

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE  
NATIONAL TECHNICAL UNIVERSITY OF UKRAINE  
" IHORY SIKORSKY KYIV POLYTECHNIC INSTITUTE"

**Volodymyr Shymkovych**

## **Technologies of Artificial Intelligence**

**LABORATORY WORK #2**

**Uninformed search**

Kulubecioglu Mehmet

IM-14 FIOT

Kyiv  
IHORY SIKORSKY KYIV POLYTECHNIC INSTITUTE  
2024

## 1. Problem Definition

In this assignment, I implemented various uninformed search algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening Search (IDS), Depth-Limited Search (DLS), and Bidirectional Search—in Python. The objective was to apply these algorithms to a defined search problem, measure their performance in terms of execution time and the number of nodes visited, and compare their efficiencies.

## 2. Search Task Representation

To define the search task, I established the following components:

- **State Space (S):** All possible states the agent can occupy.
- **Initial State (si):** The starting point of the search.
- **Goal State (sg):** The target state the agent aims to reach.
- **Actions:** The possible moves or transitions the agent can perform from a given state.
- **Transition Model:** A function that defines the result of performing an action in a state, leading to a new state.
- **Goal Test:** A function that determines whether a given state is the goal state.

For this assignment, I used a simple graph-based representation where nodes represent states, and edges represent possible transitions between states.

## 3. Graph Representation

I modeled the environment as a graph using an adjacency list. Here's the example graph I used:

```
4  # graph (adjacency list representation)
5  graph = {
6      '1': ['2', '3'],
7      '2': ['4', '5'],
8      '3': ['6', '7'],
9      '4': ['8'],
10     '5': [],
11     '6': [],
12     '7': [],
13     '8': []
14 }
15
```

In this graph:

- Node '1' is the initial state.
- Node '8' is the goal state.
- The robot can move from one node to its connected nodes as defined in the adjacency list.

## 4. Implementation of Uninformed Search Algorithms

### Breadth-First Search (BFS)

BFS explores all nodes at the current depth before moving to the next level, ensuring the shortest path in unweighted graphs.

```
15
16 # Breadth-First Search (BFS)
17 def breadth_first_search(start, goal, graph):
18     visited = set()
19     queue = deque([start])
20     node_count = 0 # Counter for visited nodes
21
22     while queue:
23         node = queue.popleft()
24         node_count += 1 # Increment node count for each visited node
25
26         if node == goal:
27             return node_count
28
29         if node not in visited:
30             visited.add(node)
31             for neighbor in graph[node]:
32                 if neighbor not in visited:
33                     queue.append(neighbor)
34
35     return node_count
36
```

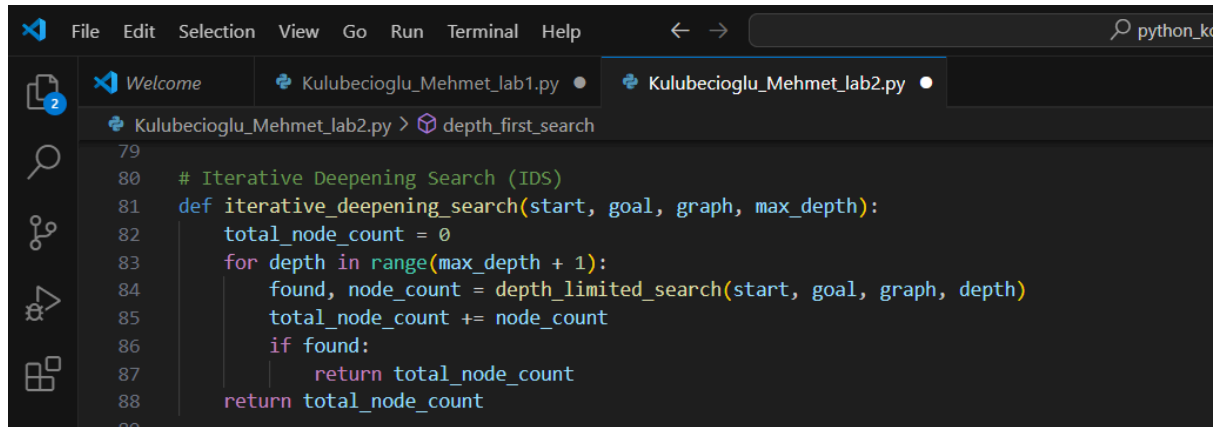
### Depth-First Search (DFS)

