Assignment 4: Markov Decision Processes

Vinodh Kumar Sunkara, Microsoft Redmond (vsunkara6)

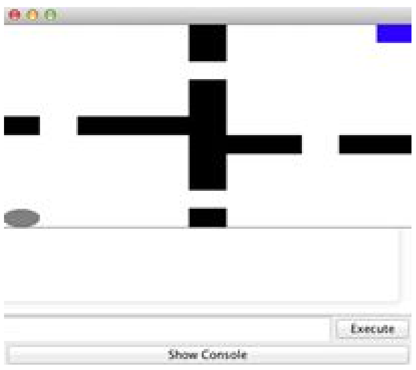
MDP (Markov Decision Process) provides a formal description of a real world where the agent makes decisions by interacting with the world in a sequence of steps to reach the final destination. Here we apply planning and learning algorithm to couple of MDP Problems.
The First problem is a GridWorld with large no of states and solved using BURLAP API.
The second one is a Maze with smaller number of states and solved using ABAGAIL API.

The goal of planning or learning is to find a *policy*, which is a mapping from states in the MDP to actions that the agent takes.

Apply MDP to reach Goal in a GridWorld with large number of states

GridWorld is a very interesting case study concept and also part of many analytical puzzles. Many real time problems are easily solved by putting in grid world. Gaming industry has also evolved in terms of grid games, especially the snake games. Let's explore this as our first MDP problem. Sequential problems are intriguing due to the components that contribute to the makeup of the problem and how each of the components affects the solution. The infrastructure that we will focus on is the Markovian transition model and additive rewards which is called a Markov decision process (MDP). The model is of a grid world, an m-by-n world that contains obstacles, terminals, a starting initial state, and where each state in the world has a reward value that can be a negative or positive value. In the grid world, there is also a probabilistic move formulation that specifies the probability of moving in each of the four directions (North, South, East, and West) given that the agent has chose to go towards a particular direction. Given a grid world with obstacles, terminals, a start state, rewards, and moves, we will find the optimal policy of the grid world, the policy that yields the highest expected utility. The result is a world where each state has a direction and each direction points to the best direction to take when the agent is at that state; the best direction being the direction that will yield the highest expected utility. To solve for the optimal policy, we use two different algorithms: value iteration and policy iteration. Each of the algorithms will produce an optimal policy. We will be observing the results of these two algorithms and comparing them with each other.

The GridWorld problem simulates a world where the agent can travel north, south, east or west.



To understand the basic MDP problem we can first run
java -cp mdp_tests.jar:lib/burlap.jar com.mdp.tests.SimpleGridExplorer

Pressing either the n , s , e or w keys will help the agent move in the Grid towards the target Goal . The agent can take the actions -> north , south , east , west !

'Transition Dynamics' is one of the most important features of MDP which defines the probability of the world changing to states' in the next discrete time step when the agent takes action *a* in the current state *s*

The fact that the world can change stochastically is one of the unique properties of an MDP compared to more classic planning/decision making problems

With high probability (0.8) the agent will move in the intended direction, and with some low probability (0.2) move in a different direction.

In our code, we set the probability to 1 to avoid stochastic transitions which sometimes may lead to the wrong direction.

The domain consists of an agent and grid locations. Here the actions are unconditional.

A terminal state is a state that once reached causes all further action of the agent to cease. This is a useful concept for defining goal-directed tasks (i.e., action stops once the agent achieves a goal) or failure scenarios.

Planning with Value Iteration (VI)

Lets run the program with a VI planner.
java -cp mdp_tests.jar:lib/burlap.jar com.mdp.tests.GridExplorerValueIterator
Here VI computes the policy for the entire state space that is reachable from the initial state.

If we set (discount factor) γ to 0.99 in our grid world, the agent will reach the goal as fast as possible, because waiting longer would result in the eventual +1 goal reward being more heavily discounted.

Unlike the deterministic planners (BFS, DFS, A*) it directly works on reward function and termination function , rather than a Goal condition.
It also considers a discount factor which specifies how much future rewards are favored over immediate rewards.
The Value Iteration algorithm will converge to the optimal Value function if we simply initialize the value for each state to some arbitrary value, and then iteratively use the Bellman equation to update the the value for each state.

In the first run, we use Uniform Reward Function which assigns -1 as reward.

# states: 104 , Passes: 20 , max time steps : 20 ,
Value iteration learned in 132 ms.

If we use a GoalBased Reward Function (reward for terminal function 20)
# states: 104 , Passes: 13 , max time steps : 20
Value iteration learned in 115 ms.

We see that performance has improved by applying reward!

