CS 7641 Machine Learning — Assignment 4

X. Sheldon Gu
xgu60@gatech.edu

What are Markov decision processes (MDPs)

The history of Markov decision processes (MDPs) can be tracked back to as early as the 1950s,[1] and since then MDPs are widely used in different areas, which include but not limit to economics, manufacturing, robotics and automated controls. MDPs provide a mathematical model for decision making, more precisely, MDPs are discrete time stochastic control processes. Typical MDPs contain four parameters:

S: refer to a set of possible world states.

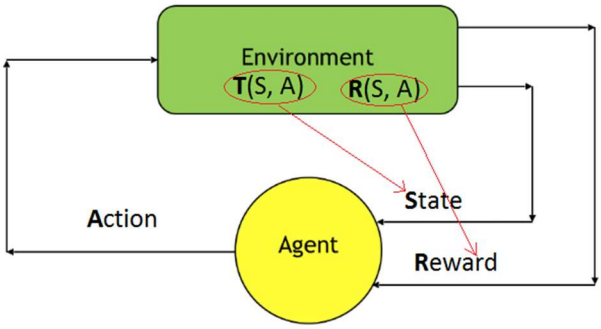
A: refer to actions that the agent can take under certain states.

R: refer to a function to calculate rewards based on states and actions.

T: refer to a transition function, takes old state S_1 and action A_1 to get the new states S_2.

One illustration has been given in Figure 1.

Figure 1. The illustration of a simple MDP.



At each time step, the agent is in a state, and the agent will make a decision to choose a specific action, which leads the agent to a new state and at the same time the agent will get a reward. Most time the transition function (T) and reward function (R) are invisible to the agent. The core problem of MDPs is to find a "policy" for the agent: how to choose actions based on current states to maximize the rewards.

Grid World

Grid world is a simple world, and the whole world is made by NXN grids. Figure 2A shows a 11X11 grid world. The agent (gray circle) lives in a grid and has four actions: move up, down, left and right. When the agent hits wall (black boxes) or grid world boundaries, it just stays at original place. The goal (blue box) is set at the beginning of game. The reward for reaching goal position can also be set at the beginning (e.g. 5 points), and any other move is - 0.1 points. The transition function of grid world is also quite straight forward. The grid world has all parameters that a typical MDP needs.

Although looks very simple, a grid world can represent many important applications in real world which impact our daily life. Figure 2B shows the guidance map from a working GPS, connecting a starting position to a destination. How GPS finds the shortest paths,

and how can it adjust outputs based on customer's preference (e.g. avoid highways)? The real map can be considered as a much more complicated version of grid world, and the agent can only move on grids overlays roads. By changing the reward function, e.g. give low value (-0.2 points) to grids on highway and relative high value (-0.1 points) to grids on local roads, GPS will provide a route to destination avoiding highways. Figure 2C shows a popular vacuum cleaning machine iRobot. How iRobot can vacuum all spots in a room and at the same time save the energy (not repeatedly vacuuming same spots)? It can be done by using a specific reward function. If the room is a NXN grid world, at the beginning, set all grids a high reward value (e.g 100 points). Then dynamically update the reward of grids, and set a low value to grids that have been visited (e.g. 50 points for the second visit, 0 for the third visit and -1 for later on visits). After period of time of training, iRobot will find the best way to clean a room.

