SAPIENZA
UNIVERSITÀ DI ROMA

The report on a project
for Artificial Intelligence course

Probabilistic Reasoning and Learning

"Fire Control"

Reinforcement Learning (RL)

Dynamic Q-learning with Experience Replay (DQLER)

Project and report are done by: Nazgul Mamasheva
Supervisor: Prof. Luca Iocchi

Introduction

Previously I worked on “Fire control” project. This project is a Multi Agent System project, which simulates wildfires and unmanned aerial vehicles(UAVs), which will try to extinguish wildfires. The main aim of that project was to implement a Control Network Protocol (CNP) simulation, which will allow agents to communicate with each other, make auctions for tasks(wildfires) and choose “wisely” some tasks among others and then extinguish those chosen wildfires. When an agent goes to the chosen task, i.e. to the field on fire, that agent have to make an exploration of that area. The “Fire control” project was about standard programming without any machine learning components.

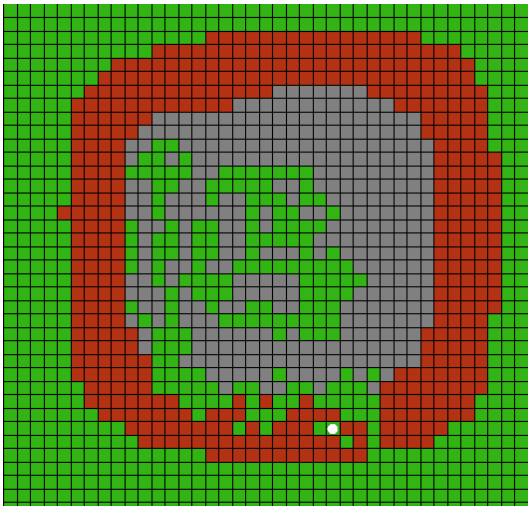
So, I thought what if an agent(UAV) will be trained for wisely exploring and extinguishing the wildfires. The wildfires tend to extend in similar way, if agent will face a new wildfire, it will know how to act effectively.

Description of the problem

The world is a two dimensional grid of size *w* and *h*. Each cell has the following properties:
- Position, a pair *(x,y)* identifying the position of the cell
- Type, each cell assumes one of the following types:

- NORMAL, cell is in a good shape and trees are safe
- FOAMED, a fire has been previously extinguished and trees are safe
- FIRE, fire is devastating the cell
- BURNED, the cell is completely lost

Cells normally start with status NORMAL. After a while, a fire starts to propagate, cells move from NORMAL to FIRE. Cells reached by UAVs in time, are treated with a special foam that can extinguish the fire. After the treatment, a cell cannot be ignited again; cells treated with the special foam move from FIRE to FOAMED. After a certain time, a burning cell moves from FIRE to BURNED and cannot go back to the NORMAL status again.



UAV has limited perception range: the camera mounted on UAV has the footprint that coincide with the size of a cell; i.e. it perceives only the status of the cell above which it is located. Agent store its own local knowledge as a collection of known cells. When a cell is inspected from a close range (i.e. at the level of a single cell), the agent automatically updates their knowledge with the information about current location. An agent has to select a cell within the set of available cells coming from those that lie in the area of agent’s current task.

white circle is an UAV; red cells are areas on fire; green cells are either normal trees or foamed trees; grey cells are cells burned out with no possibility to be recovered;

Formal model of the problem

- The state space of the problem is S = { s_1, s_2, … , s_k }, where S is a set of states,

s_k is the cell on the field with (x, y) coordinates and k is m*n.

If the width and height of the field are m and n respectively, i.e. width = m and height = n, then number of states will be m*n.

- A is a set of actions, where $a_t \in A$ is the action executed by the agent at time t.

A = { Upper-Left, Up, Upper-Right, Left, Right, Down-Left, Down, Down-Right }

The max number of possible actions from current state to next state is 8, i.e. neighbor cells of the current cell. The number of possible actions might be less than 8, if some of neighbor cells are invalid.

