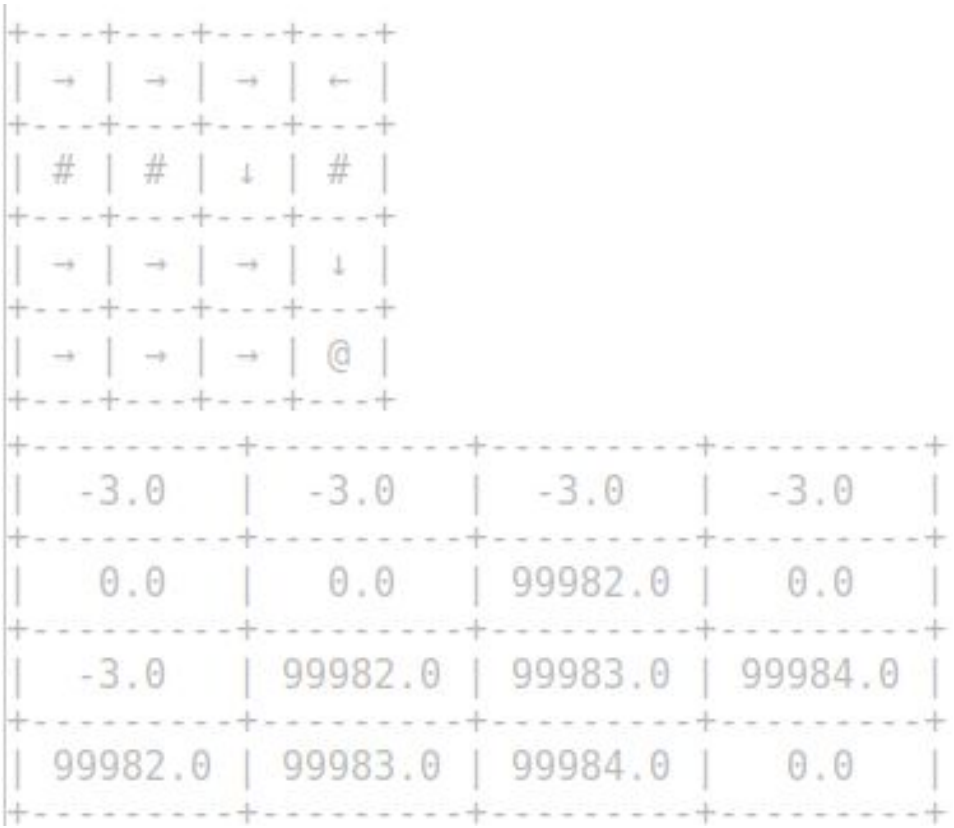
$$V_{k+1} = BV_k$$

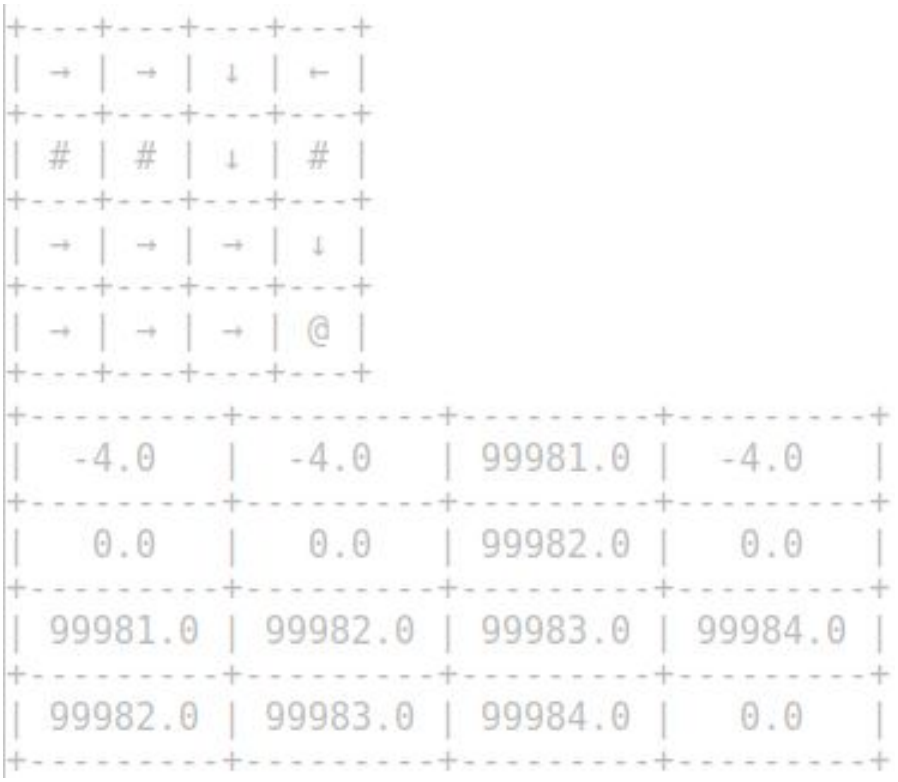
$$\pi_{k+1}(s) = \arg\max_a R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s')$$