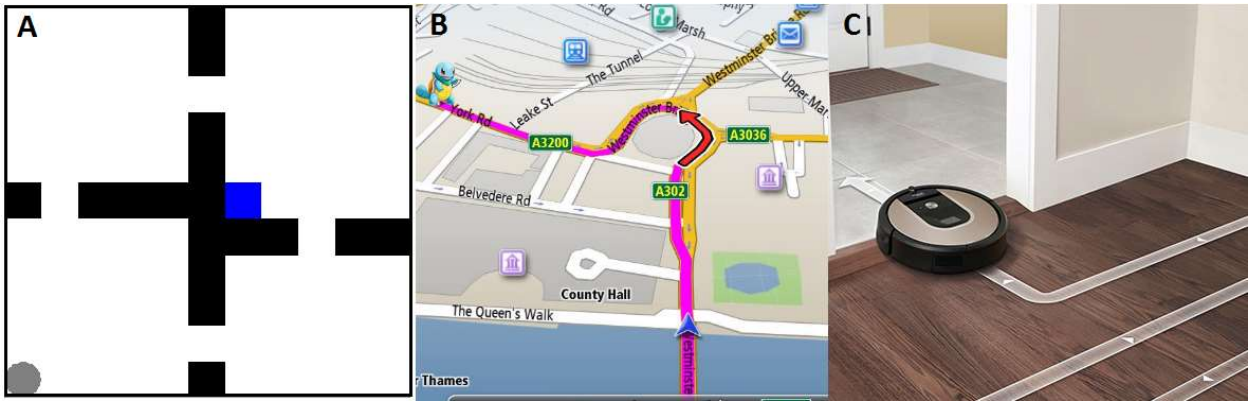


Figure 2. A. the illustration of a 11X11 grid world. The gray circle refers to the agent, and the blue box refers to the goal position. The black grids are walls which separate the 11X11 grid world into 4 rooms. **B.** an illustration of GPS guidance map from a starting point to a destination. **C.** an illustration of iRobot vacuum cleaner cleaning the room.

Block Dude

Block Dude is an old flash game that originally developed individually by Brandon Stemer and Fred Coughlin in later 1990's.[2] The puzzle game consists of eleven levels, and what shown in Figure 3A1 is the simplest level with slight modifications. The world of Block Dude consists of the agent itself (blue box), movable blocks (gray boxes), unmovable blocks (green boxes) and the door (black box, the goal). The game ends when the agent reaches the door. The task is how can the agent moves the movable blocks to reach the door. The agent's actions include change orientations, move forward, climb (one grid), move down, pick up and put down the movable blocks. Compared with Grid World which has less than NXN states, Block Dude obviously has much more states. Even the simplest version of Block Dude shown in Figure 3A1 has more than 1000 states (agent positions

x agent orientations x movable block 1 positions x movable block 2 positions). Another major difference between Grid World and Block Dude is that in Block Dude there are a few key steps that must be done to complete the task. As shown in the red box in Figure 3A(2-4), to overpass the 2 grid high green walls, the agent must move the movable blocks and use them as ladders. In other words, the agent must have a “plan” to complement the task. The solutions for solving Block Dude puzzle are used in Artificial Intelligence and providing convenience for our daily life. For example, a developed robot nanny can help to take care of new babies (Figure 3B). Here, the baby is the movable block and a baby cradle and a baby playground can be the goal position. The task will be move the movable blocks to the goal positions.

