HW-5: Reinforcement Learning

Subhashini Venugopalan (EID: SV8754)
vsubhashini@utexas.edu

Abstract

This report presents the experiments and results of implementing modular reinforcement learning algorithms on gridworld. The task of navigating an obstacle ridden grid is divided into two sub-tasks (modules). One module is to learn how to reach the destination of the grid (end of the sidewalk) and the other module learns to avoid obstacles. We develop reinforcement learning agents that learns each of the modules separately. The final agent that navigates the obstacle grid is developed as a combination of these two individual modules.

1 Introduction

This experiment focuses on the task of implementing reinforcement learning algorithms on agents based on a modular approach. The objective in the grid world is to design an agent that can traverse a grid with obstacles and get to the other end of the grid/sidewalk. A reinforcement learning algorithm tries to learn a strategy to navigate the grid by experimenting in each step as opposed to having a fixed deterministic strategy that is formulated ahead of time (independent of the grid). The advantage of an agent that uses the reinforcement learning approach is that it is capable of choosing an optimal strategy and adapting to different grid layouts, different obstacle layouts, and also complex reward policies.

A reinforcement learning agent is defined as a Markov Decision Process (MDP) or a POMDP (partially observable markov decision process). This implies that an agent's actions is determined by the state in which it is in and the reward it gets by performing a chosen action. To be more clear, an agent has 4 main functions:

- measure the **value** of a state.
- determine a **policy** or the best action to perform in a particular state.
- determine the **quality** of a state and action combination.
- **update** and learn all the measures (above values) from experience.

1.1 Modular reinforcement approach

A modular approach to reinforcement is to divide the entire task (such as navigating the grid) into sub-tasks and have the agent learn each of the sub-tasks individually. Then you combine both sub-tasks together learning to weigh them appropriately in order to solve the whole task successfully. With respect to the gridworld, this translates to having an agent to learn two sub-tasks separately:

- a module that learns to walk to the other end (a destination) of a plain grid
- a module that learns how to avoid obstacles (in an obstacle world where there may not be a final destination)

The final step is to learn how to weight both modules in order to navigate a grid with obstacles and reach the final destination.

2 Methodology

The performance of a reinforcement learning agent depends largely on the design of the state space and the reward model. After defining the state space, the rate of learning can be controlled by the learning rate, the discount rate and exploration probabilities (i.e. the agent should try to deviate away from the best actions once in a while inorder to explore and arrive at possibly a more optimal strategy.)

- **State space model** is chosen based on the task or sub-task of the module. In a gridworld, the state can be defined as:
  - the co-ordinate position on the grid.
  - the neighbourhood (e.g. elements in the north, south, east and west directions, or all cells surrounding the current cell)
  - the type of current cell (e.g. the grid may consists of different types of cells and the state can be based on the type of the current cell or neighbouring cells.)
- **Reward model** is also decided based on the task and may vary for the same task if the state space is defined differently. The actions of the agent will be largely governed by the reward model. Some examples for reward models are:
  - reward only on completing the task.
  - reward for living. i.e. you might want to reward an agent just for moving and exploring the space.
  - reward for small subtasks such as moving in the direction of the eventual goal.
  - reward (or penalty) for not doing (or doing) an action.

2.1 Algorithms

The q-learning algorithms with $\epsilon-$greedy strategy was implemented for the modules. The exact algorithm implemented can be found in the book by Sutton and Burto [SB98]. The q-learning parameters to determine the value of a state ($V(s)$), quality of state and action pair ($Q(s,a)$), and policy of a state ($\pi(s)$) are implemented based on the algorithm described in the online version of the book http://webdocs.cs.ualberta.ca/~sutton/ebook/node40.html

2.2 Code snippets

Listing 1: Value of a State

```
def getValue(self, state):
  """
    Returns max_action Q(state,action)
  """
  actions = self.getLegalActions(state)
  if len(actions) < 1:
    return 0.0
  return max([self.QValues[(state, action)] for action in actions])
```