If we set a probability 0.8 for Transition Dynamics, then with GoalBased Reward Function, the latency increases i.e. stochastic transition slows down decision making process.
# states: 104 , Passes: 20 , max time steps : 23
Value iteration learned in 224 ms.

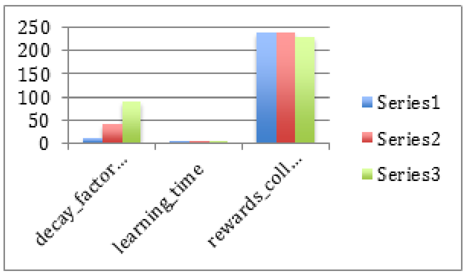
With a deterministic Transition Dynamics, if the Reward for reaching Terminal is increased to 60 from 20,
# states: 104 , Passes: 13 , max time steps : 20
Value iteration learned in 97 ms.

When the maximum change in the value function of any state is less than that specified threshold value (0.001 in this case), planning will stop.

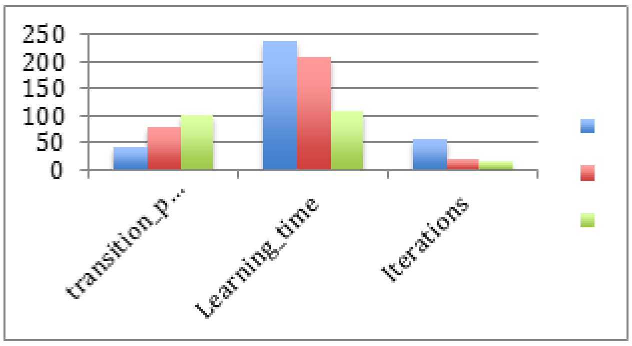
The policy is derived from the Value Function.

Learning time and #iterations decreases as Reward value increases (assuming transition probability=1)

Reward	**Learning_time**	**Iterations**
-1	586	20
5	703	20
45	757	14



Learning time and #iterations decreases as transition_probability increases
(assuming reward for terminal function = 5)



** Increase in discount factor does not affect / improve the overall performance

So we observe 'Deterministic Transition with Higher Goal based Reward' is the most optimal plan in Value Iteration approach.

Planning with Policy Iteration (PI)

Comparison between VI and PI :

In VI :
• Every pass (or "backup") updates both utilities (explicitly, based on current utilities) and policy (possibly implicitly, based on current policy)

In PI :
• Several passes to update utilities with frozen policy
• Occasional passes to update policies

Lets run
java -cp mdp_tests.jar:lib/burlap.jar com.mdp.tests.GridExplorerPolicyIterator

Here we first apply UniformCostFuntion where all transitions return -1 as default reward.

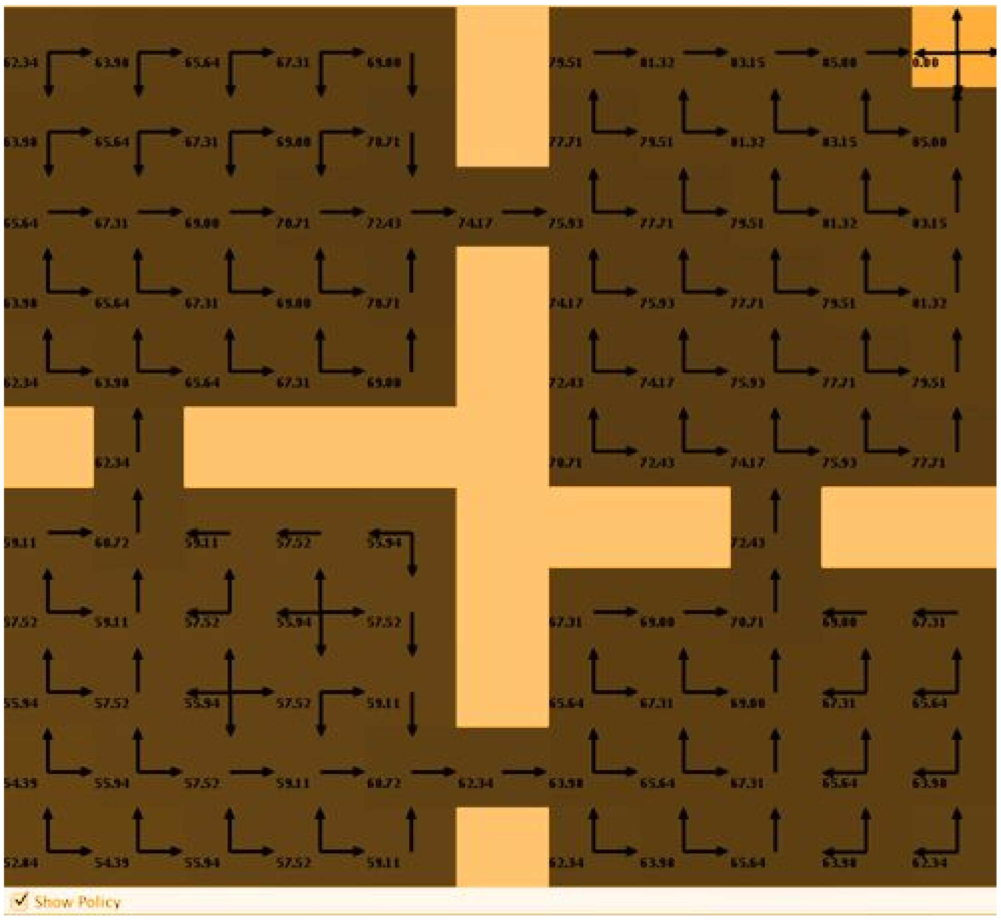
PI picks an initial policy, usually just by taking rewards on states as their utilities and computing a policy according to the maximum expected utility principle.
Then it iteratively performs two steps: value determination, which calculates the utility of each state given the current policy, and policy improvement, which updates the current policy if any improvement is possible.
The algorithm terminates when the policy stabilizes.