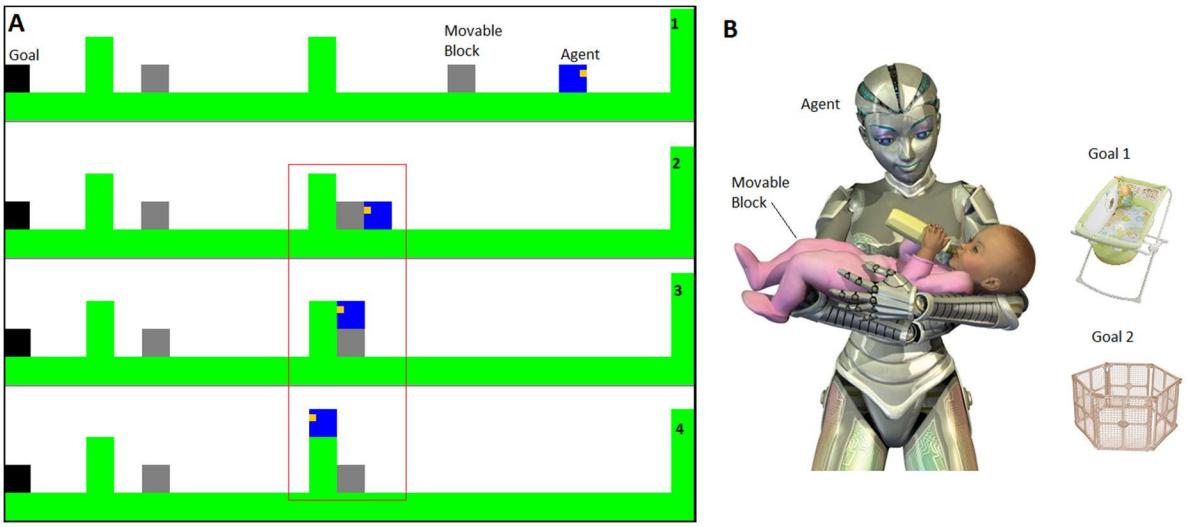


Figure 3. A. the illustration of Block Dude game at a low level setting. Blue box refers the agent, and the small yellow window on the blue box refers the agent’s orientation. Black box refers the goal position. The gray boxes are movable objects. The agent can change orientations, move forward, move up (only 1 grid) and down, pick up and put down movable blocks. To overcome the high green walls to reach goal position, the agent has to move the gray box and uses them as ladders (2-4 in red box). **B.** a robot nanny takes care of her master’s baby. After feeding the baby, robot nanny will move the baby to a baby cradle (goal 1) or a baby playground (Goal 2) by different settings.

Value iteration versus policy iteration

The core problem of MDPs is to find a "policy" for the agent: how to choose actions based on current states to maximize the rewards. Value iteration and policy iteration are two popular approaches to solve the problem. Value iteration is also called backward induction, which finds optimal policy by finding the optimal value function. The “Bellman equation” is used in value iteration.[1] Policy iteration algorithm manipulates the policy directly, and the iteration terminates when the policy does not change.[3]

In this experiment, classes *ValueIteration* and *PolicyIteration* from Brown-UMBC Reinforcement Learning and Planning (BURLAP) java code library[4] were both used. The

following are some parameters used for Grid World experiment. The parameters for Block Dude experiment is slightly different, which needs larger iteration number.

Grid World, the value iteration:

Planner planner = new ValueIteration(domain, 0.99, hashingFactory, 0.001, 100);

Where the discount factor gamma=0.99, the termination threshold maxDelta=0.001, and the maximum iterations maxIterations=100.

the policy iteration:

Planner planner = new PolicyIteration(domain, 0.99, hashingFactory, 0.001, -1, 1, 100);

Where the discount factor gamma=0.99, the termination threshold of value iteration maxEvalDelta=-1, so evaluation never terminate by value function, maxEvaluationIterations=1, maxPIDelta=0.001 and maxPolicyItrations=100.

The output is visualized in Figure 4 as color maps. Girds with high values are colored blue and grids with low values are colored red. The policies are shown as arrows for direction. After comparison between Figure 4A and C, B and D, it seems value iteration and policy iteration deliver same results. And since grid world is a simple world, it is quite obvious that the provided policies are correct.

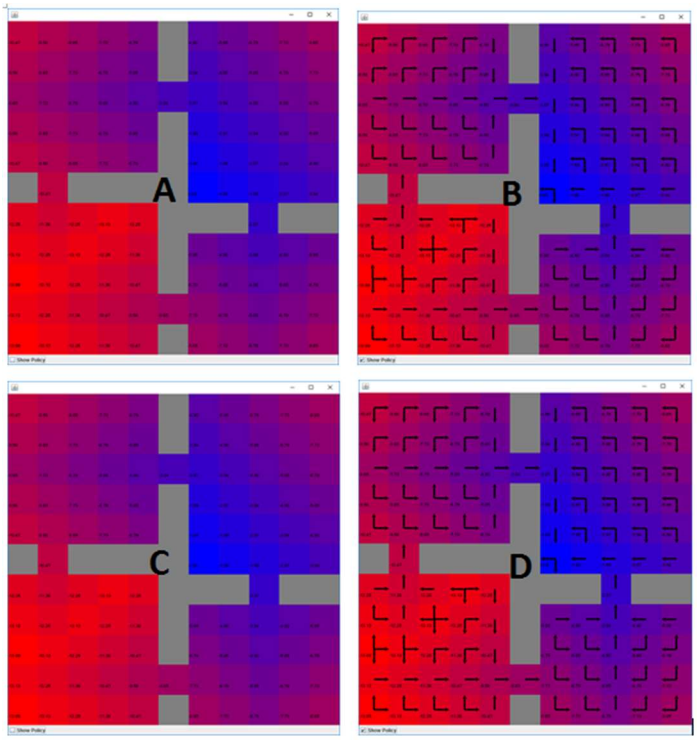


Figure 4. The output values (**A, C**) and policies (**B, D**) of a 11X11 grid world after value iteration (**A, B**) and policy iteration (**C, D**) using Brown-UMBC Reinforcement Learning and Planning (BURLAP) java code library. For value iteration: gamma=0.99, maxDelta=0.001, maxIterations=100. For policy iterations: gamma=0.99, max-EvalDelta=-1 (evaluation never terminate by value function), maxEvaluationIterations=1, maxPIDelta=0.001 and maxPolicyItrations =100.