Listing 2: Updating Q Values

```
def update(self, state, action, nextState, reward, environment):
  """
    The parent class calls this to observe a
    state = action => nextState and reward transition.
  """
  oldQsa = self.QValues[(state, action)]
  nextActions = self.getLegalActions(nextState)
  if state=="TERMINAL_STATE":
    return
  if len(nextActions)==0:
    maxFutureValue=self.QValues[(nextState, action)]
  else:
    maxFutureValue=max([self.QValues[(nextState, nextAction)] for nextAction in ↩
        nextActions])
  self.QValues[(state, action)] = oldQsa + self.alpha * ( reward + self.gamma * (↩
      maxFutureValue) - oldQsa )
```

Listing 3: Determine Policy

```
def getPolicy(self, state):
  """
    Compute the best action to take in a state.  Note that if there
    are no legal actions (in terminal state) returns None.
  """
  maxqval=float("-inf")
  bestAction=[]
  actions = self.getLegalActions(state)
  if len(actions)==0:
    return None
  for action in actions:
    env=state
    if self.environment!=None:
      env = self.environment.getCurrentStateEnv(state, self.qtype)
    qValue = self.QValues[(env,action)]
    if qValue > maxqval:
      maxqval=qValue
      bestAction=[action]
    elif qValue==maxqval:
      bestAction.append(action)
  return random.choice(bestAction)
```

Listing 4: $\epsilon-$greedy action

```
def getAction(self, state):
  """
    Compute the action to take in the current state.  With
    probability self.epsilon, we should take a random action and
    take the best policy action otherwise.
  """
  legalActions = self.getLegalActions(state)
  action = None
  if len(legalActions)==0:
    return action
  if util.flipCoin(self.epsilon):
    return random.choice(legalActions)
  else:
    return self.getPolicy(state)
```

3 Experiments

Grid Grids of dimension 3×10 was considered (this was chosen since I also display the agent's motion, and it helps with visualizing the algorithm). The grid space can be changed.

Action An agent can perform 4 actions, moving north, south, east and west.

Episodes An agent learns to navigate the grid over multiple episodes. In each episode the layout of the grid is changed for the obstacle module. Note that in the walking module, the grid remains fixed.

3.1 Walking Module

State space The state space considered for the walking module is chosen as the grid co-ordinate (x, y). This choice is based on the fact that the agent should realize as it gets closer to the destination. Another choice for the state space I experimented with was by considering the neighbourhood of the cell in the four directions of action (north, south, east, west). However, the grid co-ordinate space proved to be better.

Reward model For the walking module the rewards were decided as follows:

- large reward (10) on reaching the end of the side walk.
- small reward $(+0.5)$ for taking action towards the goal and small penalty (-0.5) otherwise (deviating in a different direction).

We would like to mention that in the walking module the agent is rewarded on *departing* a state (as mentioned in Russel and Norvig [RN03]) i.e. when the agent performs an action a to move from state s_1 to state s_2, it learns the reward of state s_1 (after it has departed from it).

3.2 Obstacle module

State space For the obstacle module, the state of the agent was determined by the neighbouring cells in the north, south, east and west directions. The neighbouring cells could each take a value of "wall", "obstacle", or empty space. This choice of state was more appropriate considering that the layout of obstacles is changed in each episode. The chosen state space is reasonably small $(4^3 = 64)$ and can be learned well in a few (50) episodes.

Reward model For the obstacle module the rewards were decided as follows:

- medium penalty (-2) for bumping into an obstacle.
- small reward (0.25) for taking an action that leads to an empty space.

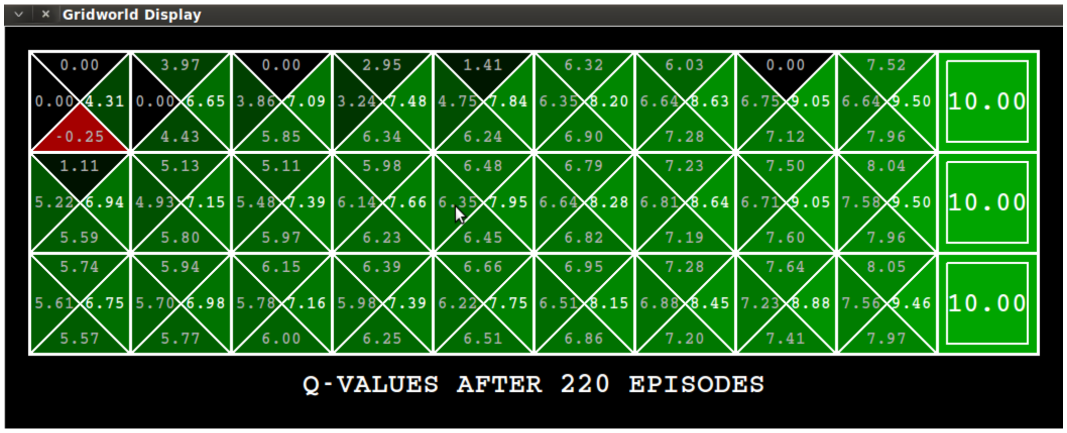
It is important to note that in our obstacle module the agent is rewarded on *arriving* at a state (the opposite of the walking module) i.e. when the agent performs an action a to move from state s_1 to state s_2, it learns the reward of state s_2. For example, if an agent went from s_1 and bumped into an obstacle in s_2, it gets a penalty (reward in state s_2) for bumping into the obstacle.