(x-1, y-1)	(x, y-1)	(x+1, y-1)
(x-1, y)	(x, y)	(x+1, y)
(x-1, y+1)	(x, y-1)	(x+1, y+1)

- P_a (s, s') is the transition function, performing an action a at state s, the agent arrives at state s'. In dynamic environment the transition function is still deterministic, i.e.

P^t (s, s') = $P^{t+\Delta t}$ (s, s')

- R_a (s, s') is the reward function, indicating the reward agent receives after performing action a at state s leading to s'. r_t indicates the reward received at time t. In dynamic environment the reward function is not always deterministic, i.e. R^t (s, s') ≠ $R^{t+\Delta t}$ (s, s')

- Q-function:
  - α (alpha), α ∈ [0, 1] is a learning rate
  - γ (gamma), γ ∈ [0, 1] is the discount factor, indicating the priority of immediate rewards compared to later rewards
  - e (epsilon), γ ∈ [0, 1] an exploration probability for e-greedy method

Solution algorithm

For this dynamic environment problem as a solution was implemented “Dynamic Q-learning with Experience Replay” (DQLER) algorithm.

Q-learning algorithm: Q-learning uses its last experience to update its policy.

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha[r_t + \gamma \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t)]$$

e-greedy exploration method: e-greedy is used along with Q-learning, with e ∈ [0, 1].

$$a_t = \begin{cases} \arg\max_a Q(s_t,a) & \text{with probability } 1-\epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

Experience Replay algorithm: Experience Replay uses the same Q-learning function, but with different approach for updating policy. Instead of using the last experience to update the policy, it will store experiences in buffer for later use. Those experiences in buffer later can be used multiple times for updating the policy.

Input: RL algorithm L, exploration probability ϵ, discount factor γ, learning rate α, number of experiences to replay N, number of replays K, number of iterations before learning Z

1: $Q \leftarrow Q_0$ // Initialize Q-function
2: $M \leftarrow \emptyset$ // Set of experiences to replay
3: **while** not reached stopping criterion **do**
4: $c = 0$ // iteration counter
5: **for** $t = 1, 2, .., T$ **do**
6: $a_t = \begin{cases} \arg\max_a Q(s_t,a) & \text{with probability } 1-\epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$
7: perform action a_t
8: observe new state s_{t+1} and reward r_t
9: $E \leftarrow \{s_t, a_t, r_t, s_{t+1}\}$
10: $M \leftarrow update(M, E, N)$
11: **end for**
12: $c = c + 1$
13: **if** $c = Z$ **then**
14: $Q \leftarrow L(Q, M, K, \gamma, \alpha)$
15: $c = 0$ // restart counting
16: **end if**
17: **end while**

on the line 10 in above algorithm, updating M will be executed as algorithm below. Instead of saving all experiences E = { s_t, a_t, r_t, s_{t+1} } to buffer M, it will save last N experiences. If M is full, then the least recent experience will be removed from M and the most recent experience will be added to buffer as a last element.

Input: Database M, experience E, number of experiences N

1: **if** $size(M) < N$ **then**
2: $M \leftarrow M \cup E$
3: **else**
4: remove least recent element from M
5: $M \leftarrow M \cup E$
6: **end if**

Output: M

Dynamic Q-learning with Experience Replay (DQLER): DQLER is based on Q-learning and Experience Replay with a modified e-greedy exploration strategy for dynamic environments. It uses the Experience Replay algorithm in above.
The agent tends to go back to visited locations multiple times. For this project time is crucial, therefore execution should prevent agent to revisiting already visited states multiple times.

