MSc Computer Science

School of Computing, Engineering and Digital technologies

Artificial Intelligence

Frozen Lake using Artificial Intelligence

Name: Yaswanth Sai Chinthakayala

Student ID: W9640628

Contents

1. Abstract

This report implements various artificial intelligence techniques with Frozen Lake which is a gymnasium environment. This environment is similar to maze, the agent needs to reach the goal without falling into ice holes. Q-learning is used for reinforcement learning and for graph algorithms Depth First Search (DFS) and Dijkstra are used. The methodology involves training an agent using q-learning where the agent takes decisions through a q-table, the table values update for every episode and DFS, Dijkstra uses nodes and graphs for finding the path. This analysis shows that the reinforcement learning gradually increases its accuracy with training with a set of parameters. While DFS finds the path that may not be optimal, Dijkstra finds the optimal path choosing the best among all the paths. The results are evaluated with the path length and time taken. This comparison of AI techniques shows its strength and limitations and allows to understand and use them in appropriate environments. The findings highlight the wider implications for AI applications and go beyond the problem domain.

2. Introduction

Artificial Intelligence has become a trending technology where many challenging issues are being solved in many fields. In this project, a gymnasium environment called Frozen Lake is solved by using various AI techniques. The main objective of this Frozen Lake environment is to find the path from start node to goal node without encountering any ice holes.

The Frozen Lake environment is a difficult task, requiring an agent to navigate a grid-world with the goal of reaching a designated location while avoiding hazards. Our primary objective is to investigate the RL and graph algorithms in addressing this problem. RL, specifically Q-learning, enables the agent to learn optimal strategies through trial and error, adapting its decision-making based on experienced rewards. On the other hand, graph algorithms, including DFS and Dijkstra's, are employed to model and analyse the Frozen Lake as a graph, exploring paths and determining the shortest route.

This study not only contributes to the understanding of AI techniques in solving complex problems but also aims to compare the outcomes of graph algorithms in the specific context of the Frozen Lake scenario.

3. Method

Gymnasium is a popular AI tool kit by open AI. This toolkit has popular environments like classic control, toy text and Atari mainly developed for reinforcement learning where the agent makes decisions by interacting with the environment.

In the artefact, Frozen Lake is implemented which is a part of toy text environments. In frozen lake, the agent needs to traverse from start node to goal node without falling into any holes. This environment has a special feature which turns on slippery mode, enabling this doesn't allow agent to move in required direction sometimes if the land is slippery.

Figure 1: Frozen Lake

The Agent has action space of four discrete values. They are:

0	Move left
1	Move down
2	Move right
3	Move up

The agent has an observation space of sixteen discrete values in this case, Generally the observation space is equal to the total no of cells in the frozen lake.

There are only positive rewards in this environment, they are no negative rewards for this.

Reward for reaching the goal: +1

Reward for falling in hole : 0

Reward for not reaching : 0

For termination of the episode the conditions are:

- Agents falling into the ice holes.
- Agent traversing from start to goal.
- If agent takes more than 100 steps.

3.1 Gymnasium terms:

Render:

Render helps in visualizing the environment in different modes such as human and rgb. Human rendering opens a pygame window, whereas the rgb mode renders the image in the jupyter file itself.

Reset:

This function resets the environment and gives the current information of agent.

Step:

This function helps the agent to move. Choose any random action and pass it to step. For instance, pass step (2) for frozen lake, the agent moves right.

Close:

This helps the applications like pygame or other apps, to close. If this is not passed at the end, the window will stay open in not responding mode.

This environment will be trained by reinforcement learning using epsilon-greedy approach using q-learning. The agent will take actions with more q-table values. This algorithm balances both the exploration and exploitation in learning improving its accuracy.

4. Reinforcement Learning

Reinforcement Learning is a part of Machine Learning, RL is nothing but learning by doing. Agents learn to choose the right moves/decisions by interacting with the environment. The agent trains to maximize the reward over time by making random moves. The agent learns by feedback in form of the positive or negative rewards.

The key words of reinforcement learning are agent, environment, state, action, reward and policy.

Agent	The entity that makes the decisions.
Environment	The external system which the agent performs actions.
State	Current circumstances, configuration of the environment.
Action	The set of possible moves/decisions that agent can make.
Reward	Positive feedback for performing appropriate actions.
Policy	This is the strategy which agent uses to perform next action based on current state. These actions lead to greater cumulative rewards.

