Second Assignment: Report

- Víctor Pérez Piqueras
- José Javier Atiénzar González

Content

1. Problem Description

As we did in the previous assignment, we have a maze with a series of elements, the maze used in the assignment is composed of a matrix of cells and walls around it that limit it. There is an agent(hamster) located randomly, that must reach the final state, which is the cheese.

We will have green cells which are walls, where the hamster can't move, blue cells which represent water, that make the hamster go slower and brown circle cells, which represent tunnels, if the hamster dives in any of the tunnels it will randomly go to another tunnel. The hamster should avoid cats.

It must be considered that:

- The reward is 100 if it gets the cheese.
- The reward is -100 if it encounters a cat.
- Both hamster and cat are final states.

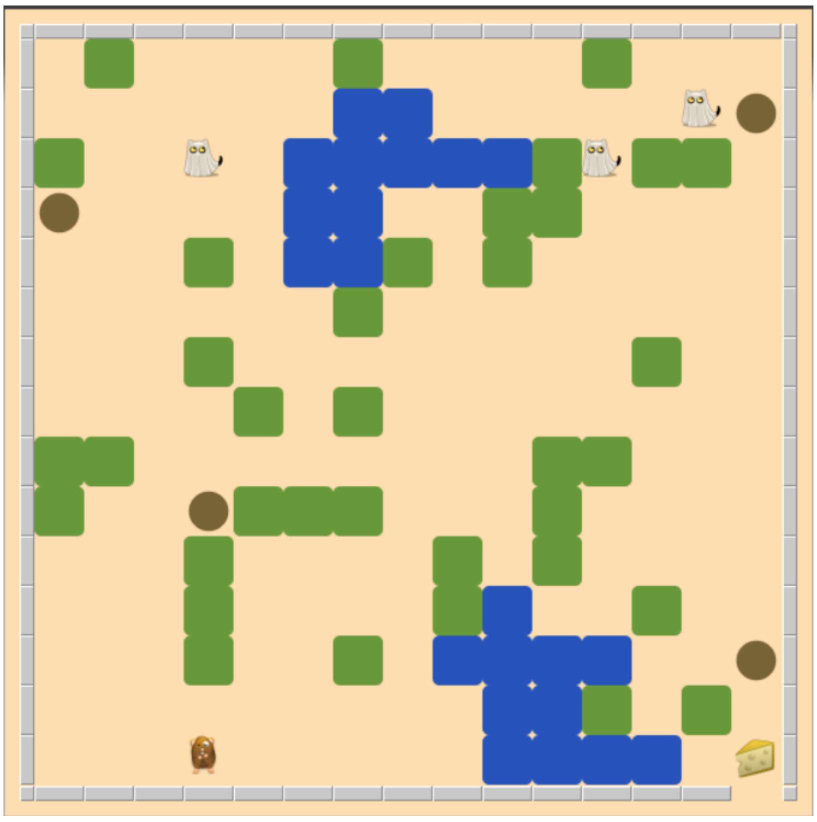

Figure 1: Maze

2. Problem formalization

The problem will be faced in 2 different ways:

- Markov Decision Process (MDP): The environment will be observable, and all components of the problem will be known. Given a Markov transition model that will determine for each state the probability of reaching other states. 2 algorithms are implemented:
    - Value Iteration: this algorithm applies the Bellman's update rule to the utilities of the problem to reach a steady state where the values correspond to the solution of obtaining an optimal policy.
    - Policy Iteration: it takes an initial policy and evaluates the policy, as in the value iteration algorithm, but only considering the actions of the policy given. Then, given those utilities, estimates the new policy. It finishes when there is no variation between a policy and its improved version.

- Model Free (MF): The environment now is not observable. In free models the agent learns the utility of carrying out actions in states. It is denoted as Q(s,a), that represents the maximum utility calculated over all sequences that start by selection the action *a* in the state *s.* The method uses the immediate reinforcement R(s,a) obtained from applying the action over the state.
    - Q Learning: the optimal policy is learned from local observations. The goal is to estimate the Q, that is stored in a table where each cell contains the utility from executing an action over a state.

