VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLIGENCE

IT159IU

REPORT LAB 2

Instructor:

Dr. Nguyen Trung Ky

Dr. Ly Tu Nga

Nguyen Huynh Ngan Anh - ITDSIU23003

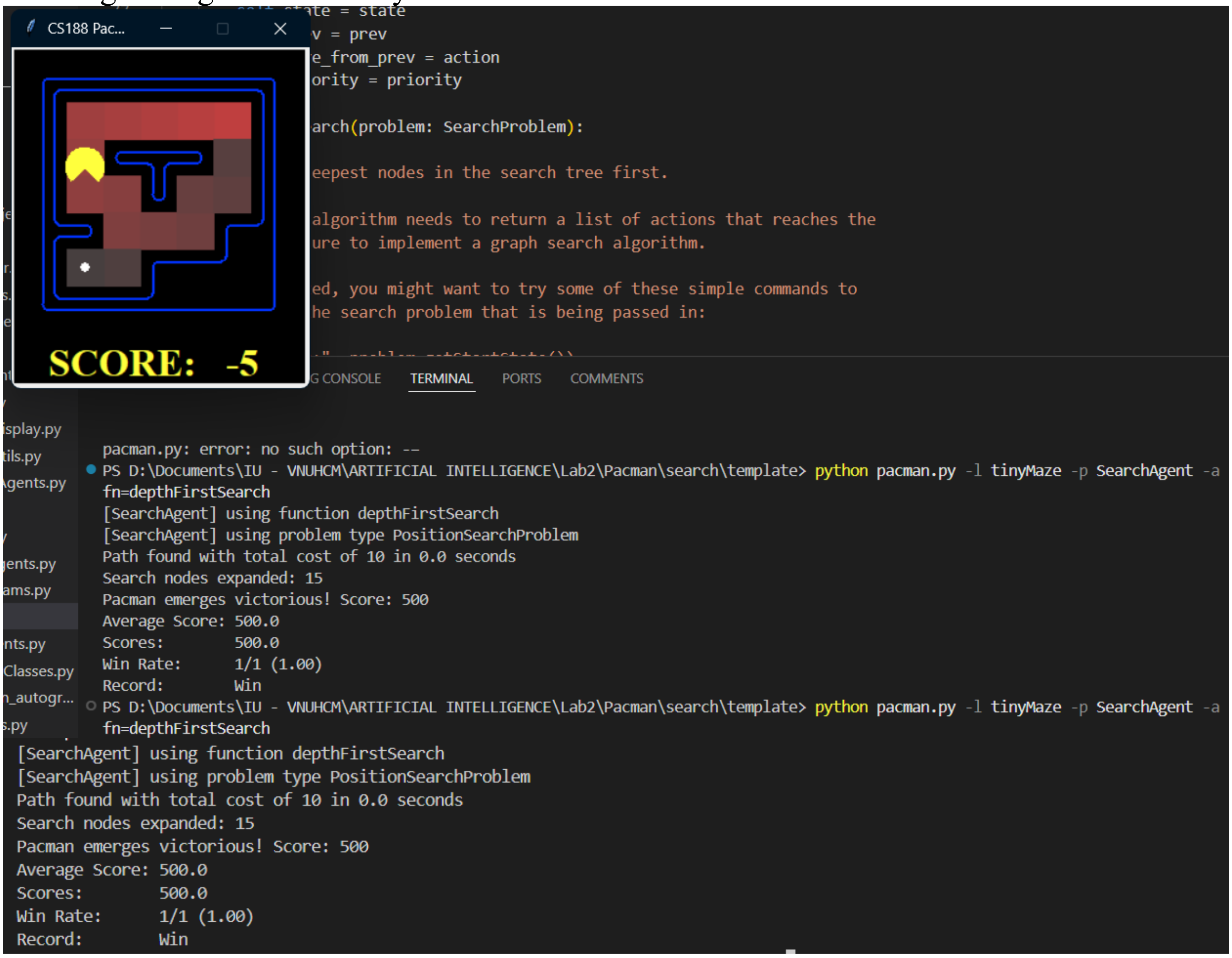
0. Setup:

Tried running the below command to run the game

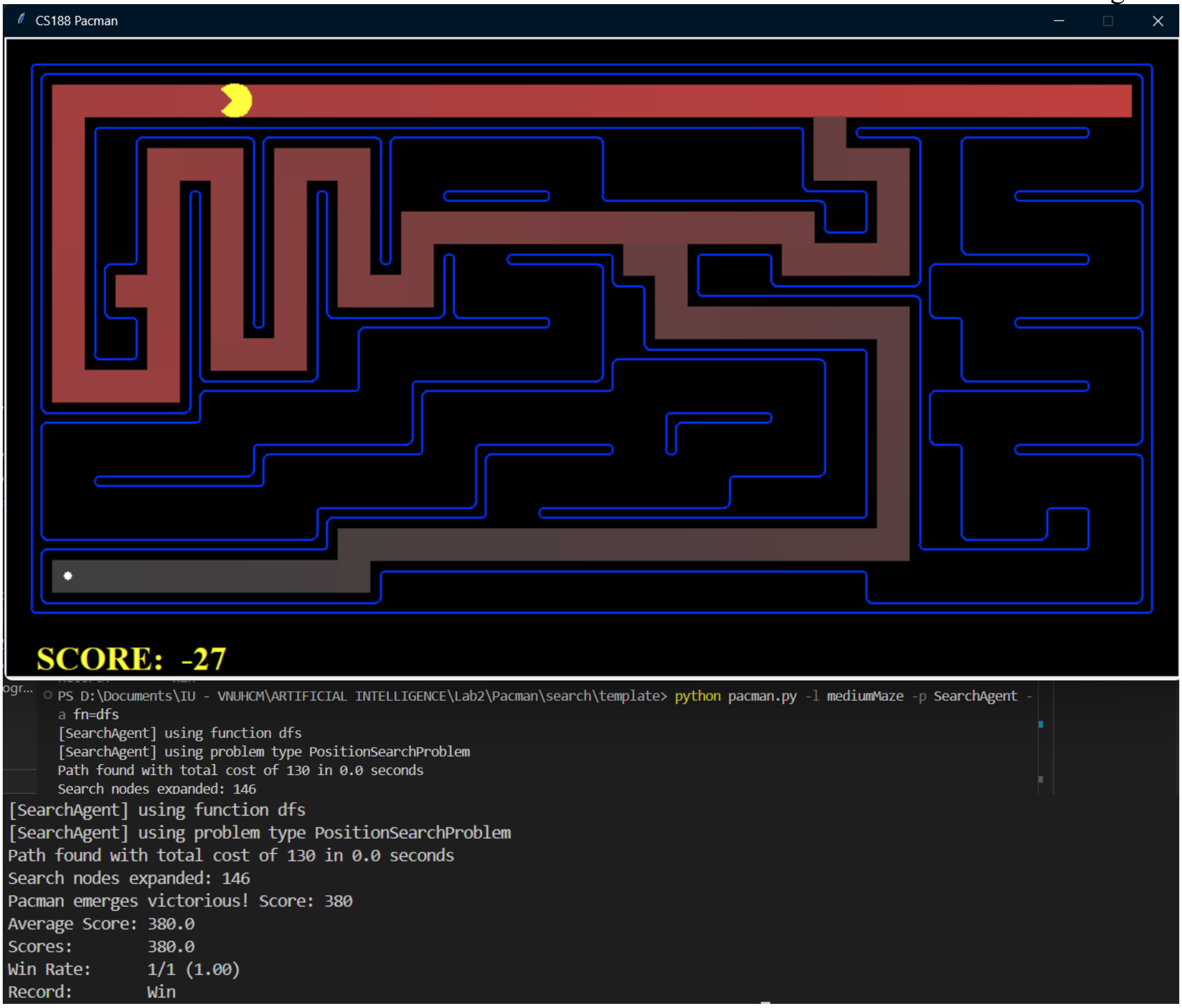
```
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
[SearchAgent] using function tinyMazeSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 0
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