There are various policies which are used for reinforcement learning. The agent's internal policy is updated by receiving positive and negative feedback. This is done repeatedly which improves its decision-making abilities.

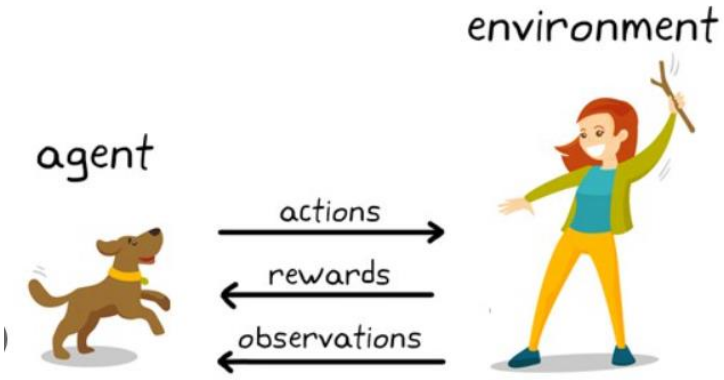


Figure 2: Reinforcement learning

As they provide a more dynamic approach than classic machine learning, reinforcement learning techniques like Monte Carlo, state–action–reward–state–action (SARSA), and Q-learning are revolutionizing the industry.

Three different kinds of RL implementations are:

- Policy-oriented: A deterministic approach or policy is used by RL to maximize cumulative reward.
- Value-oriented: The goal of RL is to maximize any given value function.
- Model-driven: The agent learns to function within the limitations of the virtual model that reinforcement learning develops for a given environment.

4.1 Epsilon Greedy Policy:

This is a fundamental policy in reinforcement learning, which addresses the exploration and exploitation trade-off. For every new action, the agent decides to make new actions or exploit the current best action. This decision parameter is called epsilon. Exploration means choosing a random action. Exploitation means choosing the current best-known action.

4.2 Q-learning:

The main aim of q-learning is to calculate the ideal q value by using the Bellman's equation. For this a matrix will be created and all the q-values will be inserted. They will be updated for every interval using the formula.

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

The Q-learning iteration

Q(s,a) = q-table value at current state, action.

Gamma = discount factor

R = Reward

From the above figure, learned rate is clearly shown.

where α is the learning rate—a crucial hyperparameter that must be adjusted because it governs convergence.

The next stage is to start putting the Q-Learning algorithm into practice. But it's crucial to look at how exploration and exploitation are traded off. This makes sense because the agent didn't know anything about the environment at first. In the lack of information, there is a tendency to learn new things instead of using what is already known. As the agent learns more about the surroundings, the emphasis gradually moves from learning new things to applying what it has previously learned. If this critical stage is skipped, the Q-value function could converge to a local minimum that is usually far from the optimal Q-value function.

In order to solve this problem, a threshold will be set and will decrease after each episode using the exponential decay algorithm. With this method, a variable is evenly sampled over [0,1] at each time step 't'. The agent will investigate its surroundings if the variable is smaller than the threshold; if not, it will make use of the information it already has.

4.3 Implementation:

All the necessary libraries are imported. Numpy for numerical python, Gymnasium for the frozen lake environment, matplotlib for plots.

```
import numpy as np
import gymnasium as gym
import random
from matplotlib import pyplot as plt
from gymnasium.envs.toy_text.frozen_lake import generate_random_map
```

Initializing the environment with the map size of 8 with render mode set to human which means the execution will open in a pygame.

```
# Goal is to reach to the end node
env = gym.make('FrozenLake-v1',render_mode="human",is_slippery=False,map_name = "8x8" )

#Initialize the environment and display the current state in pygame window
observation, info = env.reset()
```

```
#Examining the environment
print ("Action Space of the cart pole is:" , env.action_space)
print ("Observation Space of the cart pole is:",env.observation_space)
```

```
Action Space of the cart pole is: Discrete(4)
Observation Space of the cart pole is: Discrete(64)
```

Let us test the agent movement and confirm our working environment by making some random movements.

Let us test the environment and agent ¶

```
for _ in range(15):
    action = env.action_space.sample()
    observation, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        env.reset()
```

```
env.close()
```

Initialize the q-table and environment with zeroes.