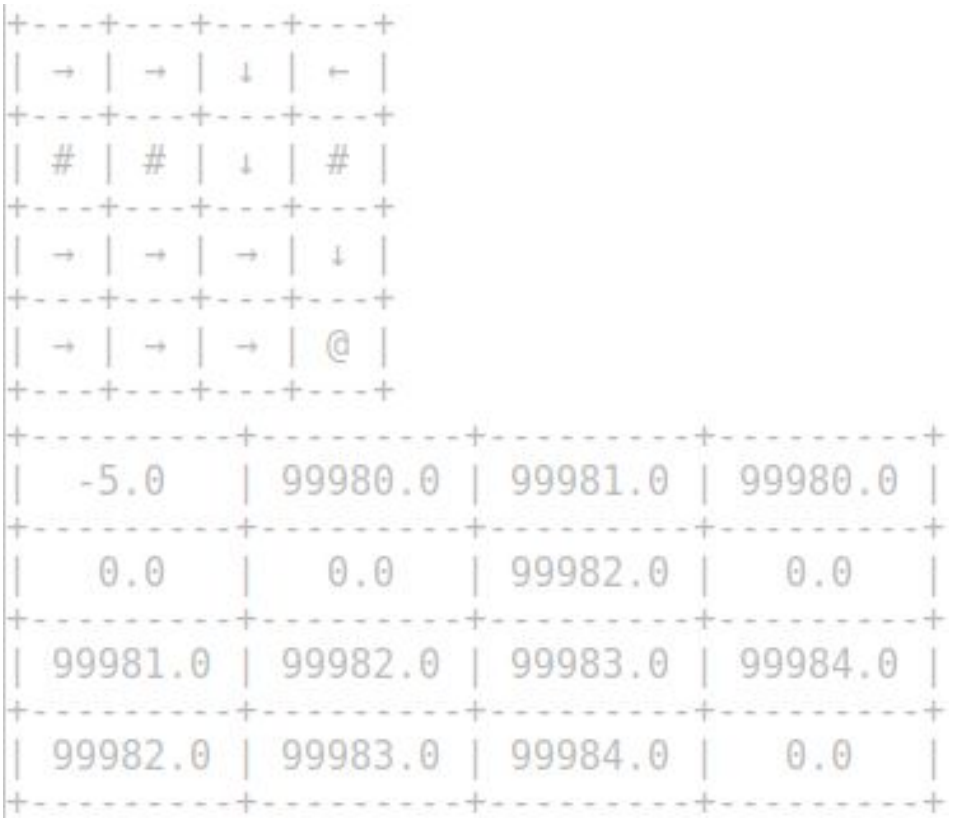

→ → → →
→
→ → → →
→ → → @
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0


→ → → ←
↓
→ → → ↓
→ → → @
-1.0 -1.0 -1.0 -1.0
0.0 0.0 -1.0 0.0
-1.0 -1.0 -1.0 99984.0
-1.0 -1.0 99984.0 0.0