The color maps provide very detailed information about value distributions and policies for each grid. However, they did not answer a few essential questions, e.g. how many iterations do they (value iteration and policy iteration) take to converge, how long do they take to converge, and which one converges fast. To answer these questions, two methods were added in the original code (*valueIterationExp* and *policyIterationExp*). Instead of setting a large iteration number, in both methods, different iteration numbers were tested, and with different maxDelta. When the termination threshold maxDelta was set as -1, the iterations never converge, so the running time continuously increases with iteration numbers. When maxDelta was set as 0.001, 0.01, or 0.1, the running time increases with iteration number and plateaus once it converges. The results are presented in Figure 5. Both value iteration and policy iteration converge after ~20 iterations. The policy iteration takes a bit longer running time compared with value iteration.

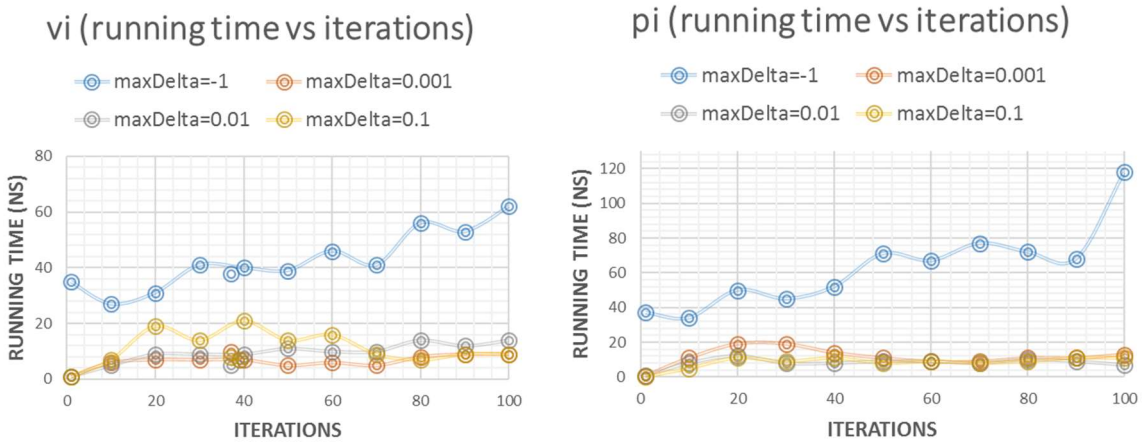


Figure 5. running time versus iterations for the 11X11 grid world. **Left:** value iteration. The experiment was performed by calling class: ValueIteration from Brown-UMBC Reinforcement Learning and Planning (BURLAP) java code library, setting discount factor gamma = 0.99, termination threshold maxDelta = -1 (never terminate), 0.001, 0.01, 0.1, maxIterations = 1 to 100. **Right:** policy iteration. The experiment was performed by calling class: PolicyIteration from BURLAP java code library, setting gamma = 0.99, maxEvalDelta = -1 (evaluation never terminate by value function), maxEvaluationIterations = 1, maxPIDelta = -1, 0.001, 0.01, 0.1, and maxPolicyItrations = 1 to 100.

Same experiments have been done on Block Dude. Since Block Dude has much more states, it is difficult to give a map view of value distribution and policies for each states. Instead, the final optimized policies are presented in Figure 6. It seems that both value iteration and policy iteration deliver same results, and are identical with human players' solution. It is worth to point out that final policies contain the essential steps to finish the game, using movable blocks as ladder to climb the wall (shown in red circles in Figure 6C).