Training will bee done using q-learning algorithm, this is a reinforcement learning algorithm

```
#Intitalizing q-table with zeroes
q_table = np.zeros([env.observation_space.n, env.action_space.n])
```

```
#Making Frozen lake environment with 8*8 size
env = gym.make('FrozenLake-v1',is_slippery=False,map_name = "8x8")
```

```
print("No of rows:",len(q_table)) #rows
print("Nuber of columns",len(q_table[0])) #Columns
```

```
No of rows: 64
Nuber of columns 4
```

Set all the parameters to maximize the rewards, parameters are learning rate, discount factor, exploration, and exploitation trade off.

```
alpha = 0.9  #learning rate
gamma = 0.9  #discount factor
epsilon = 1  #exploration-exploitation trade-off
number_of_episodes = 20000
```

For training the agent, the current state is recorded, and the program is run until 20,000 episodes. The q-table is updated for every iteration increasing the cumulative rewards.

```
rewards_per_episode = []
for i in range(number_of_episodes+1):

    observation , info = env.reset()

    terminated = False
    truncated = False

    cumulative_reward = 0

    while not terminated and not truncated:

        #choosing action
        if ( random.uniform(0,1) < epsilon ):
            action = env.action_space.sample()
        else:
            action = np.argmax(q_table[observation])

        # choosing an action
        new_observation,reward,terminated,truncated,info = env.step(action)

        cumulative_reward += reward

        # updating q-table
        old_q_value = q_table[observation, action]
        next_max = np.max(q_table[new_observation])

        new_q_value = (1 - alpha) * old_q_value + alpha * (reward + gamma * next_max)

        q_table[observation, action] = new_q_value

        observation = new_observation

    rewards_per_episode.append(reward if i<13000 else 1)

    if i % 1000 == 0:
        print(f"Episode #: {i}")
```

If the agent is tested with updated q-table values, the agent reaches the goal in a short amount of time.

5. Graph Algorithms:

Graph is a group of nodes that are connected with the edges. Graphs are used in wide variety range of applications nowadays. Some of the applications are GPS, web pages and social media connections. In this artefact, two popular algorithms DFS and Dijkstra are used.

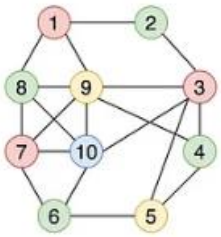


Figure 3: Graph

5.1 Depth First Search:

DFS is a tree/graph data structure, this starts at initial node and starts traversing the graph until the reach of goal or reaching end. A graph traversal technique called Depth-First Search (DFS) is used to methodically investigate and travel across nodes. It begins at a selected node, travels as far as feasible down each branch, and then traverses back. This method of exploration is similar to following a path through a maze until it comes to a stop, at which point you go back and attempt different paths.

To record visited nodes and exploration order, DFS keeps track of them on a stack, also known as a recursion call stack. The procedure is recursive in nature, visiting a node, designating it as visited, and then investigating its neighbours in turn. Until all reachable nodes are visited, the procedure keeps going.

DFS's capacity to explore a large portion of the graph before traversing back is one of its primary features, which might be useful.

Steps:

1. Initialize the stack as a list.
2. Select the start node and start appending it to the stack.
3. Continue traversing, if a non-visited node is found, push it to the top of the stack.
4. Repeat this and the last step until no nodes left to visit for the top of stack.
5. If all the nodes are visited pop the last element from the list.
6. Repeat the 2,3 and 4 steps until there is no element left in the stack.

5.2 Implementation:

The environment is created with a random generated map size of 8 setting render mode to human. The initial state is stored by using reset.

```
env = gym.make('FrozenLake-v1',is_slippery=False,render_mode = "human", desc = generate_random_map(size=8))
state,info = env.reset()
```