→ → → ←
↓
→ → → ↓
→ → → @
-2.0 -2.0 -2.0 -2.0
0.0 0.0 -2.0 0.0
-2.0 -2.0 99983.0 99984.0
-2.0 99983.0 99984.0 0.0


→ → → ←
↓
→ → → ↓
→ → → @
-3.0 -3.0 -3.0 -3.0
0.0 0.0 99982.0 0.0
-3.0 99982.0 99983.0 99984.0
99982.0 99983.0 99984.0 0.0

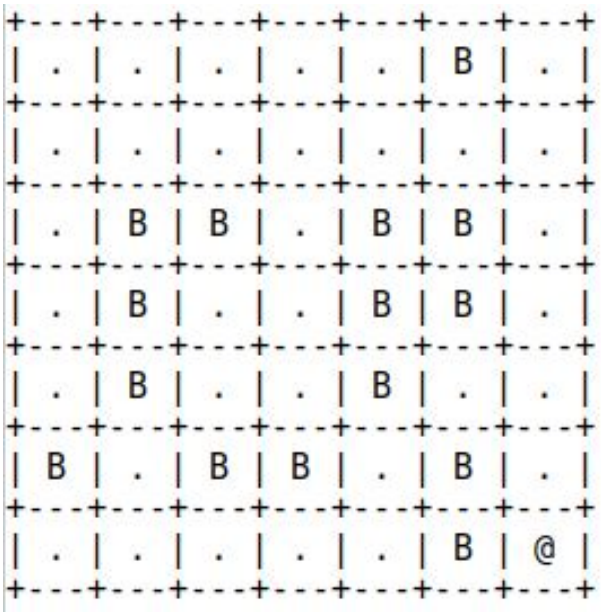

→ → ↓ ←
↓
→ → → ↓
→ → → @
-4.0 -4.0 99981.0 -4.0
0.0 0.0 99982.0 0.0
99981.0 99982.0 99983.0 99984.0
99982.0 99983.0 99984.0 0.0


→ → ↓ ←
↓
→ → → ↓
→ → → @
-5.0 99980.0 99981.0 99980.0
0.0 0.0 99982.0 0.0
99981.0 99982.0 99983.0 99984.0
99982.0 99983.0 99984.0 0.0

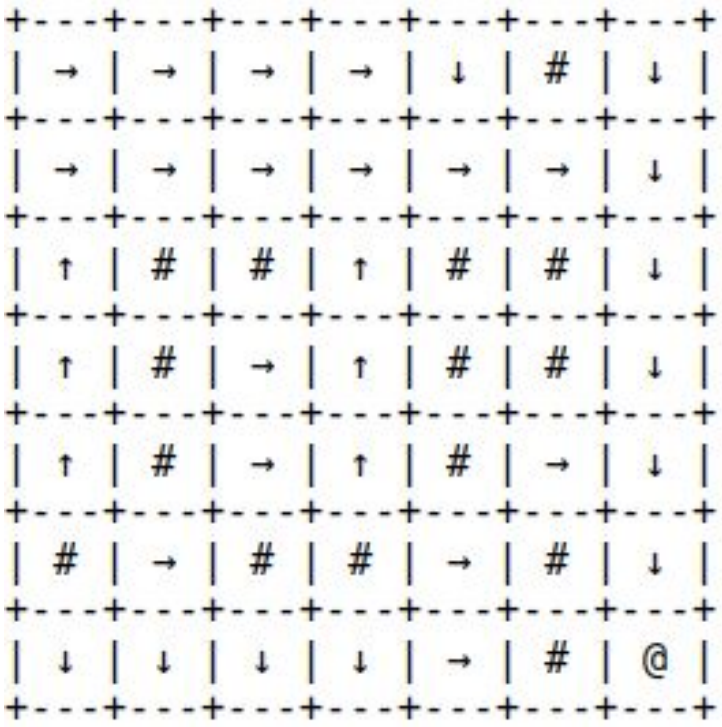
Both of them converge to the optimal policy