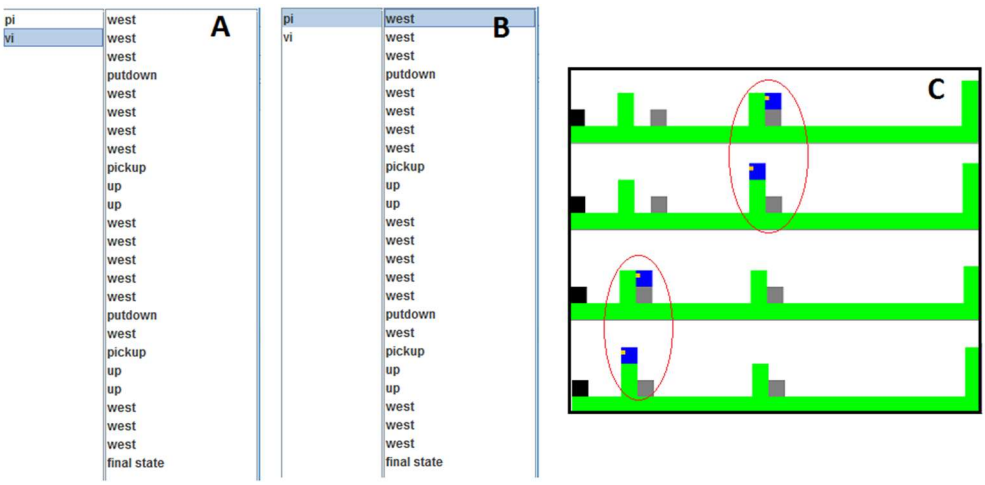


Figure 6. The output of value iteration (**A**), policy iteration (**B**) and detailed illustration of Block Dude agent's movement (**C**). using BURLAP java code library. For value iteration: gamma = 0.99, maxDelta = 0.001, maxIterations = 100. For policy iterations: gamma = 0.99, maxEvalDelta = -1 (evaluation never terminate by value function), maxEvaluationIterations = 1, maxPIDelta = 0.001 and maxPolicyItrations = 100.

To answer the same questions, e.g. how many iterations do they take to converge, how long do they take to converge, and which one converges fast. *ValueIterationExp* and *policyIterationExp* were also performed. The results are presented in Figure 7. Both value iteration and policy iteration converge after ~40 iterations. The policy iteration takes longer running time to converge (400 ns vs 300 ns) compared with value iteration.

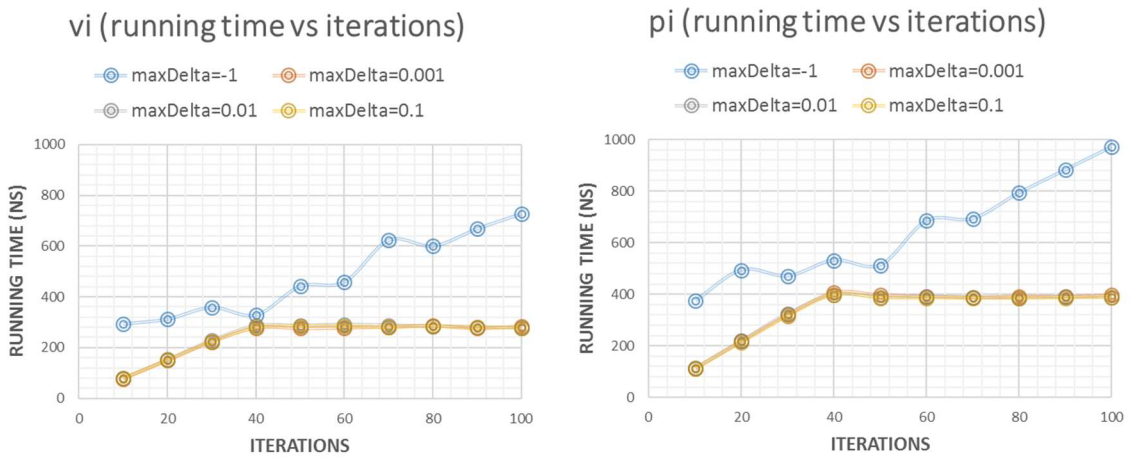


Figure 7. running time versus iterations for Block Dude experiment. **Left:** value iteration. The experiment was performed by calling class: ValueIteration BURLAP java code library, setting discount factor gamma = 0.99, termination threshold maxDelta = -1 (never terminate), 0.001, 0.01, 0.1, maxIterations = 10 to 100. **Right:** policy iteration. The experiment was performed by calling class: PolicyIteration from BURLAP java code library, setting gamma = 0.99, maxEvalDelta = -1 (evaluation never terminate by value function), maxEvaluationIterations = 1, maxPIDelta = -1, 0.001, 0.01, 0.1, and maxPolicyItrations = 10 to 100.

Using Q Learning algorithm to solve Grid World and Block Dude.