1. Exercise 1:

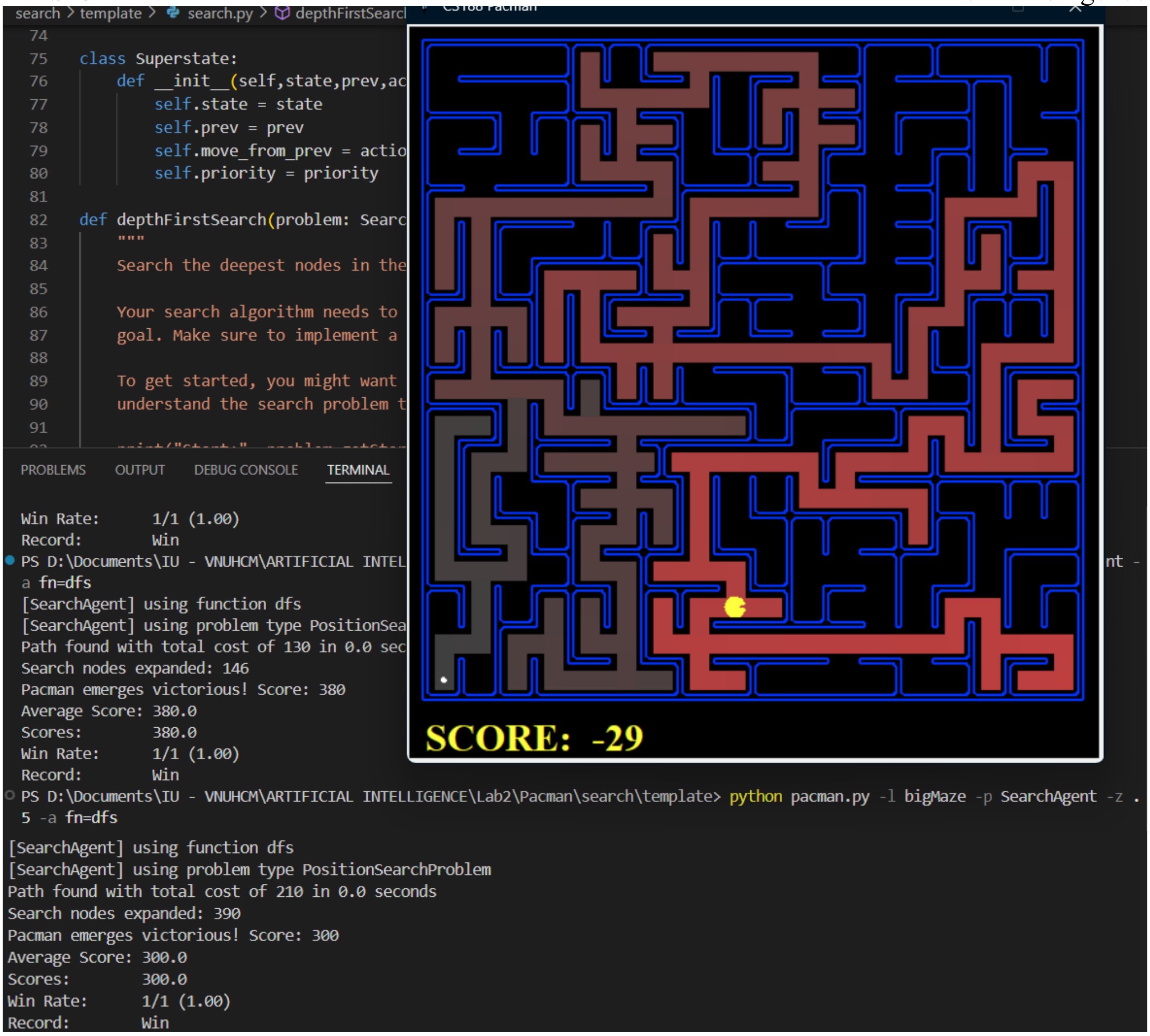
Running dfs algorithm for tinyMaze



Running dfs algorithm for mediumMaze



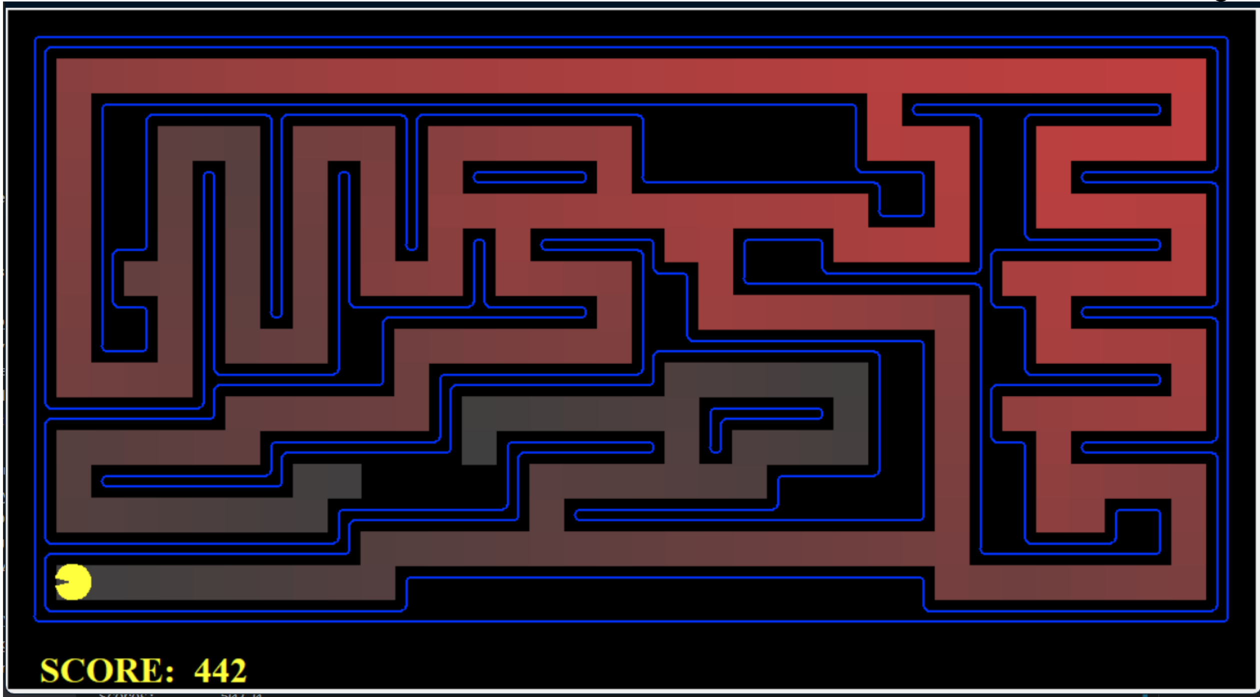
Running dfs algorithm for bigMaze



2. Exercise 2:

Running bfs algorithm for tinyMaze, mediumMaze and bigMaze