Sample Runs [Full Iteration Tracing is available in the attached files]:

Run 1: N = 7, Barriers = 15, Gamma = 0.5



Policy Iteration converges to the following policy:



With the following final values

46.80517578125	95.6103515625	193.220703125	388.44140625	778.8828125	0.0	3121.53125
95.6103515625	193.220703125	388.44140625	778.8828125	1559.765625	3121.53125	6245.0625
46.80517578125	0.0	0.0	388.44140625	0.0	0.0	12492.125
22.402587890625	0.0	95.6103515625	193.220703125	0.0	0.0	24986.25
10.2012939453125	0.0	46.80517578125	95.6103515625	0.0	24986.25	49974.5
0.0	0.0	0.0	0.0	0.0	0.0	99951.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0

And the following Statistics

```
Time Taken To Solve in ms: 60
Number of Iterations to Solve: 15
Path Cost from Each Cell:
+------+------+------+------+------+------+------+
| 12.0 | 11.0 | 10.0 | 9.0  | 8.0  |  B   | 6.0  |
+------+------+------+------+------+------+------+
| 11.0 | 10.0 | 9.0  | 8.0  | 7.0  | 6.0  | 5.0  |
+------+------+------+------+------+------+------+
| 12.0 |  B   |  B   | 9.0  |  B   |  B   | 4.0  |
+------+------+------+------+------+------+------+
| 13.0 |  B   | 11.0 | 10.0 |  B   |  B   | 3.0  |
+------+------+------+------+------+------+------+
| 14.0 |  B   | 12.0 | 11.0 |  B   | 3.0  | 2.0  |
+------+------+------+------+------+------+------+
|  B   |  NA  |  B   |  B   |  NA  |  B   | 1.0  |
+------+------+------+------+------+------+------+
|  NA  |  NA  |  NA  |  NA  |  NA  |  B   | 0.0  |
+------+------+------+------+------+------+------+
```

Value Iteration converges to the following policy:

```
+---+---+---+---+---+---+---+
| → | → | → | → | ↓ | # | ↓ |
+---+---+---+---+---+---+---+
| → | → | → | → | → | → | ↓ |
+---+---+---+---+---+---+---+
| ↑ | # | # | ↑ | # | # | ↓ |
+---+---+---+---+---+---+---+
| ↑ | # | → | ↑ | # | # | ↓ |
+---+---+---+---+---+---+---+
| ↑ | # | → | ↑ | # | → | ↓ |
+---+---+---+---+---+---+---+
| # | ↓ | # | # | ↓ | # | ↓ |
+---+---+---+---+---+---+---+
| → | → | → | → | ← | # | @ |
+---+---+---+---+---+---+---+
```

With the following final values

46.80517578125	95.6103515625	193.220703125	388.44140625	778.8828125	0.0	3121.53125
95.6103515625	193.220703125	388.44140625	778.8828125	1559.765625	3121.53125	6245.0625
46.80517578125	0.0	0.0	388.44140625	0.0	0.0	12492.125
22.402587890625	0.0	95.6103515625	193.220703125	0.0	0.0	24986.25
10.2012939453125	0.0	46.80517578125	95.6103515625	0.0	24986.25	49974.5
0.0	-2.0	0.0	0.0	-2.0	0.0	99951.0
-2.0	-2.0	-2.0	-2.0	-2.0	0.0	0.0