Q-learning is a model-free reinforcement learning technique. Unlike value iteration or policy iteration, Q-learning is able to compare the expected utility of the available actions without requiring a model of the environment.[5] In this experiment, class *LearningAlgorithmExperimenter* from Brown-UMBC Reinforcement Learning and Planning (BURLAP) java code library was used for parameter tuning. Class *QLearning* was also used to compute running time and output final policy.

The parameter tuning for Q Learning algorithm in solving Grid World (Figure 8) and Block Dude (Figure 9) are shown. Different values of discount factor (gamma), initial conditions and learning rates were examined. The final results were evaluated by cumulative steps and average reward. It seems that both discount factor and initial conditions have slight effects for both cumulative steps and average reward, while the learning rates have much more dramatic effects on both cumulative steps and average reward (Figure 8E, F and Figure 9E, F). The learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. A factor of 0.6 seems work both for Grid World and Block Dude experiment.

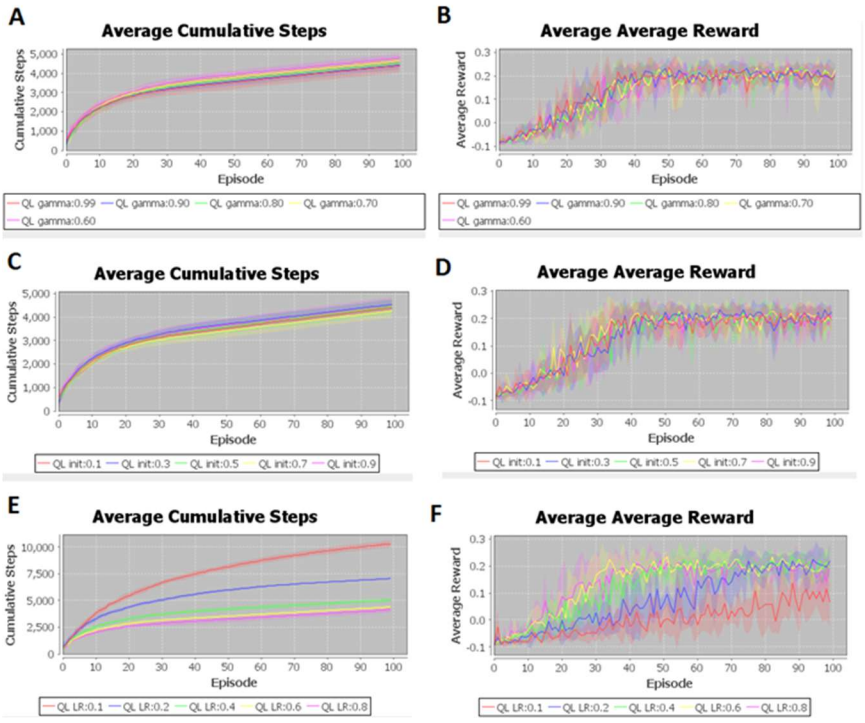


Figure 8. Parameter tuning of Q learning algorithm to solve Grid World problem using different discount factor gamma (**A, B**), different initial value (**C, D**), and different learning rates (**E, F**). A, C, E record the cumulative steps versus episodes, and B, D, E record average reward versus episodes. The present data points are average of five individual runs. In A, B, init=0.5, learning rate=0.6. In C, D, gamma=0.99, learning rate=0.6. In E, F, gamma=0.99, init=0.5.