When transition dynamics is configured with probability 0.8
Policy iteration learned in 589 ms with max time steps : 28

When transition dynamics is configured with probability 1.0
Policy iteration learned in 578 ms with max time steps : 20

Now with a deterministic Transition Dynamics, if we apply a Goal based Reward Function, we see that Policy Iteration takes longer time (647 ms) to converge.

The main advantage of PI it's guaranteed to converge and it calculates utilities for a fixed policy (not optimal utilities) until convergence.
Let's take a look at the action sequence :



Affect of 'increase in Reward' on "Learning time" :

Reward	Learning_time
-1	596
5	703
45	752
85	715
95	697

For a larger state space solving a system of linear equations by PI is not efficient.

Learning with Reinforcement Learning Algo (RL)

The **difference between a learning algorithm and a planning algorithm** is that

- a **planning algorithm** has access to a model of the world, or at least a simulator,

- whereas a **learning algorithm** involves determining behavior when the agent does not know how the world works and must learn how to behave from direct experience with the world (model-free)

We choose QLearner that runs multiple episodes of learning to solve the problem (or one very long episode if it is a continuing task rather than an episodic task).

Planning will be terminated when max Q-value change within an episode is less than the threshold.

```
java -cp mdp_tests.jar:lib/burlap.jar com.mdp.tests.GridExplorerQLearnerWithoutPlotter
```

If we run the algo with stochastic transient dynamics (p=0.8), then it takes 547 ms and completes in 157 steps.

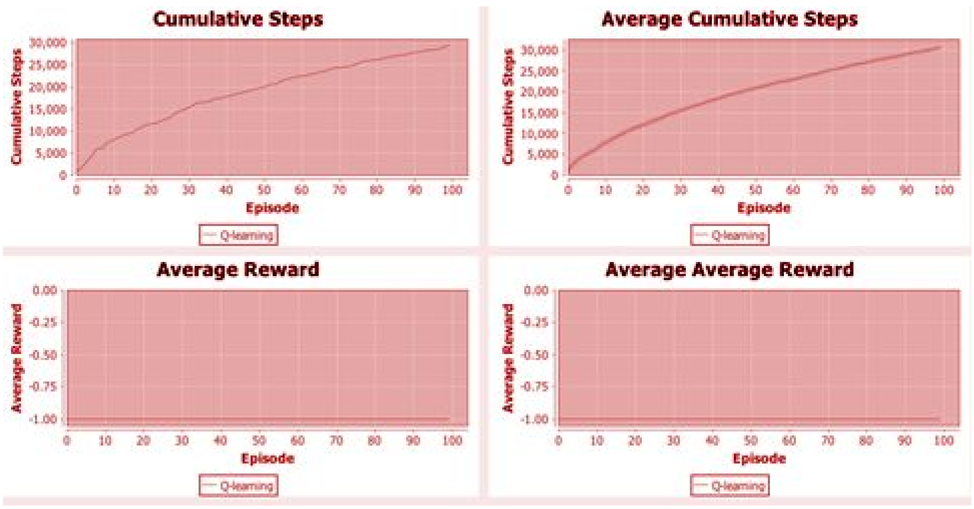
If we run the algo with deterministic transient dynamics, then it takes 513 ms and completes in 20 steps.