```
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```




```
Win Rate:      1/1 (1.00)
Record:        Win
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l mediumMaze -p SearchAgent -
a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
```

```
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l bigMaze -p SearchAgent -z .
5 -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3. Exercise 3:

Running ucs algorithm for tinyMaze, mediumMaze and bigMaze

```
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l mediumMaze -p SearchAgent -
a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l tinyMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l bigMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\Lab2\Pacman\search\template> python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

4. Conclusion

	Depth-First Search			**Breadth-First Search**			**Uniform-Cost Search**		
Maze	**#nodes explored**	**Solution length**	**Is it optimal?**	**#nodes explored**	**Solution length**	**Is it optimal?**	**#nodes explored**	**Solution length**	**Is it optimal?**
tiny	15	10	No	15	8	Yes	15	8	Yes
medium	146	130	No	269	68	Yes	269	68	Yes
big	390	210	No	620	210	Yes	620	210	Yes

The DFS search algorithm begins at the root node and proceeds as far as it can go along each branch before turning around. It employs a stack data structure to keep track of the nodes that need to be visited. DFS is easy to use and can be used to resolve issues that call for a scan of the complete network, but it might not always find the best answer and might become trapped in an endless cycle.
The BFS begins at the root node and examines every node at the current depth before going on to the following. It employs a queue data structure to maintain an account of the locations that need to be viewed. BFS is helpful when the answer is near the root node and is assured to identify the quickest route between two nodes in an unweighted network.
By reducing the route's overall cost while considering the cost of each edge in the network, the UCS search method determines the optimal path. It uses a priority list data structure to

maintain an account of the sites that need to be viewed. Although UCS will always find the best answer, it may take longer than other methods, particularly in extensive networks with many links.