4 Results

This author prefers to demonstrate the correctness of the algorithms by visualizating them on a small 3×10 grid.

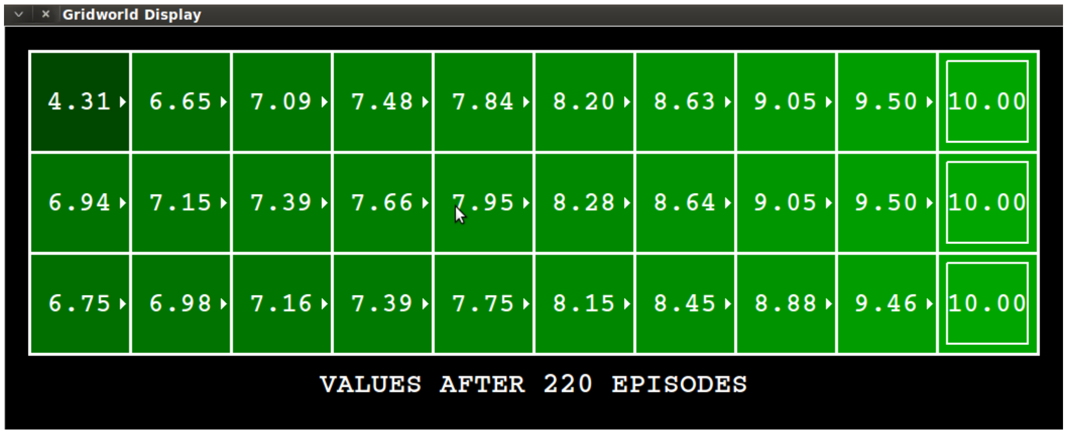
4.1 Walking module

- Destination reward = 10
- Reward for departing east = 0.5
- Penalty for wrong direction = -0.5
- Learning rate = 0.5
- Epsilon = 0.3
- Noise = 0.2 (exploration rate)
- State space = grid position (x, y)



0.00 / 0.00 / 4.31 / -0.25	3.97 / 0.00 / 6.65 / 4.43	0.00 / 3.86 / 7.09 / 5.85	2.95 / 3.24 / 7.48 / 6.34	1.41 / 4.75 / 7.84 / 6.24	6.32 / 6.35 / 8.20 / 6.90	6.03 / 6.64 / 8.63 / 7.28	0.00 / 6.75 / 9.05 / 7.12	7.52 / 6.64 / 9.50 / 7.96	10.00
1.11 / 5.22 / 6.94 / 5.59	5.13 / 4.93 / 7.15 / 5.80	5.11 / 5.48 / 7.39 / 5.97	5.98 / 6.14 / 7.66 / 6.23	6.48 / 6.35 / 7.95 / 6.45	6.79 / 6.64 / 8.28 / 6.82	7.23 / 6.81 / 8.64 / 7.19	7.50 / 6.71 / 9.05 / 7.60	8.04 / 7.58 / 9.50 / 7.96	10.00
5.74 / 5.61 / 6.75 / 5.57	5.94 / 5.70 / 6.98 / 5.77	6.15 / 5.78 / 7.16 / 6.00	6.39 / 5.98 / 7.39 / 6.25	6.66 / 6.22 / 7.75 / 6.51	6.95 / 6.51 / 8.15 / 6.86	7.28 / 6.88 / 8.45 / 7.20	7.64 / 7.23 / 8.88 / 7.41	8.05 / 7.56 / 9.46 / 7.97	10.00

Figure 1: The Q-Values for each state action combination learned for walking.