This problem can be formalized as reinforcement learning one with the following:

- Set of States: the states that will be used for the problem definition are the states of the hamster at every move.
- Set of Actions: the possible actions that the hamster can do that are the 4 moves (Left; Right; Up; or Down) and Dive if it's in a hole.
- Transition model: They have a reward associated since they are penalized. The transition T(s,a,s') has a reward associated R(s) + Rt(s, a, s') where Rt represents the displacement time.
- Reward model: For each pair state-action, the hamster will be given a reward, depending on the new state generated by applying that action.
- Goal: find a policy that assigns an action to each state of the problem, so the agent finds the goal obtaining the maximum reward.

For this problem *Reinforcement learning* is the technique applied:

- The agent can now learn from the interaction with the environment what are the best actions (policy) to reach a goal.
- The agent will select actions according to a policy that aims to reach the goal.
- The agent will receive feedback of the results of its actions as rewards.
- This reward is used to modify the policy.
- It tries to maximize the reward in the long term.

3. Description of the implementation

- Maze: Allows building and representing a maze with the characteristics described above. The implementation supports on objects of class *utils.Position*, that correspond to (x, y) coordinates, and represent the maze with three data structures that can be accessed (at package level):

    - Cells: Matrix that represent the cells of the maze.
    - posHamster: Represents the position of the hamster, it is randomly generated at the start.
    - posCheese Represents the position of the cheese which is set in the bottom right corner.
    - posCats: Array that represents the position of every cat.
    - holeList Array that represents the holes of the problem.
    - The method generate(): makes a new random maze and sets the cats, cheese, walls, and water.

- MazeState: Extends the class *State* to represent the states of the problem. In such a case, each state is defined by the coordinates (Y; X) where the hamster.
    - The methods are:
        - Equals: compares if 2 states are the same.
        - hashCode: returns the hash value of the object.
        - toString: prints the position of the state.

- MazeProblem: Interface used so that both MazeProblems can be shown graphically.

- MazeProblemMDP: We will focus on the following:
    - *getPossibleActions*: given a state, it returns a list with all the possible actions:
        - If there is a hole it will add “Dive” to the list of possibleActions.
        - If there is a reachable cell in any of the 4 directions it will add it to the possibleActions list.
        - Then returns the list.

    - *getReward:* Returns the reward of an state.
        - If it is a cheese it returns 100.
        - If it is a cat it returns -100.
        - Else it returns a 0.
        - *isFinal*: checks if the state is a cat or a cheese and returns true.
        - *getTransitionReward*: given an initial state, the action and the state it wants to go to this function returns a reward considering the distance between both states, depending on the type of cells it will make it worse if it goes into water or better if it dives into a hole.
        - getAllStates: Returns a collection with all possible states.

    - getTransitionModel returns the transition model for a state and an action.
    - mazeTransitionModel: Generates the transition model for a certain state in this particular problem. Assumes that the action can be applied.

- MazeProblemMF:
  We will focus on the following because the other methods are the same as in the MDP:
  - mazeTransitionModel. Generates the transition model for a certain state in this particular problem. Assumes that the action can be applied.
  - readNewState, for every state and action it will check if it encounters a wall, if it encounters it the state will not change, else it will create a new solution state with the new state which is then returned.

- MazeAction: Implements the interface *Action*. In the maze, and regardless of the state, the available actions are four: *{UP, DOWN, LEFT, RIGHT, DIVE}*.

- MazeView: Allow showing a maze and the effect of a sequence of actions.