For performing DFS or Dijkstra algorithm, a graph structure is needed. Matrix also can be used as a graph, wrapper attribute desc is used to visualize the picture in matrix form. Then it is changed to string format using python.

```
env_desc = env.get_wrapper_attr('desc')
```

```
env_desc
```

```
array([[b'S', b'F', b'F', b'F', b'H', b'F', b'H', b'F'],
       [b'F', b'F', b'F', b'F', b'F', b'F', b'H', b'H'],
       [b'F', b'H', b'F', b'F', b'F', b'F', b'F', b'F'],
       [b'F', b'F', b'F', b'H', b'F', b'F', b'F', b'F'],
       [b'F', b'H', b'F', b'F', b'H', b'H', b'F', b'H'],
       [b'F', b'H', b'F', b'F', b'F', b'F', b'F', b'F'],
       [b'F', b'F', b'F', b'F', b'F', b'F', b'F', b'F'],
       [b'F', b'F', b'F', b'F', b'H', b'F', b'F', b'G']], dtype='|S1')
```

```
#converting env_desc to a matrix
env_matrix = []
for row in env_desc:
    temp = []
    for cell in row:
        temp.append(cell.decode('UTF-8'))
    env_matrix.append(temp)
```

```
env_matrix
```

```
[['S', 'F', 'F', 'F', 'H', 'F', 'H', 'F'],
 ['F', 'F', 'F', 'F', 'F', 'F', 'H', 'H'],
 ['F', 'H', 'F', 'F', 'F', 'F', 'F', 'F'],
 ['F', 'F', 'F', 'H', 'F', 'F', 'F', 'F'],
 ['F', 'H', 'F', 'F', 'H', 'H', 'F', 'H'],
 ['F', 'H', 'F', 'F', 'F', 'F', 'F', 'F'],
 ['F', 'F', 'F', 'F', 'F', 'F', 'F', 'F'],
 ['F', 'F', 'F', 'F', 'H', 'F', 'F', 'G']]
```

The DFS function is implemented with the algorithm as stated above iterating through all directions.

```python
def find_path(matrix):
    def dfs(x, y, path):
        if not (0 <= x < rows and 0 <= y < cols) or matrix[x][y] == 'H' or visited[x][y]:
            return False

        #adding the current path
        path.append((x, y))
        visited[x][y] = True

        #returning if goal node is found
        if matrix[x][y] == 'G':
            return True

        # traversing four directions
        if (dfs(x + 1, y, path) or
            dfs(x - 1, y, path) or
            dfs(x, y + 1, path) or
            dfs(x, y - 1, path)):
            return True

        # If path not found, backtrack
        path.pop()
        return False

    rows = len(matrix)
    cols = len(matrix[0])
    visited = [[False] * cols for _ in range(rows)]

    start = None
    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 'S':
                start = (i, j)
                break

    path = []
    if dfs(start[0], start[1], path):
        return path
    else:
        return None
```

This algorithm gives output of the path if it exists. The output will be in the format of matrix co-ordinates.

```python
# printing the path
result = find_path(env_matrix)
if result:
    print("Path found:", result)
else:
    print("Path not found.")
```

```
Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (7, 1), (6, 1), (6, 2), (7, 2), (7, 3), (6, 3),
(5, 3), (4, 3), (4, 2), (3, 2), (2, 2), (1, 2), (0, 2), (0, 3), (1, 3), (2, 3), (2, 4), (3, 4), (3, 5), (2, 5), (2, 6), (3, 6),
(4, 6), (5, 6), (6, 6), (7, 6), (7, 7)]
```

For frozen lake there is a set of actions defined. To visualize the same in the pygame, mapping should be done accordingly. A mapping function is implemented and converted to actions.

Action List

0: Move left

1: Move down

2: Move right

3: Move up

```
#mapping path actions according to agent
def map_path_to_actions(path):
    actions = []
    for i in range(1, len(path)):
        current_x, current_y = path[i - 1]
        next_x, next_y = path[i]

        if current_x < next_x:
            actions.append(1)  # Move down
        elif current_x > next_x:
            actions.append(3)  # Move up
        elif current_y < next_y:
            actions.append(2)  # Move right
        elif current_y > next_y:
            actions.append(0)  # Move left

    return actions
```

```
actions = map_path_to_actions(result)
print(actions)
```

```
[1, 1, 1, 1, 1, 1, 1, 2, 3, 2, 1, 2, 3, 3, 3, 0, 3, 3, 3, 3, 2, 1, 1, 2, 1, 2, 3, 2, 1, 1, 1, 1, 1, 2]
```

Action List

0: Move left

1: Move down

2: Move right

3: Move up

```
#mapping path actions according to agent
def map_path_to_actions(path):
    actions = []
    for i in range(1, len(path)):
        current_x, current_y = path[i - 1]
        next_x, next_y = path[i]

        if current_x < next_x:
            actions.append(1)  # Move down
        elif current_x > next_x:
            actions.append(3)  # Move up
        elif current_y < next_y:
            actions.append(2)  # Move right
        elif current_y > next_y:
            actions.append(0)  # Move left

    return actions
```

```
actions = map_path_to_actions(result)
print(actions)
```

```
[1, 1, 1, 1, 1, 1, 1, 2, 3, 2, 1, 2, 3, 3, 3, 0, 3, 3, 3, 3, 2, 1, 1, 2, 1, 2, 3, 2, 1, 1, 1, 1, 1, 2]
```

The same can be used to test the environment and this path reaches the goal node.