On the line 6 in Experience Replay algorithm instead of using standard e-greedy method, it will use another strategy in below: A_n – actions at state s_t that lead to states which are not visited by agent. A_v – actions at state s_t that lead to states which are already visited by an agent. If the buffer of visited states A_n is not empty, then strategy will choose the action among all actions in A_n which has the maximum Q. If A_n is empty, it will choose action from A_v which has the maximum Q.

$$a_t = \begin{cases} \arg\max_a Q(s_t, a \in A_n) \text{ if } A_n \neq \emptyset \\ \arg\max_a Q(s_t, a \in A_v) \text{ if } A_n = \emptyset \end{cases} \quad \text{w.p. } 1-\epsilon$$
$$\text{random action} \qquad \text{w.p. } \epsilon$$

Implementation

Data structure:

If the width and height of the field are m and n respectively, i.e. width = m and height = n, then number of states will be k = m*n. Number of actions from one state to another state is maximum 8. It might be less, if two states are not neighbor by position.

Q matrix	action_1	action_2	…	action_8
state _1				
state _2				
…				
state _k				

There is no reward matrix, because environment is dynamic, which means reward of each cell will change over time. After each step iteration the simulation updates each cell's status in forest field. So, if program need to get a reward from exact cell, it will check the status of that cell at that time and if, for example, cell's status is fire, then reward will be 1.0. Agent can move only within the area of the wildfire, cells of which can be on fire, extinguished or burned.

Type of the cell	Reward
CellType. FIRE	1.0
CellType.EXTINGUISHED	-0.25
CellType. BURNED	-0.5

Static variables:

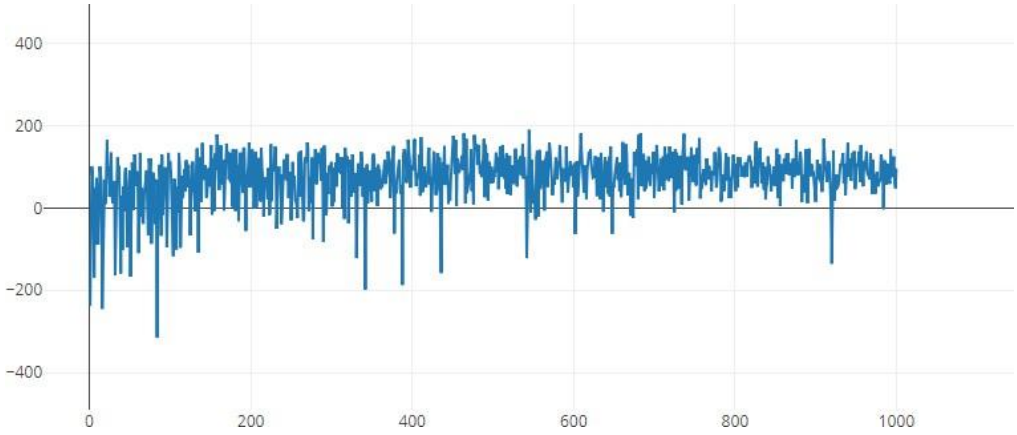
- N – number of experiences to store in M
- M – buffer for store last N experiences
- Z – number of simulations before updating Q matrix
- K – mini-batch, after each Z simulations an experience from M will be chosen randomly K times in order to update Q matrix

Main implementation details:

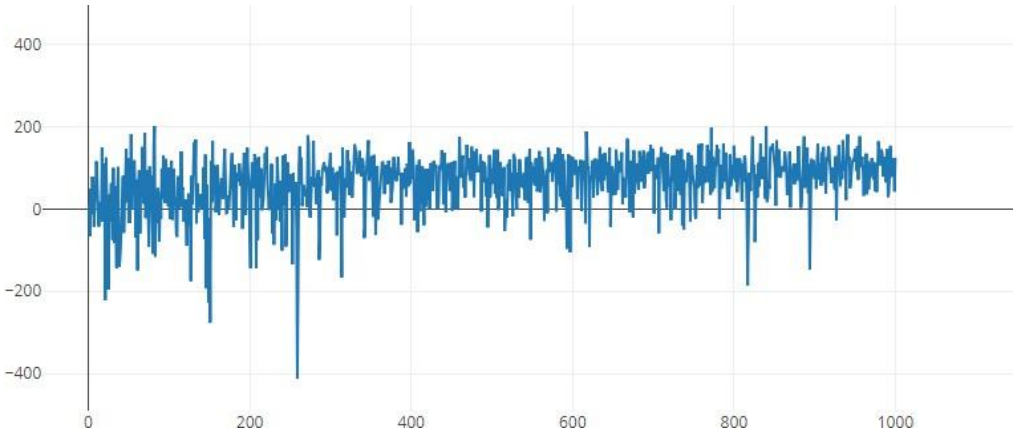
The project was done on MASON library for simulation. The program is developed on java 8. The implementation was done by algorithms which were described in above. There was added new file Buffer.java with simple class for storing experiences and deleted DataPacket.java file from initial files, because for this project we don't need to implemented Contract Net Protocol simulation, so we don't need this script. The reinforcement learning components were implemented in two scripts, they are UAV.java and Ignite.java. The other given initial scripts were not changed. The model and graphic parts are strictly separated in MASON. So, if I want to run on console I will enter one command, if for visual representation another command. For this project only one agent was involved. For each simulation agent's first move will be the origin of the wildfire, i.e. centroid position of the task(fire).

Experimental evaluation

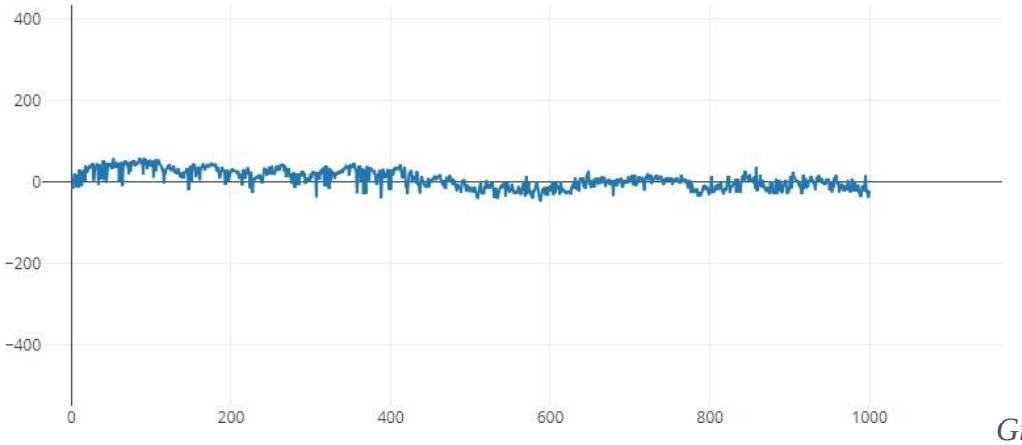
	Graph-1: DQLER	Graph-2: DQLER	Graph-3: Q-Learning
Number of simulations	1000	1000	1000
α (alpha)	1.0	0.1	0.1
γ (gamma)	0.99	0.9	0.9
e (epsilon)	0.1	0.1	0.1
N	50	100	-
Z	1	1	-
K	10	10	-



Graph-1: DQLER



Graph-2: DQLER



Graph-3: Q-Learning

Discussion of the results

1st and 2nd graphs show us there are some learning progresses by using DQLER. The agent does not learn quickly and it makes some progresses gradually. Graph 1 shows better performance than Graph 2. It might be because of large learning rate $\alpha = 1$ and large discount factor, which shows the importance of next state's policy. And 3rd graph shows that Q-learning fails for this problem. Q-learning is not successful in non-stationary environment.

Conclusion

Working on dynamic system by involving machine learning component was interesting project experience. I got more familiar with MASON library, which allows to make real world simulations. This project can be extended by involving more agents, make them to share their knowledge about the exploring areas. Also, it will be interesting to implement other learning techniques for dynamic environment. The implementation can be improved by detailed checking and finding out weaknesses for special cases.