- Value Iteration (MDP): Implements a method.
  - learnPolicy:
    - It firstly checks if the problem is an instance of the MDPLearningPRoblem, creates a new utilities HashMap.
    - Then checks if the state is final if it is it will set the reward of the state in the utilities HashMap, if not it will set the utilities to 0.
    - We will iterate all the states, if the state is not final it will set the auxiliary utility to (0.0), for every action in the state it will apply the following formulae.
      newUtility = U(s')=R(s)+y*max[T(s,a,s')*U(s')], if it is greater than the aux utility aux will take the new utility value and save the action.
    - It will save the difference between the utilities and then update the utilities matrix for that state. It will update delta if the difference is bigger than auxdelta and will set the policy for the action. It will do this until auxdelta is greater or equal to maxDelta, which means that it is converging.

- Policy Iteration (MDP): Implements three methods.
  - learnPolicy:
    - It firstly checks if the problem is an instance of the MDPLearningProblem, then it creates a random policy and an auxiliary.
    - Then it creates an empty utilities HashMap.
    - For every state of the policy it will set a random action if it is not final.
    - It will evaluate the policy and update the utilities, then will improve the policy with the given utilities. It will do this until policies are the same and will return the solution policy.
  - policyEvaluation:
    - For every state which is not final it will set the utilities to 0.

        - We will save the utility of the state, calculate the new one and calculate the difference between them, if the difference is greater than auxdelta it will update auxdelta. It will do this until auxdelta is bigger than what we permit and finally will return the utilities.

    - policyImprovement:
        - We create a new policy, then for every state we get the first action and calculate the expected utility for it and save it.
        - For every possible action for that state we calculate the expected utility. Compare it and save the action that has the best utility. We will update the new policy and return it.

- Q-Learning:
    - learnPolicy:
        - Iterates over a number of iterations. Every iteration the algorithm selects an action according to the QTable, reads the new state and reward and updates the QTable with the new data, until a final state is reached.
        - Then it generates the optimal policy from the QTable.
    - Improvements:
        - The algorithm is greedy and sometimes it does not converge to optimal policies. The action is chosen randomly with probability *e* and greedily with probability *(1-e).*
        - Using α with decay: α(n)=1/n, the algorithm allows important changes at the beginning, but only small ones when the number of interactions of the agent with the environment is high and it must have reached a good policy.

4. Performance evaluation

Algorithm\Size	10	20	30	40
ValueIteration	-20,81	-1,58	-9,59	-9,01
PolictyIteration	-21,24	-1,61	-9,71	-9,00
Q-Learning	13,54	22,20	5.20	4,24

Table 1: Average utilities, gamma 0.9, seed 0.