Testing the found path for agent

```
for action in actions:
    env.step(action)
```

```
env.close()
```

5.3 Dijkstra Algorithm

Dijkstra algorithm is also known as shortest path finding algorithm. It calculates the shortest path from start node to all nodes considering all the possibilities, this ensures that the goal is reached.

In a graph with non-negative edge weights, the shortest path between nodes can be found using Dijkstra's algorithm. Dijkstra's algorithm looks for the most efficient path in terms of cumulative weights, in contrast to Depth-First Search, which traverses a graph without taking into account the edge weights. Without going into particular algorithmic details, here's an explanation:

Beginning at a source node, Dijkstra's algorithm methodically investigates nearby nodes, choosing the path with the lowest cumulative weight at each stage. To effectively select the subsequent node to investigate, it keeps track of a priority queue or a comparable data structure. Dijkstra's algorithm begins by giving nodes approximate distances and updates them gradually with the smallest distances that can be discovered while exploring.

By the time the algorithm finishes running, it makes sure that it has discovered the shortest path between each source node and every other node in the graph. Dijkstra's algorithm is especially helpful in situations like network routing and logistics, when choosing the best course is essential.

Finding the shortest path is guaranteed by Dijkstra's algorithm, which is capable of handling graphs with non-negative weights. It might not work as well, though, in graphs with negative cycles or weights.

Steps:

1. Initialize a queue and set data structures.
2. Start from a node, add the current node to the queue.
3. Choose the shortest distance node and add it to the visited set.
4. Update the values of adjacent nodes for every traversal.
5. If dist(v)+weight(u,v) < dist(u), if new distance is found updated it the least value.
6. Else don’t change the values.

5.4 Implementation:

The initial steps for DFS and Dijkstra are similar up to some extent. The environment creation, converting the visual into a matrix.

```
env = gym.make('FrozenLake-v1',is_slippery=False,render_mode = "human", desc = generate_random_map(size=8))
state,info = env.reset()
```

```
env_desc = env.get_wrapper_attr('desc')
```

```
env_desc
```

```
array([[b'S', b'F', b'F', b'F', b'F', b'F', b'F', b'H'],
       [b'H', b'F', b'H', b'F', b'F', b'F', b'F', b'F'],
       [b'F', b'F', b'H', b'F', b'F', b'F', b'F', b'F'],
       [b'F', b'F', b'F', b'F', b'H', b'F', b'H', b'F'],
       [b'F', b'H', b'F', b'F', b'H', b'F', b'F', b'F'],
       [b'F', b'F', b'F', b'F', b'F', b'F', b'F', b'H'],
       [b'F', b'F', b'F', b'H', b'F', b'F', b'F', b'H'],
       [b'F', b'F', b'F', b'F', b'F', b'F', b'F', b'G']], dtype='|S1')
```

converting env_desc into matrix form

```
env_matrix = []
for row in env_desc:
    temp = []
    for cell in row:
        temp.append(cell.decode('UTF-8'))
    env_matrix.append(temp)
```

```
env_matrix
```

```
[['S', 'F', 'F', 'F', 'F', 'F', 'F', 'H'],
 ['H', 'F', 'H', 'F', 'F', 'F', 'F', 'F'],
```

The algorithm for the Dijkstra is implemented for the frozen lake.

```
import heapq

def dijkstra(matrix):
    rows, cols = len(matrix), len(matrix[0])
    start, goal = None, None

    # To find start and goal coordinates
    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 'S':
                start = (i, j)
            elif matrix[i][j] == 'G':
                goal = (i, j)

    if not start or not goal:
        raise ValueError("Start or goal not found in the matrix.")

    # Actions (left, down, right, up)
    movements = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    # Initialize distance matrix with infinite values
    distances = [[float('inf')] * cols for _ in range(rows)]

    # Priority queue to store (distance, vertex) pairs
    priority_queue = [(0, start)]
    distances[start[0]][start[1]] = 0

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # If the current vertex is the goal, stop the algorithm
        if current_vertex == goal:
            break

        for movement in movements:
            dx, dy = movement
            new_x, new_y = current_vertex[0] + dx, current_vertex[1] + dy

            # if the new coordinates are within the matrix boundaries
            if 0 <= new_x < rows and 0 <= new_y < cols and matrix[new_x][new_y] != 'H':
                new_distance = current_distance + 1

                # Update the distance if the new path is shorter
                if new_distance < distances[new_x][new_y]:
                    distances[new_x][new_y] = new_distance
                    heapq.heappush(priority_queue, (new_distance, (new_x, new_y)))
```

