CLSQL: Continuous Local Search Q-Learning

Improved Q-Learning Algorithm Based on Continuous Local Search Policy for Mobile Robot Path Planning

Abdallah Amr Mostafa Osama Tarek Shohdy Zeiad El-shabrawy

Faculty of Engineering
Egypt Japan University for Science and Technology
MTE311 Seminar on MTE
Dr. Haitham El-Hussieny

February 27, 2023

Table of Contents

Motivation

Problems in the QL algorithm:

- slow convergence speed
- large number of iterations

Improved Q-learning algorithm based on a continuous local search policy, CLSQL.

1. A prior knowledge to initialize the Q-table.
2. Adding the local search policy.
3. Adjusting ϵ-greedy policy.

Environment Prior Knowledge

- Q-tables in the initial state are equal to 0.
- Distance function is used to determine the prior knowledge of the environment.

$$D_s = \sqrt{(x_E - x)^2 + (y_E - y)^2}$$

$$Q(s,a) = \begin{cases} \frac{1}{D_s} & D_s > 0 \\ 0 & D_s = 0 \end{cases}$$

Environment Prior Knowledge

Algorithm 1: Prior Knowledge

Initialize Agent State $s(x, y)$, Destination State $s_E(x_E, y_E)$, Obstacle State s_{Obs}

Repeat

    if $s == s_{Obs}$ then

        $D_s = 0$

    else

        $D_s = \sqrt{(x_E - x)^2 + (y_E - y)^2}$

Until All s traversal completed

Local Search Policy

- As the size of the environment increases, dimensions increase.
- Large number of iterations.

Local search policy can simplify the complex environment by:

1. By setting a Local Environment based on s_{start} or s_l.
2. Centering of s_{start} or s_l and obtain the local environment grid.
3. s_l are determined based on prior knowledge.

Local Search Policy

Algorithm 2: Local Environment

Initialize LG, LS, $s_{start(x,y)}$, $s_I(x,y)$
for $i = \lceil -\frac{L_S}{2} \rceil, ..., 0, ..., \lfloor \frac{L_S}{2} \rfloor$ do
    $j = \lceil -\frac{L_S}{2} \rceil, ..., 0, ..., \lfloor \frac{L_S}{2} \rfloor$ do
        if $i + x \geq 0, \& i + x < G_S, \& j + y \geq 0, \& j + y < G_S$ then
            $LG_{i+x,j+y} \leftarrow GG_{i+x,j+y}$
    end for
end for

Local Search Policy

Search effectiveness were increased by determining s_l in the local environment.
We use the Priority Queue (PriQ) to determine the priority of s_l

$$PriQ = D_s + \frac{GS}{|r_{obs}|} P_s$$

GS is the Global Environment Size
P_s is the sum of reward in eight neighborhoods of the state s,

Local Search Policy

Algorithm 3: Intermediate Points

1 Initialize s_E, s_I

2 if $s_E \in LG$ then

3 $\quad s_I = s_E$

4 else

5 $\quad PriQ = D_s + \frac{GS}{|r_{obs}|} P_s$

6 $\quad s_I = argmin(PriQ)$

7 Return s_I

Local Search Policy

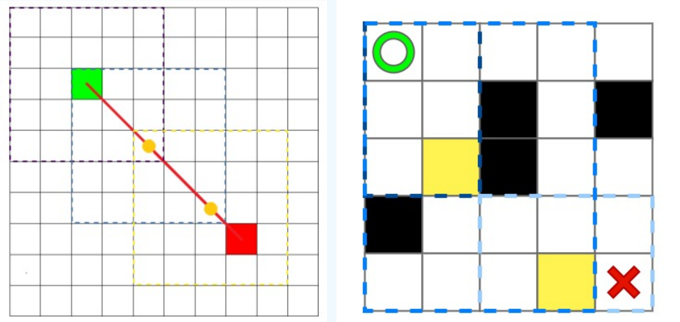

Figure: Grid maps with intermediate point.

Dynamic ϵ-greedy

Adjusted ϵ-greedy policy:

1. ϵ is initialized at the beginning of each local environment search.
2. ϵ increases when the agent collides with obstacles.
3. ϵ decreases when the result is the non-optimal path.

Optimal Steps

when number of learning steps at this stage is far greater than the local environment size ($steps > LS^2$), optimal solution cannot be obtained.

Present s_l in the local environment is deleted, and then another new s_l is determined again.

Global optimization can be achieved only when local optimization is achieved.

The optimal steps correspond to the diagonal distance between two points p_1 and p_2.

$$D = |x_2 - x_1| + |y_2 - y_1| + (\sqrt{2} - 2) * min(|x_2 - x_1|, |y_2 - y_1|)$$

Search and Iteration Process

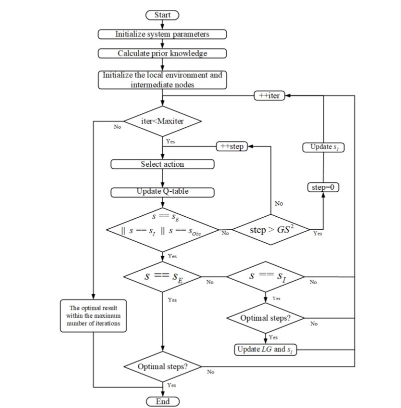

Figure: The flow of the CLSQL algorithm

Table of Contents

Implementation

```
import numpy as np
import math
import gym
import random
from time import sleep as sleep

env = gym.make("FrozenLake8x8-v1",is_slippery=False)

n_actions = env.action_space.n
n_states = env.observation_space.n

n_episodes = 40
max_steps = 275

gamma = 0.99
learning_rate = 0.86

epsilon = 0.99
max_epsilon = 1.0
min_epsilon = 0.01
decay = 0.1

rewards = []

q_table = np.zeros((n_states,n_actions))
```

Implementation

```
for s in range(n_states):
    if s in {35, 41, 42, 46, 49, 19, 52, 54, 59, 29, 63}:
        q_table[s,a] = 0
        continue
    for a in range(n_actions):
        if a == 0:
            st = s - 1
            if st < 0:
                continue
        if a == 1:
            st = s + 8
            if st < 0:
                continue
        if a == 2:
            st = s + 1
            if st < 0:
                continue
        if a == 3:
            st = s - 8
            if st < 0:
                continue
        x = int(st / 8)
        y = st % 8
        if x == 7 and y == 7:
            q_table[s,a] = 10
            continue
        q_table[s,a] = 1 / math.sqrt((7-x)**2+(7-y)**2)
```

Implementation

```python
for episode in range(n_episodes+1):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    for step in range(max_steps):

        e_e_tradeoff = random.uniform(0,1)
        if(e_e_tradeoff >= epsilon):
            action = np.argmax(q_table[state,:])
        else:
            action = env.action_space.sample()
        new_state, reward, done, info = env.step(action)
        if reward == 0 and done == True:
            q_table[state,action] = q_table[state,action] + learning_rate * ((reward-10) + gamma * np.max(q_table[new_state,:]) - q_table[state,action])
        elif reward == 1 and done == True:
            q_table[state,action] = q_table[state,action] + learning_rate * ((reward+9) + gamma * np.max(q_table[new_state,:]) - q_table[state,action])
        else:
            q_table[state,action] = q_table[state,action] + learning_rate * ((reward) + gamma * np.max(q_table[new_state,:]) - q_table[state,action])

        total_rewards += reward
        state = new_state
        if(done == True):
            break
```