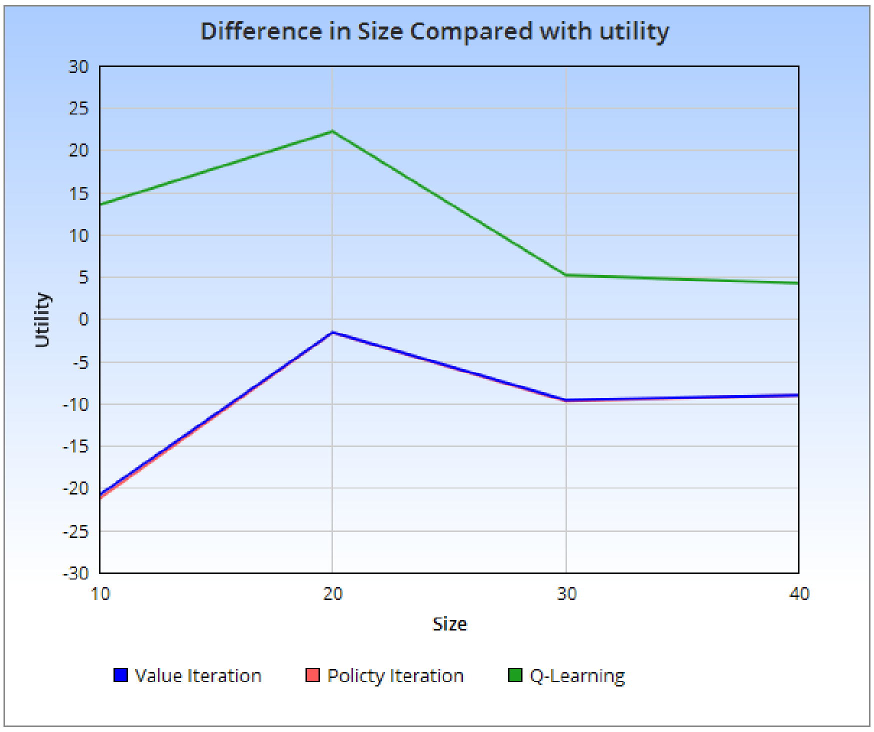

Figure 2: Average utilities, gamma 0.9, seed 0.

Algorithm\Size	10	20	30	40
ValueIteration	-14.16	-3.40	-4.44	-14.56
PolictyIteration	-14.19	-3.30	-4.42	-14.54
Q-Learning	1.09	5.40	0.5	1.58

Table 2: Average utilities, gamma 0.8, seed 0.

