Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A subsidiary organ of OIC

Laboratory Report

CSE 4618: Artificial Intelligence Lab

Name	**: Mirza Mohammad Azwad**
Student ID	**: 200042121**
Section	**: BSc in SWE**
Semester	**: 6th Semester**
Academic Year	**: 2022-23**
Date	**: 31/01/2024**
Lab No	**: 1**

Contents

Introduction

This lab primarily deals with the idea of uninformed search: **depth-first search, breadth-first search** and **uniform cost search**. The implementation details and the data structures used were provided as lab resources with a *utils.py* file that contained the data structure implementations for stack, queue and priority queue.

Using these we could implement the different algorithms in question. The idea is pretty simple and can be mapped from preliminary graph theory or algorithm courses which involve dfs, bfs and dijkstra.

We had to manipulate the search.py file and the methods present within that file to manipulate the methods to show the required behavior.

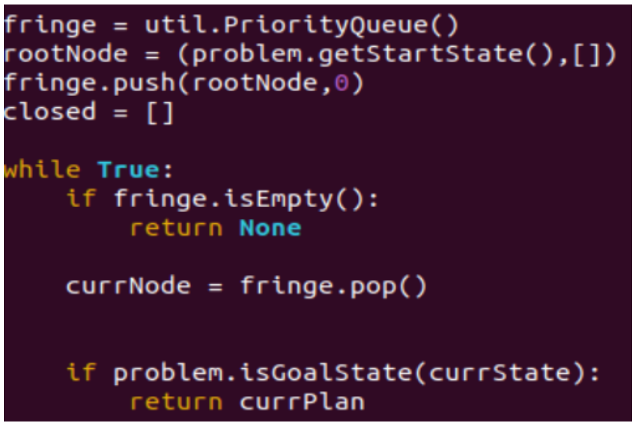
In this lab, we used our previous **conda** environment to work in the required **python 3.6** environment as required by the given files.

Analysis of the Problems

1. **Depth-First Search**: **Depth First Search** implementation using a stack as a fringe, the idea is for pacman to reach the food packet using depth-first search, this does not produce the optimal path as it searches for all possible paths and extends to the depth using the leftmost branch(depends on implementation). This leads to a possible solution but not necessarily an optimal solution.
2. **Breadth-First Search: Breadth First Search** implementation using a queue as a fringe, the idea is the same as above but in this case it selects the best possible path without any heuristics. The paths have no assigned costs and selects the best possible path considering all weights to be unit weights. So pac man goes to find the packet considering the path to have no costs optimally.
3. **Uniform Cost Search: Uniform Cost Search** implementation using a priority queue as a fringe, the idea while its same as the above examples. It selects the best possible path while considering the cost of the path which may have different costs based on difficulty of going to the packet. So pac man goes to the packet while considering the best path while considering the difficulty of going there as cost of the path.

Explanation of Solution

Depth First Search: The solution was given as part of the instruction. This involved a *closed* list which contains all the states already traversed or in other words 'visited'. Then we also used a stack as a fringe from the utils.py which had the implemented **stack** data structure. The *closed* list ensures that the same state isn't visited multiple times, which may lead to unnecessary computation or getting stuck in a cycle leading to an infinite loop. We visit all the states as per the plan and avoid states in the closed list. We iterate over the successors of each state which is present in the fringe. As per this implementation, **Depth First Search** seems to proceed in a pre-order traversal manner expanding the leftmost branch not visited first.
Breadth First Search: The solution was just changing the fringe from a stack to a **queue** which helps do a level-order traversal to go across each node in the same level or the successors of the same parent. The other aspects of the solution remain the same as **Depth First Search**.
Uniform Cost Search: In this case, we change the fringe from a queue to a priority queue, with the cost being the cost to go from one node to the next. The priority queue in this case is a min heap. It was also retrieved from utils.py and the queue essentially fetches the state within the priority queue with the minimum cost first. This is essentially Dijkstra's shortest path algorithm where we don't repeat the states in the closed state and we move as per the weights that are minimal allowing us to essentially go along the shortest path from a given node, hence it is a single source shortest path algorithm.