4.31 ▸	6.65 ▸	7.09 ▸	7.48 ▸	7.84 ▸	8.20 ▸	8.63 ▸	9.05 ▸	9.50 ▸	10.00
6.94 ▸	7.15 ▸	7.39 ▸	7.66 ▸	7.95 ▸	8.28 ▸	8.64 ▸	9.05 ▸	9.50 ▸	10.00
6.75 ▸	6.98 ▸	7.16 ▸	7.39 ▸	7.75 ▸	8.15 ▸	8.45 ▸	8.88 ▸	9.46 ▸	10.00

Figure 2: The Value of each state and best policy learned for walking.

4.2 Obstacle module

- Destination reward = 1
- Penalty of landing on obstacle = −2
- Learning rate = 0.5
- Epsilon = 0.3
- Noise = 0.2 (exploration rate, deviate from policy)
- State space = Four neighbouring cells in north, south, east and west (each taking value of wall, obstacle or empty space)

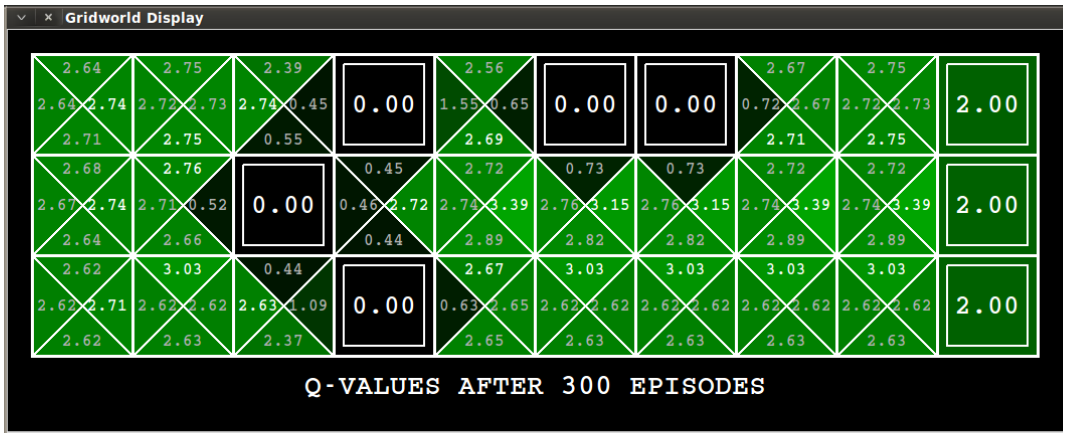


Figure 3: The Q-Values for each state action combination learned by the obstacle module.

Understanding the image

- Bright green represents higher value and dark represents low value.
- The dark green triangles inform us that the module learns to avoid moving in that direction and hence will avoid obstacles.
- Since the obstacle module did not have any specific direction, it just learns that all other directions (not leading to obstacles) are good. This can be noted from the fact that all other directions in each cell have the same value.

4.3 Combined module

- Destination reward = 35
- Learns from individual modules
- Learning rate = 0.5
- Epsilon = 0.3
- Noise = 0.2 (deviating from policy)
- State space = Neighbouring cells (N,S,E,W)
- Module weights = Walk (0.75) and Obstacle (0.25)