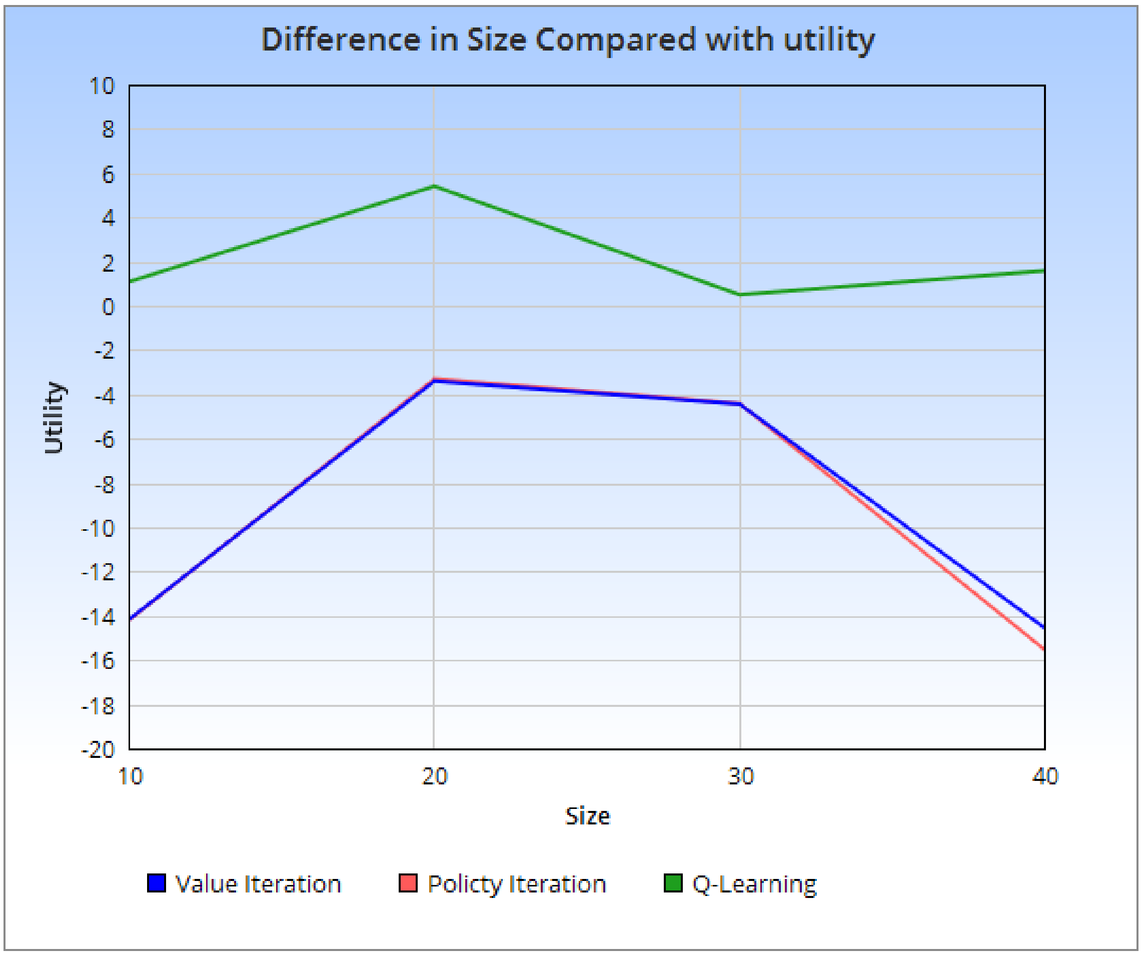

Figure 3: Average utilities, gamma 0.8

We can see that having 0,9 as gamma made the utilities get better as the size of the problem got bigger, but with gamma 0.8 when the problem got bigger the utility starting to worsen.

Algorithm\Size	10	20	30	40
ValueIteration	-6,62	1,82	-5,17	5,55
PolicyIteration	-7,05	1,69	-5,29	5,56
Q-Learning	12,45	16,8	4,7	2,66

Table 3: Average utilities difference.