DFS explores as deeply as possible along each branch before backtracking. It is more memory-efficient but does not guarantee the shortest path.

```
File Edit Selection View Go Run Terminal Help
Welcome Kulubecioglu_Mehmet_lab1.py Kulubecioglu_Mehmet_lab2.py
Kulubecioglu_Mehmet_lab2.py > depth_first_search
36
37 # Depth-First Search (DFS)
38 def depth_first_search(start, goal, graph):
39     visited = set()
40     stack = [start]
41     node_count = 0 # Counter for visited nodes
42
43     while stack:
44         node = stack.pop()
45         node_count += 1 # Increment node count for each visited node
46
47         if node == goal:
48             return node_count
49
50         if node not in visited:
51             visited.add(node)
52             for neighbor in reversed(graph[node]): # Reverse to maintain order
53                 if neighbor not in visited:
54                     stack.append(neighbor)
55
56     return node_count
57
```

## Iterative Deepening Search (IDS)

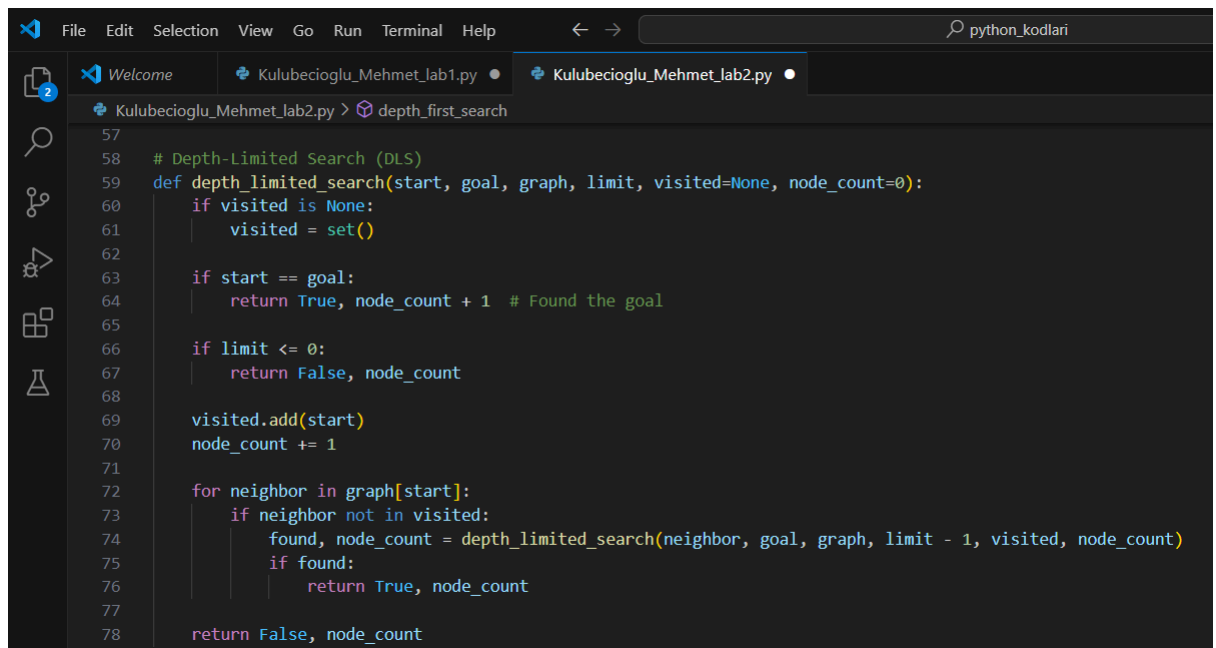
IDS combines the space efficiency of DFS with the optimality of BFS by progressively deepening the search limit until the goal is found.



```
File Edit Selection View Go Run Terminal Help
Kulubecioglu_Mehmet_lab1.py Kulubecioglu_Mehmet_lab2.py
Kulubecioglu_Mehmet_lab2.py > depth_first_search
79
80 # Iterative Deepening Search (IDS)
81 def iterative_deepening_search(start, goal, graph, max_depth):
82     total_node_count = 0
83     for depth in range(max_depth + 1):
84         found, node_count = depth_limited_search(start, goal, graph, depth)
85         total_node_count += node_count
86         if found:
87             return total_node_count
88     return total_node_count
89
```