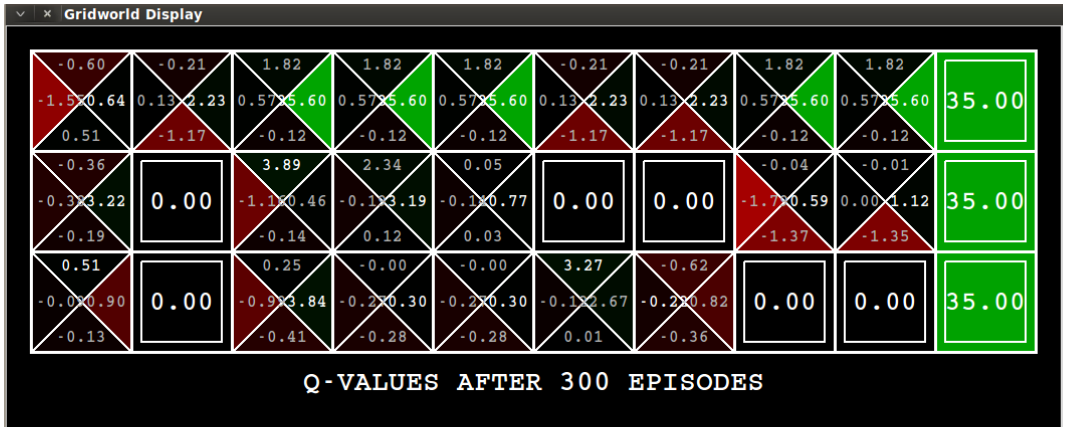
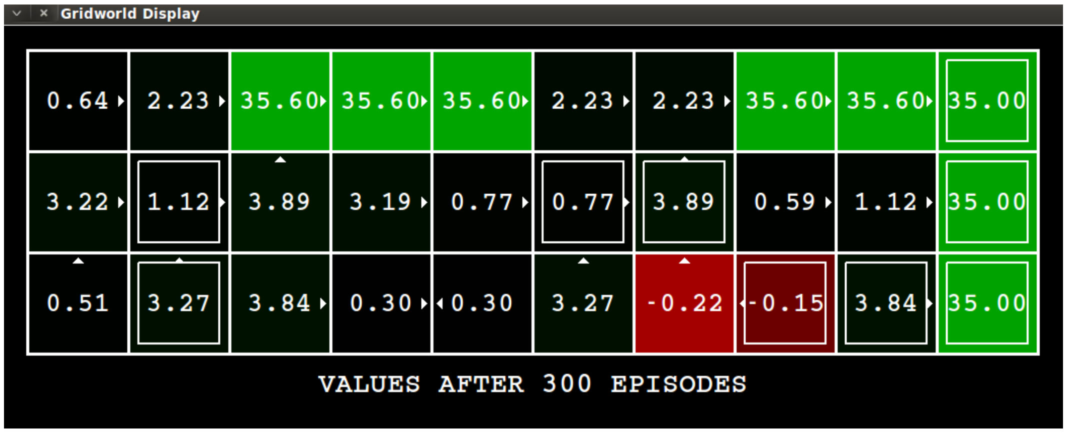


Figure 4: The Q-Values for each state action combination learned by the combined agent.



0.64	2.23	35.60	35.60	35.60	2.23	2.23	35.60	35.60	35.00
3.22	1.12	3.89	3.19	0.77	0.77	3.89	0.59	1.12	35.00
0.51	3.27	3.84	0.30	0.30	3.27	-0.22	-0.15	3.84	35.00

Figure 5: The Value of each state and best policy learned for walking.

Observations and explanations The following are some observations based on the experiment:

1. On a small grid, the number of iterations required by the walking agent to learn is less and that of the obstacle and combination agent is significantly more. This can be explained by the fact that in each episode the walking grid remains the same, where as the obstacle layouts keep changing for the other agents.

2. The walking agent takes much longer to learn on long side walk (since it's state space is defined by the grid position). Whereas the obstacle and combination agents learn faster on long side walks.

3. The learning times of the obstacle and combined agent is comparable.

4. The weights of the combined modular agent can change dramatically when the rewards for walking or obstacles is varied.

5. One can observe that the algorithms tend to converge but this takes a long time.

5 Additional Experiments performed

I performed some additional experiments but they did not necessarily result in any significant conclusion.

- Increasing the **length** of the side walk.
  - Walking agent learns a bit slowly.
  - Obstacle and modular agent learn faster. (Can be explained by state space choice)
- Varying **state spaces**.
  - For walking module, the neighbourhood state space did not work well (it did not seem to converge in a reasonable time). I am not sure why, it seems counter-intuitive to what I expected.
- Varying **weights** for modules. This was quite tricky, and it seemed to take a long time to find an appropriate weight for the combined agent when the walking and obstacle agent rewards are fixed. Normalizing the rewards helps.
- **Changing Noise**. Having a small amount of noise (0.2) and deviating from the best policy worked better than choosing the best policy at all times. This agrees with what one would expect. Since the layout of obstacles keeps changing, you may encounter some new state after several iterations.
- **Visualized** the agent motion. The agent can be controlled manually to see learning and can also be left to run on it's own for a determined number of episodes. The graphics for this was obtained from Prof. Dan Klien's course at UC Berkeley [Kli09]. I would also like to note that, only the framework for the agent and display for gridworld were borrowed and modified quite heavily to suit this assignment.