DFS and BFS have an O(V + E) time complexity and an O(V + E) volume complexity, in which V denotes the number of nodes and E represents the number of edges in the network. On the other hand, UCS uses a priority list and has a temporal complexity of O((V + E) log V). DFS and BFS require O(V) space, while UCS needs O(V Plus E) space in terms of space complexity.

In conclusion, DFS, BFS, and UCS are all helpful search algorithms, each with unique benefits and drawbacks. Although DFS is easy to use, it may not always yield the best result. BFS ensures the quickest route in unweighted graphs, but it may take longer in extensive networks. In big charts, UCS may be slower than other methods but provides the best outcome. The particular issue being addressed and the properties of the network determine which method should be used.

I try to visualize the searching algorithm to under the form of tree node to better understand how algorithms is optimal to each other

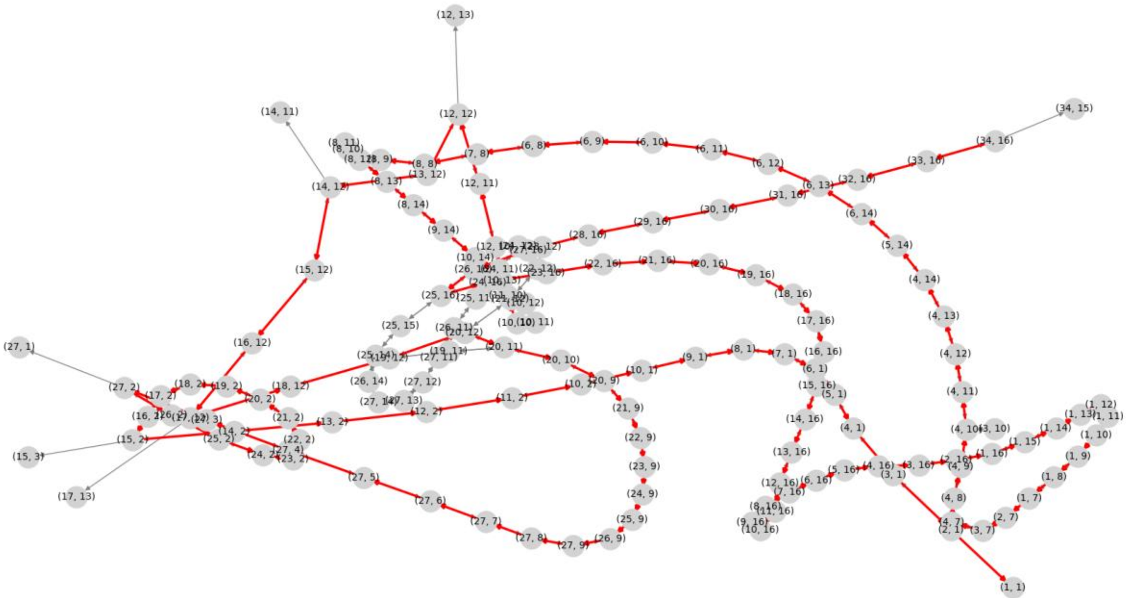
Lab2 > Pacman > search > template > search.py > uniformCostSearch

```python

def visualize_paths(graph, all_paths, optimal_path):
    pos = nx.spring_layout(graph)

    # Draw the full graph
    nx.draw(graph, pos, with_labels=True, node_color='lightgray', edge_color='gray', node_size=500, font_size=10)

    # Draw all paths in blue
    for path in all_paths:
        edges = [(path[i], path[i+1]) for i in range(len(path)-1)]
        nx.draw_networkx_edges(graph, pos, edgelist=edges, edge_color='blue', alpha=0.3, width=1)

    # Draw the optimal path in red
    optimal_edges = [(optimal_path[i], optimal_path[i+1]) for i in range(len(optimal_path)-1)]
    nx.draw_networkx_edges(graph, pos, edgelist=optimal_edges, edge_color='red', width=2.5)

    plt.title("All Paths (Blue) vs Optimal Path (Red)")
    plt.show()
```