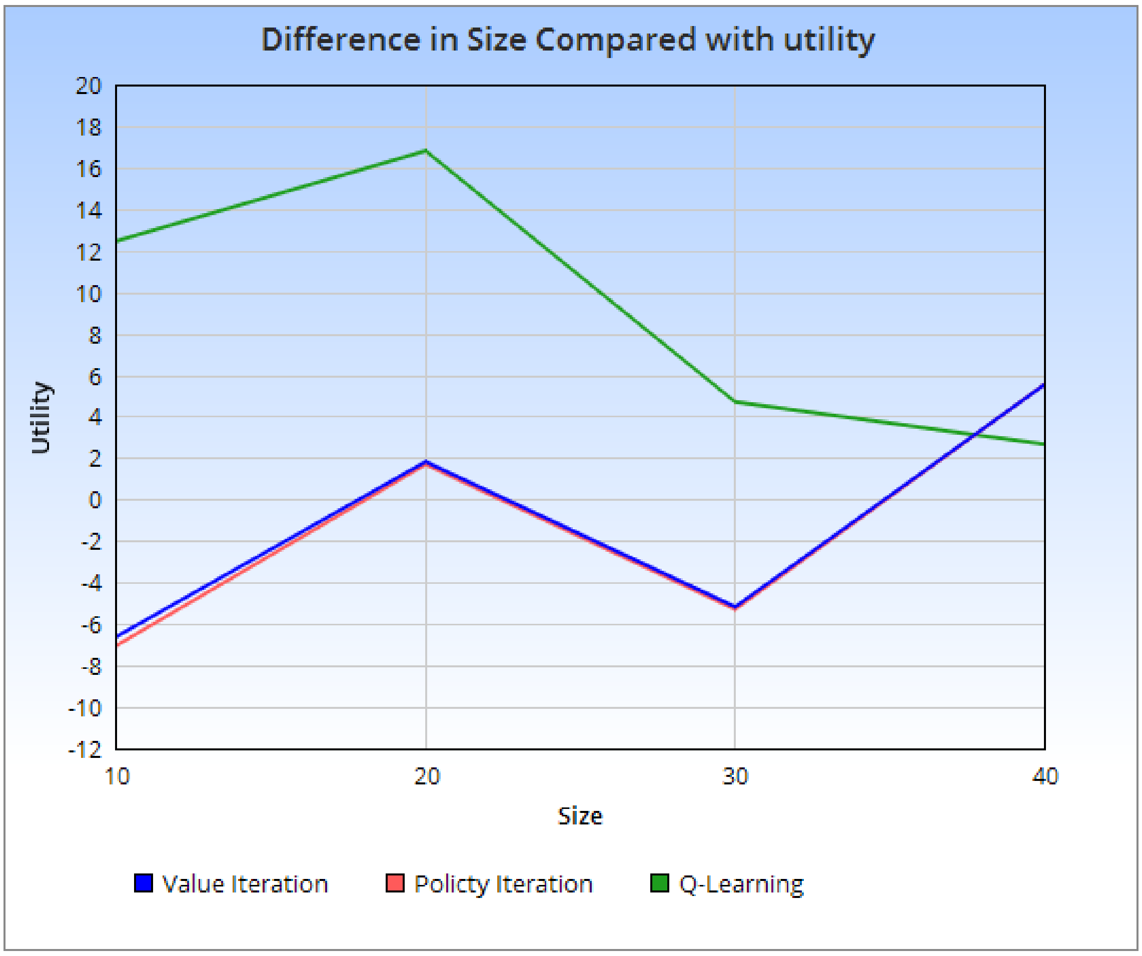

Figure 4: Difference in utilities with varying gamma.

We can see that changing gamma leads to a difference of utility between the same size. As the size grows the utilities difference in tends to be the same.

4.3 Choosing an action with a probability of being random:

- Problem: Gamma=0.9, size=20, seed=0, α =0.01, nº iterations=10000, algorithm=qLearning.
- Average utility over iterative executions of the algorithm:

probGreedy	1	0.95	0,9	0,85
Average Utility	4,5	8	4,45	4,425

Table 6: variation in probGreedy

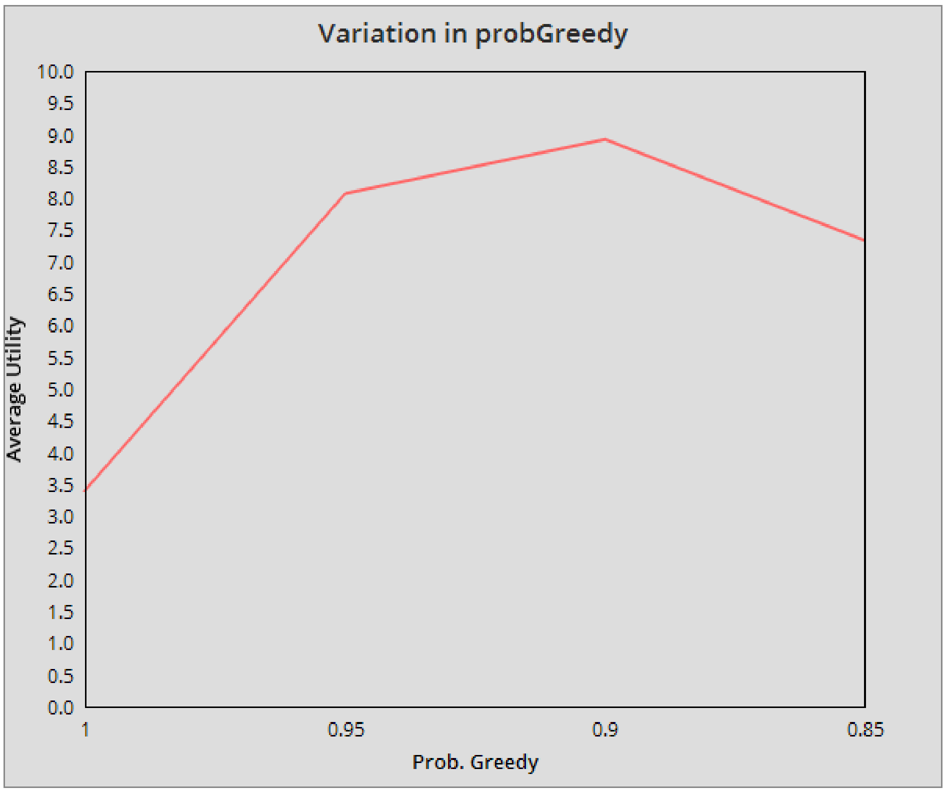

Figure 5: Variation in probGreedy

As it is shown in the Figure 5, a small possibility of taking an action randomly helps the algorithm to get better utilities, because it is a greedy one, and it can happen that it does not find the best utilities. Despite that, if the probability of choosing an action randomly is too high, the algorithm will not work as expected, because the behaviour of the algorithm loses importance in detriment of choosing randomly the actions.