Reconstructing the path

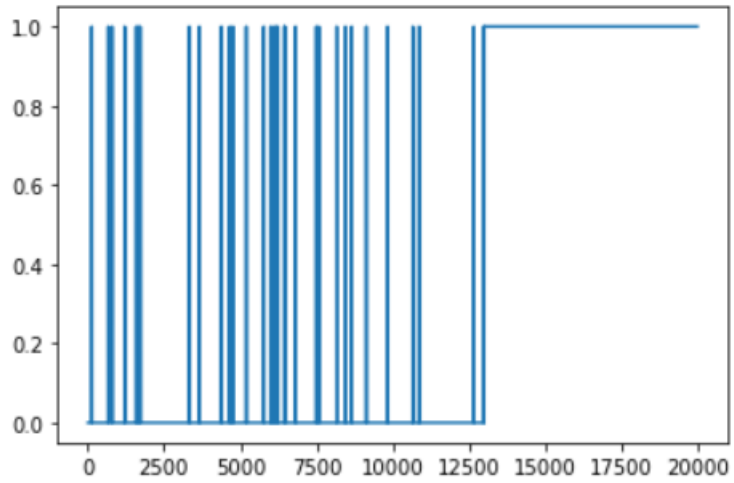
```
# Reconstructing path
path = []
current_vertex = goal
while current_vertex != start:
    path.append(current_vertex)
    for movement in movements:
        dx, dy = movement
        new_x, new_y = current_vertex[0] + dx, current_vertex[1] + dy
        if 0 <= new_x < rows and 0 <= new_y < cols and distances[new_x][new_y] == distances[current_vertex[0]][current_verte
            current_vertex = (new_x, new_y)
            break

path.append(start)
path.reverse()

return path
```

After mapping all the actions, the agent finds the path with minimum number of steps.

6. Results and Discussions

6.1 Reinforcement Learning

Reinforce learning finds the shortest path after iterating over thousands of episodes, the accuracy and cumulative rewards increase gradually and will give the best output once the training completes.



If the results are carefully observed, in the start the target reaching the goal is very less, after 5000 episodes the agent reaching the target is very frequent. Once the training episodes cross 13,000 the reward is always one, because the agent always finds the goal node.

6.2 DFS and Dijkstra

For evaluating Depth First Search and Dijkstra algorithms, path length and time complexities are used. This is similar to the space and time complexities. Firstly, the algorithms are implemented to test the working and getting desired results. To evaluate the complexity will be increased consistently and check the time complexities and path length.

- For first level, let the map size be 12, the results are as follows:

Algorithm	Path Length	Time(ms)
DFS	62	1.00
Dijkstra	22	1.99

In the first test, the path length of DFS is 62 and time taken by DFS is 1ms. For Dijkstra, the path length is 22 and time taken is 1.99 ms. Time is almost same, but the path length has high difference, Dijkstra is more efficient.

- For the second level, increase the random map size to 25.

Algorithm	Path Length	Time(ms)
DFS	144	1.00
Dijkstra	48	2.00

In the second test, DFS has 144 path length whereas Dijkstra completes the path in just 48 steps. The time is almost similar, Dijkstra is more efficient in this level as well.

- For the third time, increase the map size to 50.

Algorithm	Path Length	Time(ms)
DFS	504	1.00
Dijkstra	98	8.50

In the third test, Dijkstra outperforms DFS with the path length difference of almost 400 steps. The time taken by Dijkstra is slightly higher than DFS in this case as compared to other tests.

6.3 Discussion

Reinforce learning is a very good AI algorithm, it can be used to many daily life applications such as games like chess and checkers. Reinforce learning can be used for the problems that have very less dataset or less actions. Although it takes more time to train, the accuracy is vastly increased once the training completed. AI chess games beats humans who created the games. Reinforcement learning and graph algorithm analysis does not fair to compare each other as the DFS and Dijkstra algorithms works instantaneously while the RL takes longer time.