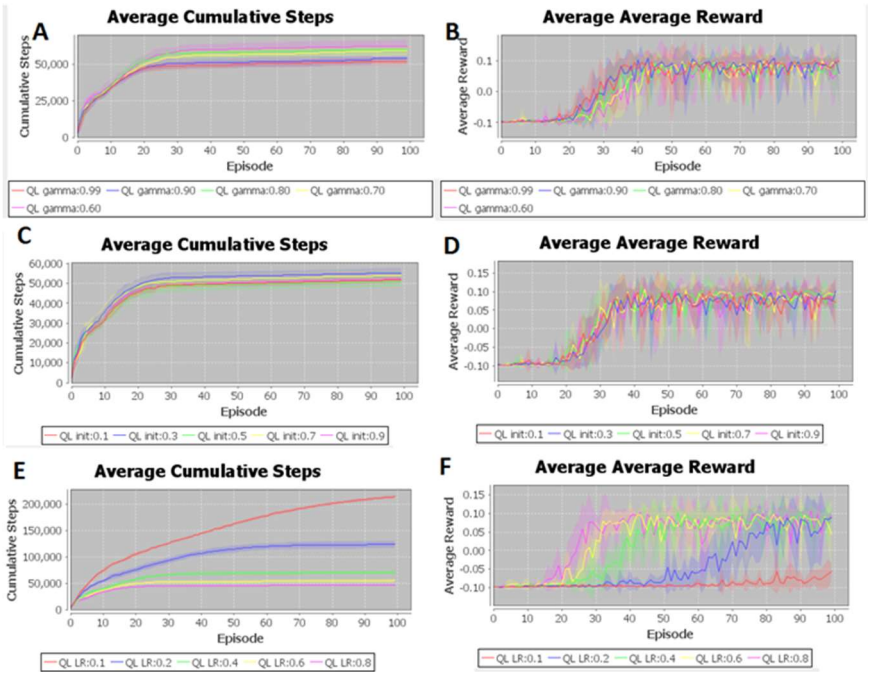


Figure 9. Parameter tuning of Q learning algorithm to solve Block Dude puzzle using different discount factor gamma (**A, B**), different initial value (**C, D**), and different learning rates (**E, F**). A, C, E record the cumulative steps versus episodes, and B, D, E record average reward versus episodes. The present data points are average of five individual runs. In A, B, init=0.5, learning rate=0.6. In C, D, gamma=0.99, learning rate=0.6. In E, F, gamma=0.99, init=0.5.

After parameter tuning, the optimized Q Learning algorithm was used to solve Grid World and Block Dude. The running time versus episodes were also recorded (Figure 10). It seems that Q Learning takes longer time to converge compared with value iteration and policy iterations for both Grid World (Figure 5) and Block Dude (Figure 7). Although the final optimized policies are identical (through manual comparison). The results are reasonable since Q Learning is a model-free algorithm and does not use a model of environment.

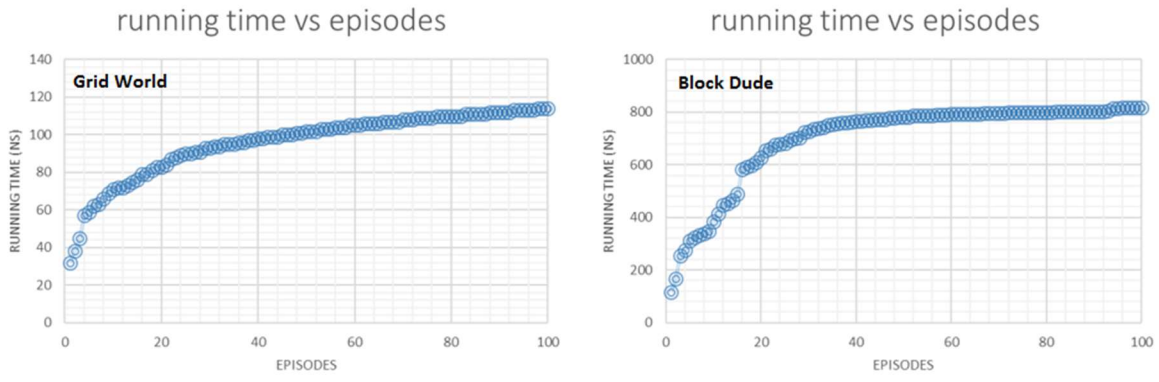


Figure 10. Running time versus episodes when using optimized Q learning algorithm to solve Grid World (Left) and Block Dude (Right). Gamma=0.99, init=0.3 and learning rate=0.6.

The influence of reward function (reward for wandering around)

Although Grid World and Block Dude have different state numbers (100 versus more than 1000), their behaviors in value iteration, policy iteration and Q Learning are very similar except one need longer time to converge. One possible explanation is that even though Block Dude has much more states, many of them are just ignored or bypassed by the agent under current reward function. Thus the reward function is manipulated to examine its influence on the agent's behavior using Q Learning algorithm.

In previous experiment, goal reward (the transition to goal state) was set as 5, and default reward (any non-goal state transition) was set as -0.1, which means any non-goal state transition will cost the agent 0.1 point. The default reward was changed into -1, -0.001, and 0, and Q Learning experiments were repeated. It seems there are only slight differences after default reward changes (data not shown). However, when the default reward changes into a positive value (0.1), interesting things happen (Figure 11).