4.4 Using α with decay:

- Problem: Gamma=0.9, size=20, seed=0, α =0.01, nº iterations=10000, algorithm=qLearning.

Algorithm	Average Utility
Q-Learning with α decay	-13.17
Q-Learning without α decay	-12.43

Table 7: α with decay

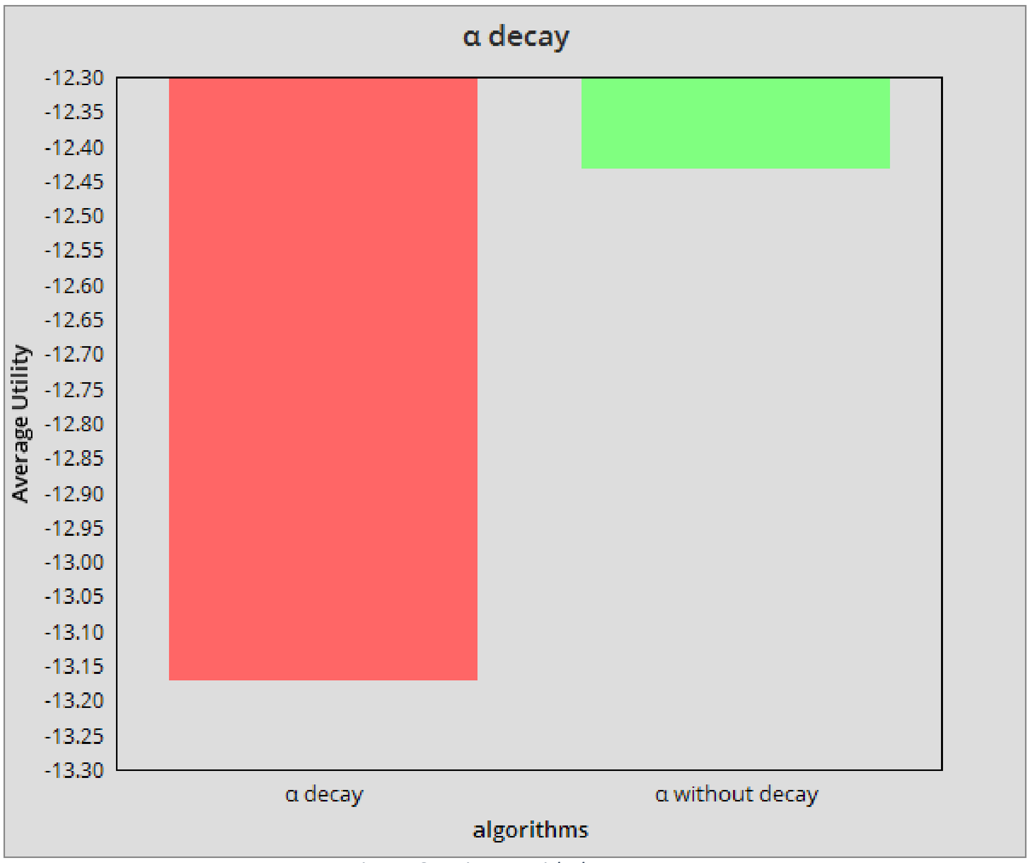

Figure 6: Using α with decay

For this problem, when the α parameter is reduced as the number of iterations increases, it makes the algorithm allow less changes. As the obtained utility is higher without α decay, it makes us think that for this problem it is better to allow high changes even in further iterations.