The DFS algorithm finds the path, it does its work, but the path length is very large as the size of the map increases. The time that took for DFS is less. On the contrary, the Dijkstra algorithm takes a bit longer than DFS and path length is very less compared to DFS. Practically considering, the amount of time taken for the moves will be larger. So, the Dijkstra is way better than DFS for complex problems.

In general, the Dijkstra should perform the analysis and find the path in less time. In the frozen lake, the graph is not weighted and not directional. If all these complexities are added to the frozen lake, then the Dijkstra would perform a lot better than DFS both in terms of space and time complexities.

7. Commercial and Potential Risks:

There are various potential risks for this to implement in real life. Both DFS and Dijkstra algorithms have different computational complexities, if these algorithms are computationally expensive then it is hard to implement them in the real world or resource constrained environments. Reinforcement training heavily depends on the quality of the instructions and dataset. If the training is biased or incomplete, the RL results in suboptimal accuracy and decision making.

The parameter tuning in reinforcement learning is very important, they need to be completely tested before using them in real life, this poor hyper parameter tuning results in suboptimal policies. The reinforcement learning performs very well in environments they have already trained on. The only concern is its uncertainty to perform in unknown or new environments that have a lot of changes from the original environment.

There are also certain commercial risks associated with this environment. Some of them will be stated in this. The implementation and maintenance costs can use a lot of resources, significantly more computational resources are required. The integration of these algorithms into the existing systems may have different issues like compatibility issues, data formatting issues, data discrepancies and additional infrastructure. There may be regulatory and ethical issues for using these algorithms in certain areas.

The competition will be very high as the AI technology is trending technology and lot of people are researching about this topic. Also, risks like competitors using more advanced algorithms making this model difficult. In these critical applications, security is very important. Maximum security should be implemented protecting from external exploits.

It is very important to assess all these problems and upgrade the algorithms accordingly. This enhances the application usage in the daily life. Regular monitoring and staying up to date in this current world are necessary for long term success.

8. Conclusion

To conclude, frozen lake environment is solved using Reinforcement learning specifically q-learning, Depth First Search and Dijkstra. The agent learns to navigate by taking random decisions, updating the epsilon greedy policy. Meanwhile, DFS explores the path using stack and Dijkstra reaches the goal by following the queue data structure and minimum distance. The better graph algorithm is Dijkstra as it reaches the goal with the minimum path length, on the other hand DFS find the path but it may not be the optimal path. These results give important insights for applying these algorithms in diverse scenarios.

9. Personal Reflection

My reflection on Frozen Lake using AI project is initially I thought this module is very tough and will take a lot of time to implement. But, in the end this was not as expected it really was easy to implement by finding all the resources and with correct guidance. I have attended all the labs and classes, with the close interaction between tutors, I got to find resources where I can learn. Once all the learning is complete, it took me less time than expected.

In my opinion, I did very good because I implemented both graph and reinforcement algorithms with a single environment. I organized all the code clearly, by commenting and clear markdown comments in jupyter notebook. All the results are clearly explained and discussed the results.

To improve myself, I would like to focus more on doing work at once instead of revisiting the same task again and again for changes or improvements. I will improve my planning and organizing my ICA tasks for better time management. This was my personal reflection journey for this project.

10. References:

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. New Jersey: Pearson.

Kahneman, D. (2011). *Thinking, Fast and Slow*. New York: Farrar, Straus and Giroux.

Dr. Basant Agarwal (2022). *Hands-On Data Structures and Algorithms with Python*. Packt Publishing Ltd.

Bajaj, P. (2018). *Reinforcement learning - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-reinforcement-learning/.

Amine, A. (2020). *Q-Learning Algorithm: From Explanation to Implementation*. [online] Medium. Available at: https://towardsdatascience.com/q-learning-algorithm-from-explanation-to-implementation-cdbeda2ea187.

Mallawaarachchi, V. (2020). *10 Graph Algorithms Visually Explained*. [online] Medium. Available at: https://towardsdatascience.com/10-graph-algorithms-visually-explained-e57faa1336f3.

Abiy, T., Pang, H. and Williams, C. (2016). *Dijkstra's Shortest Path Algorithm | Brilliant Math & Science Wiki*. [online] Brilliant.org. Available at: https://brilliant.org/wiki/dijkstras-short-path-finder/.

Javatpoint(n.d.). *DFS Algorithm - javatpoint*. [online] Available at: https://www.javatpoint.com/depth-first-search-algorithm.