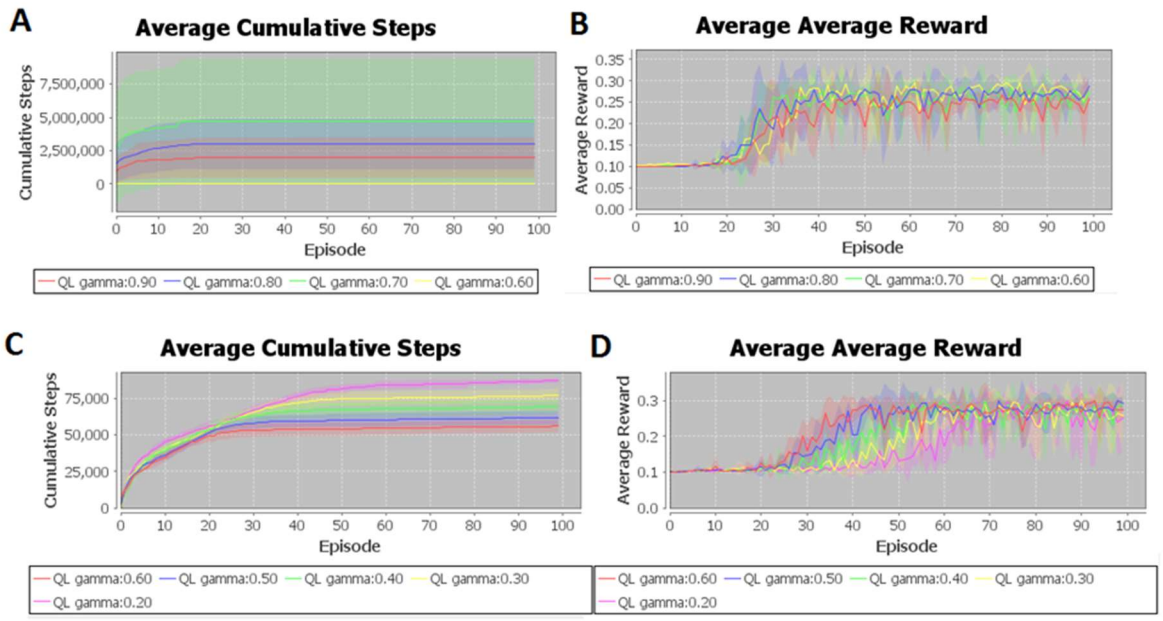


Figure 11. The influence of reward function on discount factor gamma for Q learning algorithm to solve Block Dude puzzle. The goal reward=5 and default reward=0.1. **A, B**, gamma=0.90, 0.80, 0.70, 0.60 **C, D**, gamma=0.60, 0.50, 0.40, 0.30, 0.20. A, C record the cumulative steps versus episodes, and B, D record average reward versus episodes. The present data points are average of five individual runs and init=0.5 learning rate=0.6.

In Figure 11, the discount factor starts with 0.90, since when gamma=0.99, the agent will just wander around and never terminate. That kind of behavior is easy to understand, since with a high discount factor (gamma=0.99), the agent prefers wandering around to get the infinity reward instead of the goal award. When gamma decreases, the agent need

to calculate which way gives more reward. Whether gets more wandering reward and less goal reward or gets less wandering reward and a higher goal reward. When the gamma=0.70, the agent takes more than 5,000,000 cumulative steps to reach the conclusion. When the gamma<0.6, the cumulative steps decrease to around 50,000 (99% less steps compared with gamma=0.7), which means with a low discount factor gamma, the agent quickly realizes the goal reward is more attractive, and the best strategy is to get the goal reward as soon as possible.

Reference:

[1] Bellman, R. (1957). "A Markovian Decision Process". Journal of Mathematics and Mechanics. 6.

[2] http://web.eecs.umich.edu/~gameprof/gamewiki/index.php/Block_Dude

[3] Howard, Ronald A. (1960). Dynamic Programming and Markov Processes (PDF). The M.I.T. Press.

[4] http://burlap.cs.brown.edu/

[5] Stuart J. Russell; Peter Norvig (2010). Artificial Intelligence: A Modern Approach (Third ed.). Prentice Hall. p. 649. ISBN 978-0136042594.