6 Code

To run the code, do the following on a terminal:

1. `cd src`
2. `Walk agent: python gridworld.py -a q -k 200 -g WalkGrid -q`
3. `Obstacle agent: python gridworld.py -a q -k 300 -g ObstacleGrid -q`
4. `Combined agent: python gridworld.py -a q -k 300 -g ComboGrid -q`

Choosing Options:

1. Help: `python gridworld.py -h`
2. `Manual mode: python gridworld.py -a q -k 5 -m -g ObstacleGrid`

The README file explains the options and details.

7 Manual Mode Figures

Agents leaving learning in their wake.

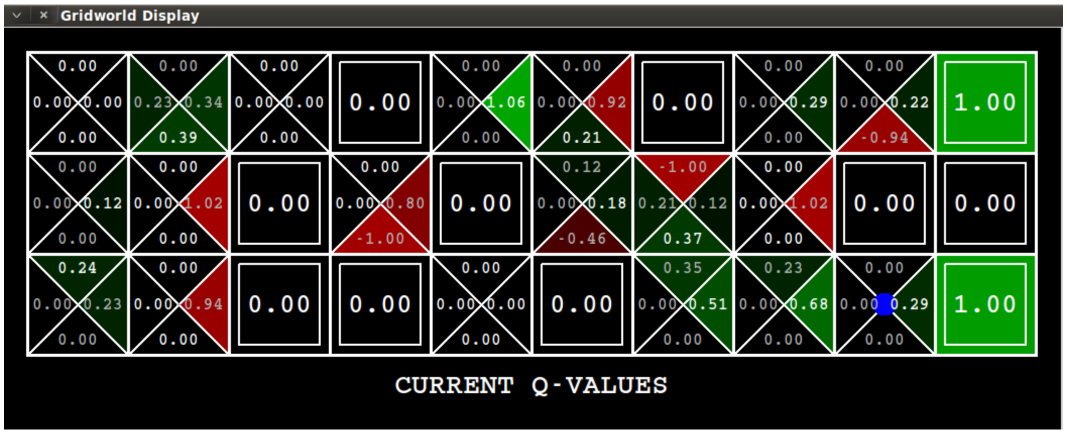


Figure 6: An obstacle agent - leaving learning in it's wake

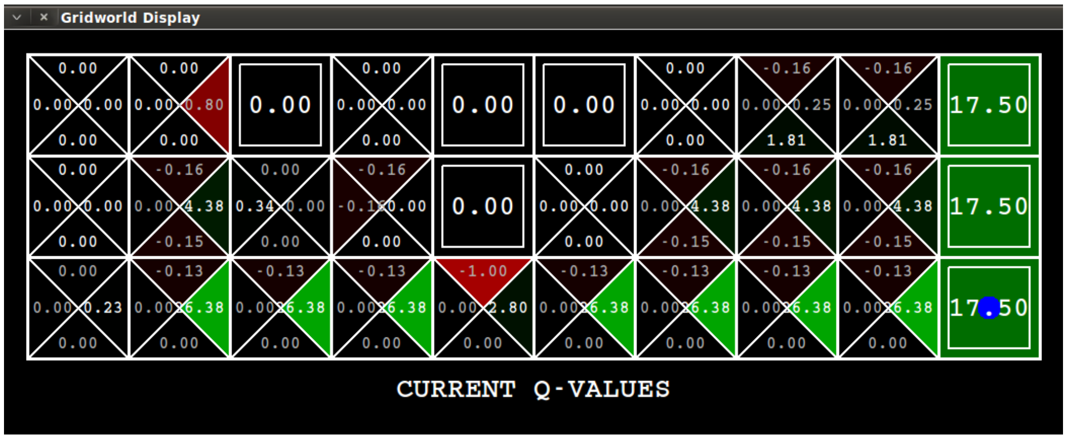


Figure 7: A combined agent learning to navigate the grid.

References

[Kli09] Dan Klien. Uc berkeley, reinforcement learning. http://inst.eecs.berkeley.edu/~cs188/fa09/projects/reinforcement/reinforcement.html, 2009. Accessed: 2013-04-19.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.