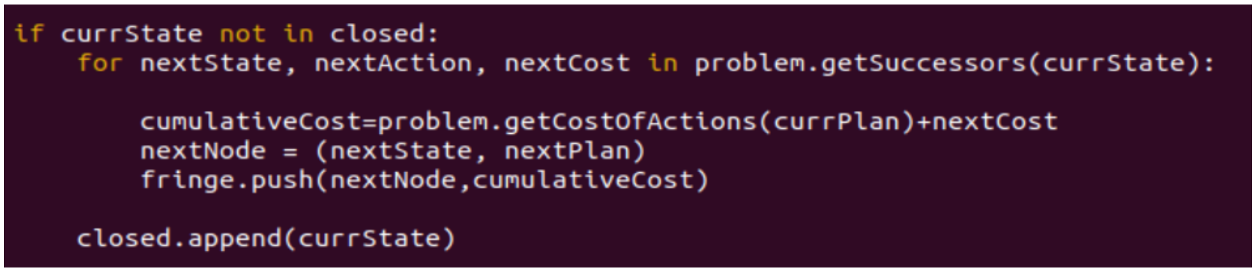


```
fringe = util.PriorityQueue()
rootNode = (problem.getStartState(),[])
fringe.push(rootNode,0)
closed = []

while True:
    if fringe.isEmpty():
        return None

    currNode = fringe.pop()


    if problem.isGoalState(currState):
        return currPlan
```



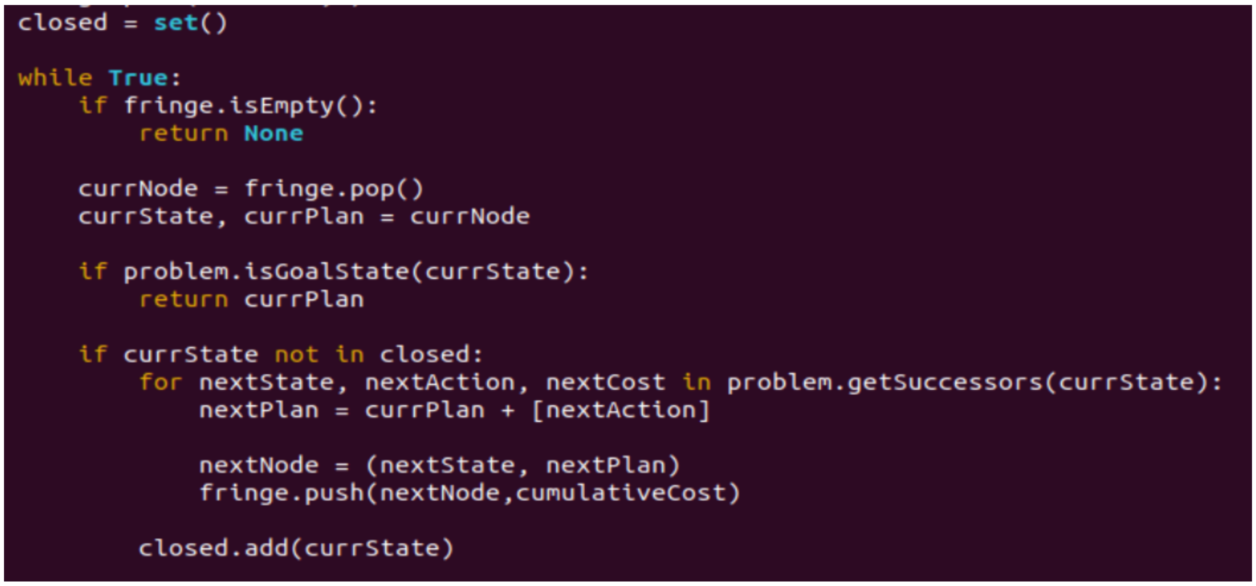
```
if currState not in closed:
    for nextState, nextAction, nextCost in problem.getSuccessors(currState):

        cumulativeCost=problem.getCostOfActions(currPlan)+nextCost
        nextNode = (nextState, nextPlan)
        fringe.push(nextNode,cumulativeCost)

    closed.append(currState)
```

Problems Faced and Interesting Findings

An optimization that we can perform is storing a closed set using a set instead of a list; it seems to improve the performance. And it was recommended theoretically for optimization reasons.



```
closed = set()

while True:
    if fringe.isEmpty():
        return None

    currNode = fringe.pop()
    currState, currPlan = currNode

    if problem.isGoalState(currState):
        return currPlan

    if currState not in closed:
        for nextState, nextAction, nextCost in problem.getSuccessors(currState):
            nextPlan = currPlan + [nextAction]

            nextNode = (nextState, nextPlan)
            fringe.push(nextNode,cumulativeCost)

        closed.add(currState)
```

Since a set is implemented using hash tables checking whether or not an element is in the set can be checked in **O(1)** but for lists it has time complexity of **O(n)** which makes the set implementation more optimal. While the set implementation in this case is shown for the uniform cost search, it can be applied accordingly for bfs and dfs too.