If we run the algo with goal-based rewards (rewards 80), then it takes 479 ms and completes in 20 steps.

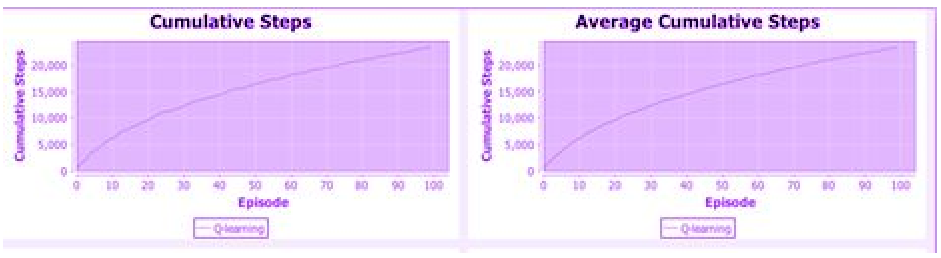
Now we plot the results of QLearning Algo through a separate application where we show the relative improvement of performance with different configurations (probabilistic + UniformReward , deterministic + UniformReward, deterministic + GoalBased Reward)

java -cp mdp_tests.jar:lib/burlap.jar com.mdp.tests.GridExplorerQLearnerWithPlotter

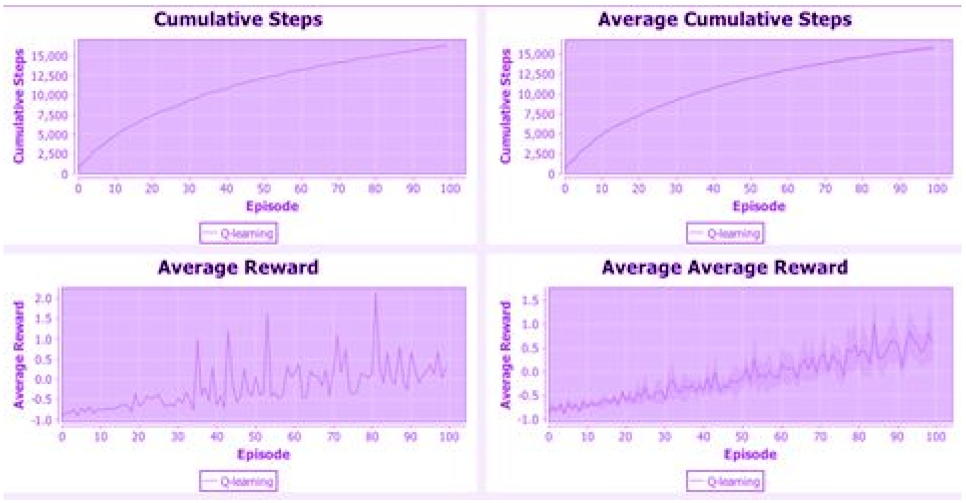
Let’s take a look at the steps per episode with transition dynamics (prob 80%) and with uniform rewards for each step.



Running QLearning with deterministic Transition Dynamics



Now we configure Terminal Function with Goal based Reward 80 instead of Uniform Reward.



So overall QLearning works best with deterministic transitions and goal-based rewards.
When we compare Planning and Learning Algorithm for apparently large no of states , we see Value-based Iteration performs best.

Apply MDP to reach Goal in a Maze with small number of states

Maze is a very common challenge taken up right from nursery stage in schools. Solving mazes is also an integral section in many mental ability tests. Many real life scenarios can also be turned into mazes with start and goal point. It has a good commercial significance and a live maze to enter and find way out, is part of many sight seeing locations in world. Let's pick this as our second MDP problem. First a very small sized Maze is chosen as dataset.

```
#x####
#    #
# ## #
#    #
#o####
```

Goal : x , Agent : o , Obstacle : # , Empty : ' '
Actions => MOVE_UP : 0 , MOVE_DOWN : 1, MOVE_LEFT : 2, MOVE_RIGHT : 3

```
java -cp mdp_tests.jar:lib/ABAGAIL.jar com.mdp.tests.MazeMDPTest
```

Planning with Value Iteration (VI)

When we select the ThresholdTrainer we see Value Iteration converges
: within 13 ms in 105 iterations.