And modify the searching algorithm to return the required value for visualization

```python
def depthFirstSearch(problem: SearchProblem):
    """
    Search the deepest nodes in the search tree first.
    Extract all paths explored and store them.
    """
    print("Start:", problem.getStartState())

    my_stack = util.Stack()
    init_state = problem.getStartState()
    init_superstate = Superstate(init_state, None, None)
    visited_list = []
    all_paths = []  # Store all paths
    my_stack.push((init_superstate, []))  # Stack stores (Superstate, Path taken)

    nodes_expanded = 0
    nodes = []
    edges = []
    costs = {}
    re_solution = []

    while not my_stack.isEmpty():
        current_superstate, current_path = my_stack.pop()
        nodes_expanded += 1
        nodes.append(current_superstate.state)
        costs[current_superstate.state] = current_superstate.priority

        new_path = current_path + [current_superstate.state]  # Update path

        if problem.isGoalState(current_superstate.state):
            all_paths.append(new_path)  # Store the entire path when goal is reached
            all_paths.append(new_path)  # Store the entire path when goal is reached
            visited_list.append(current_superstate)
            re_solution = new_path  # Use the path of states as the solution
            runner_superstate = current_superstate
            while runner_superstate.state != init_superstate.state:
                edges.append((runner_superstate.prev.state, runner_superstate.state))
                runner_superstate = runner_superstate.prev
            print(f"Nodes expanded: {nodes_expanded}")
            print(f"Nodes: {nodes}")
            print(f"Edges: {edges}")
            print(f"Costs: {costs}")
            print(f"Optimal path: {re_solution}")
            visualize_paths(nx.DiGraph(edges), all_paths, re_solution)  # Use visualize_paths to visualize the search tree
            return re_solution
        else:
            if current_superstate not in visited_list:
                visited_list.append(current_superstate)
                successors = problem.getSuccessors(current_superstate.state)
                for successor in successors:
                    successor_superstate = Superstate(successor[0], current_superstate, successor[1], successor[2])
                    edges.append((current_superstate.state, successor[0]))
                    # Ensure successor state is not already visited
                    if all(s.state != successor_superstate.state for s in visited_list):
                        my_stack.push((successor_superstate, new_path))  # Pass updated path

    print(f"Nodes expanded: {nodes_expanded}")
    print(f"Nodes: {nodes}")
    print(f"Edges: {edges}")
    print(f"Costs: {costs}")
    print(f"All Paths: {all_paths}")
    visualize_paths(nx.DiGraph(edges), all_paths, re_solution)  # Use visualize_paths to visualize the search tree if no solut
    return visited_list, nodes_expanded, nodes, edges, costs
```

I test with dfs but the visualize seems not working right, I'll try to modify it later


(12, 13)
(12, 12)
(14, 11)
(34, 15)
(34, 16)
(33, 16)
(32, 16)
(6, 13)
(6, 12)
(6, 11)
(6, 10)
(6, 9)
(6, 8)
(7, 8)
(8, 8)
(13, 12)
(8, 13)
(14, 12)
(12, 11)
(8, 14)
(9, 14)
(28, 16)
(29, 16)
(30, 16)
(31, 16)
(6, 14)
(5, 14)
(4, 14)
(4, 13)
(4, 12)
(4, 11)
(15, 12)
(16, 12)
(22, 16)
(21, 16)
(20, 16)
(19, 16)
(18, 16)
(17, 16)
(16, 16)
(25, 15)
(25, 16)
(27, 1)
(27, 2)
(18, 12)
(20, 11)
(20, 10)
(10, 1)
(9, 1)
(8, 1)
(7, 1)
(6, 1)
(5, 1)
(15, 16)
(14, 16)
(13, 16)
(12, 16)
(4, 1)
(26, 14)
(27, 12)
(21, 2)
(13, 2)
(12, 2)
(11, 2)
(10, 2)
(21, 9)
(22, 9)
(23, 9)
(24, 9)
(25, 9)
(26, 9)
(27, 9)
(27, 8)
(27, 7)
(27, 6)
(27, 5)
(15, 2)
(15, 3)
(17, 13)
(22, 2)
(23, 2)
(5, 16)
(4, 16)
(3, 16)
(3, 1)
(2, 1)
(1, 1)
(4, 8)
(4, 9)
(3, 7)
(2, 7)
(1, 7)
(1, 8)
(1, 9)
(1, 10)
(1, 11)
(1, 12)
(1, 14)
(1, 15)
(1, 16)
(3, 10)
(10, 16)