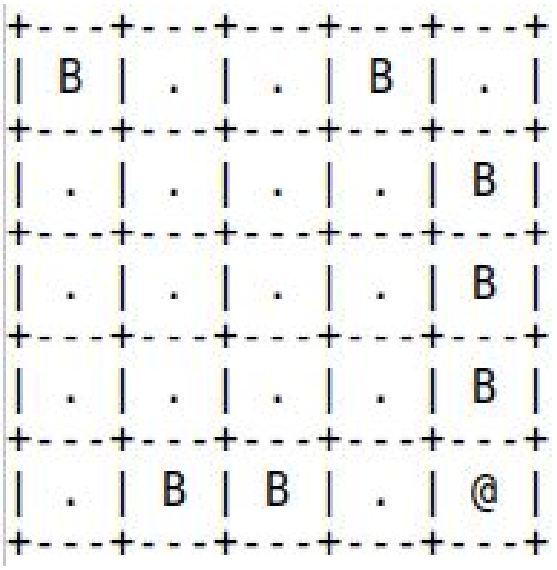
And the following Statistics

Time Taken To Solve in ms: 110
Number of Iterations to Solve: 54
Path Cost from Each Cell:

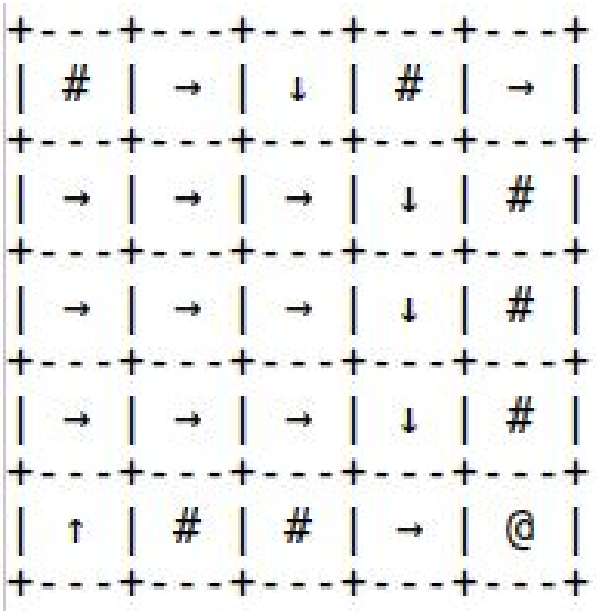
12.0	11.0	10.0	9.0	8.0	B	6.0
11.0	10.0	9.0	8.0	7.0	6.0	5.0
12.0	B	B	9.0	B	B	4.0
13.0	B	11.0	10.0	B	B	3.0
14.0	B	12.0	11.0	B	3.0	2.0
B	NA	B	B	NA	B	1.0
NA	NA	NA	NA	NA	B	0.0

Run 2: N = 5, Barriers = 7, Gamma = 0.7

Initial Maze



Policy Iteration converges to the following policy:



With the following final values

0.0	11759.016403247068	16800.023719312794	0.0	0.0
11759.016403247068	16800.023719312794	24001.46286490361	34289.233248090175	0.0
16800.023719312794	24001.46286490361	34289.233248090175	48986.048331499114	0.0
24001.46286490361	34289.233248090175	48986.048331499114	69981.49880820513	0.0
16800.023719312794	0.0	0.0	99975.0	0.0

And the following Statistics

Time Taken To Solve in ms: 31
Number of Iterations to Solve: 8
Path Cost from Each Cell:

B	7.0	6.0	B	NA
7.0	6.0	5.0	4.0	B
6.0	5.0	4.0	3.0	B
5.0	4.0	3.0	2.0	B
6.0	B	B	1.0	0.0

Value Iteration converges to the following policy:

With the following final values

0.0	11759.016403247068	16800.023719312794	0.0	-1.7976931348623157E308
11759.016403247068	16800.023719312794	24001.46286490361	34289.233248090175	0.0
16800.023719312794	24001.46286490361	34289.233248090175	48986.048331499114	0.0
24001.46286490361	34289.233248090175	48986.048331499114	69981.49880820513	0.0
16800.023719312794	0.0	0.0	99975.0	0.0

And the following Statistics