When we select ConvergenceTrainer we see Value Iteration converges
: within 8 ms in 13 iterations.
VI result : {3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 2, 2, 0, 2, 0, 1, 2, 0, 2, 0, 1, 2, 0, 0, 0, 0, 2, 2, 2, 0}

Planning with Policy Iteration (PI)

Similarly with ConvergenceTrainer, Policy Iteration converges: within 37 ms in 3 iterations
When applied ThresholdTrainer, it converges within 65 ms in 6 iterations.
PI result : {3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 2, 2, 0, 2, 0, 1, 2, 0, 2, 0, 1, 2, 0, 0, 0, 0, 2, 2, 2, 0}
For small size of data, Value iteration is faster but takes more number of steps. While Policy Iteration is slightly slower but converges in much less number of steps.

Learning with Reinforcement Learning Algo (RL)
We run QLearning with following params (we have to import the mdp_tests project and modify the params in MazeMDPTest.java)
lambda = 0.5, gamma = 0.95, alpha = 0.2, decay = 1,
strategy = EpsilonGreedyStrategy

```
java -cp mdp_tests.jar:lib/ABAGAIL.jar com.mdp.tests.MazeMDPTest
```

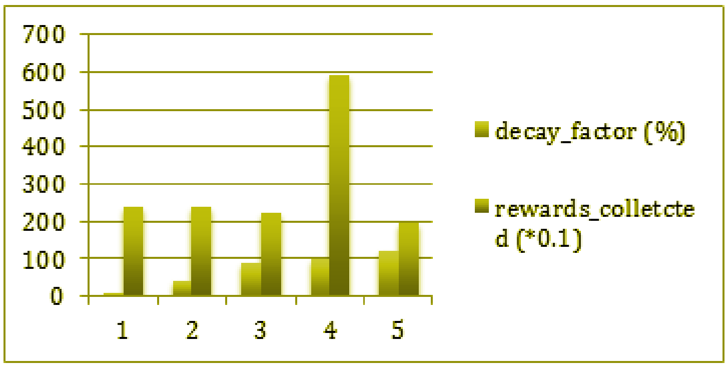
It runs 50000 iterations and takes 42 ms to acquire 837441 rewards.

If we change the strategy to DecayingEpsilonGreedyStrategy then no. of rewards starts decreasing with value of decay goes up.
But with a slower decay DecayingEpsilonGreedyStrategy (.3, .01) QLearning takes 39 ms and acquires 1236213.0 rewards.
Same way in place of FixedIterationTrainer, we can use ConvergenceTrainer to gain slight improvement of performance.
Nevertheless QLearning always return accurate result in every iteration if we consider DecayEpsilonGreedyStrategy

{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}
So far we were running 50000 iterations!
With 100 iterations, lets see the effect of increasing decay factor :

** numbers have been normalized for plotting the values :

decay_factor (%)	**rewards_colletcted (*0.1)**
10	241
40	238.6
90	224.8
99	592
120	194



Now lets just run say 10 or even smaller say 5 iterations!
Reduction in #iterations does not affect the performance of VI and PI.
But QLearning converges so fast, almost in no time !
So finally, QLearner is most preferable approach for small number of states as depicted by following tables.
Max Iterations => 10

approach	iterations	time	accuracy
Value Iteration	5	8 ms	~20%

Policy Iteration	4	40 ms	~20%
QLearner	10	0 ms	100%

Max Iterations => 100

approach	iterations	time	accuracy
Value Iteration	12	7 ms	20%
Policy Iteration	4	49 ms	20%
QLearner	100	2 ms	100%

** A sequence of actions is accurate if the agent can successfully reach the goal!
Now lets change the overall size of state space by considering slightly bigger size of maze :

```
#x###########
#     #
# #### #####
#     #
#o###########
```

Lets update the code **MazeMDPTest.java** by pointing to **testmaze_small.txt**
We can build the code using ant or directly run from Eclipse. Interestingly, Value Iteration shows poor performance. If we use ConvergenceTrainer, then we don't even get any result at all ! Accuracy is 0! Anyway when we choose ThresholdTrainer, at least the agent can now reach the Goal!
Policy Iteration offers better result than VI!
{3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 1, 2, 0, 2, 0,
1, 2, 0, 2, 0, 0, 2, 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, 0, 0, 0, 0, 0, 0}
Again QLearner emerges as the winner with lowest latency and highest accuracy!
Max Iterations => 100

approach	iterations	time	accuracy
Value Iteration	325	62 ms	~20%
Policy Iteration	3	117 ms	~50%
QLearner	100	4 ms	100%

Conclusion
So we can conclude for larger dataset Value Iteration offered best performance with lowest 'learning time' whereas QLearning always guaranteed to converge. For smaller dataset, QLearning performs best and Policy Iteration performs better than Value Iteration planner.