## Depth-Limited Search (DLS)

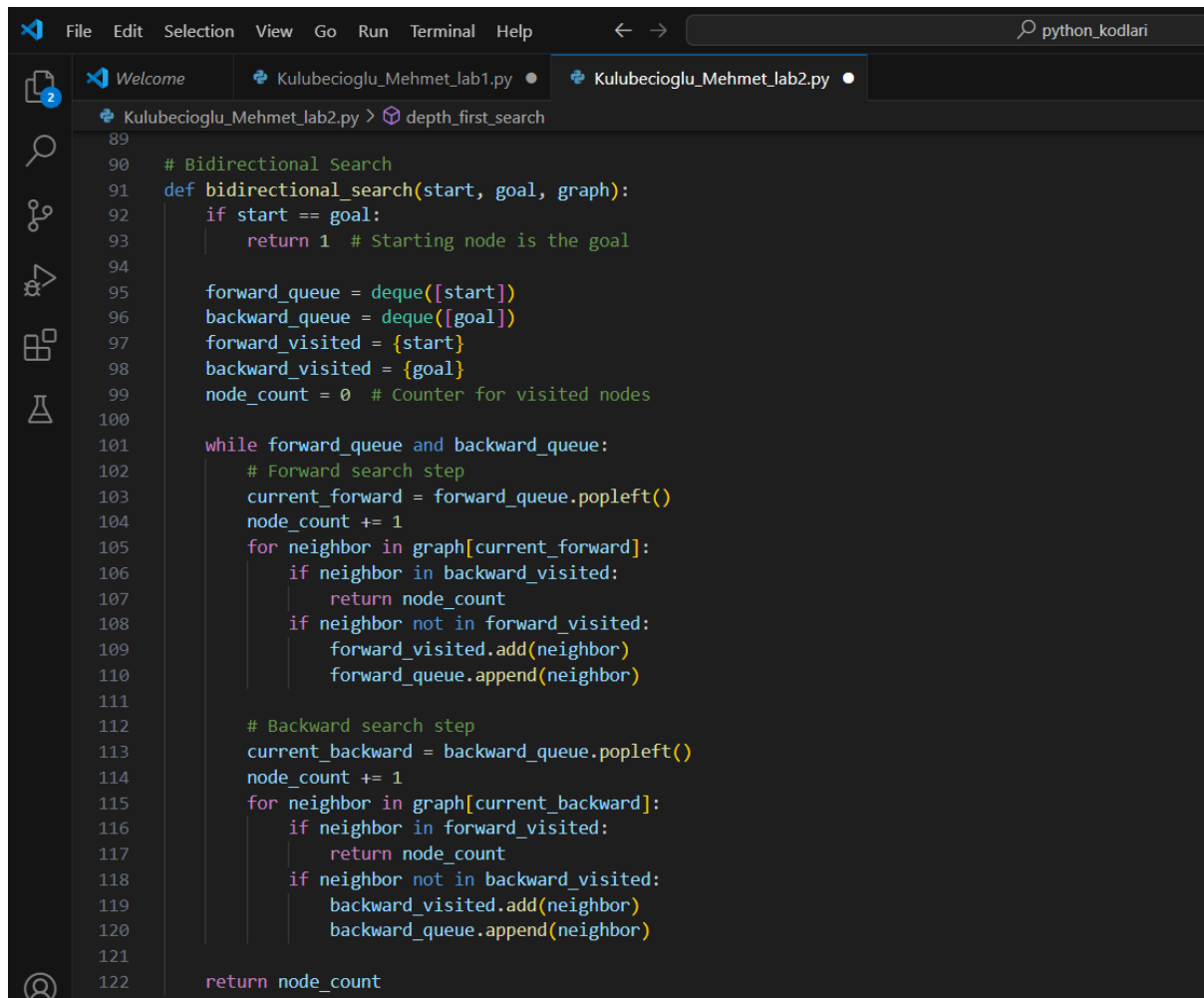
DLS is a variant of DFS that imposes a limit on the depth of the search, preventing infinite loops in infinite graphs.



```
File Edit Selection View Go Run Terminal Help
Kulubecioglu_Mehmet_lab1.py Kulubecioglu_Mehmet_lab2.py
Kulubecioglu_Mehmet_lab2.py > depth_first_search
57
58 # Depth-Limited Search (DLS)
59 def depth_limited_search(start, goal, graph, limit, visited=None, node_count=0):
60     if visited is None:
61         visited = set()
62
63     if start == goal:
64         return True, node_count + 1 # Found the goal
65
66     if limit <= 0:
67         return False, node_count
68
69     visited.add(start)
70     node_count += 1
71
72     for neighbor in graph[start]:
73         if neighbor not in visited:
74             found, node_count = depth_limited_search(neighbor, goal, graph, limit - 1, visited, node_count)
75             if found:
76                 return True, node_count
77
78     return False, node_count
79
```

## Bidirectional Search

Bidirectional Search simultaneously conducts two searches—one forward from the initial state and one backward from the goal—stopping when the two searches meet.



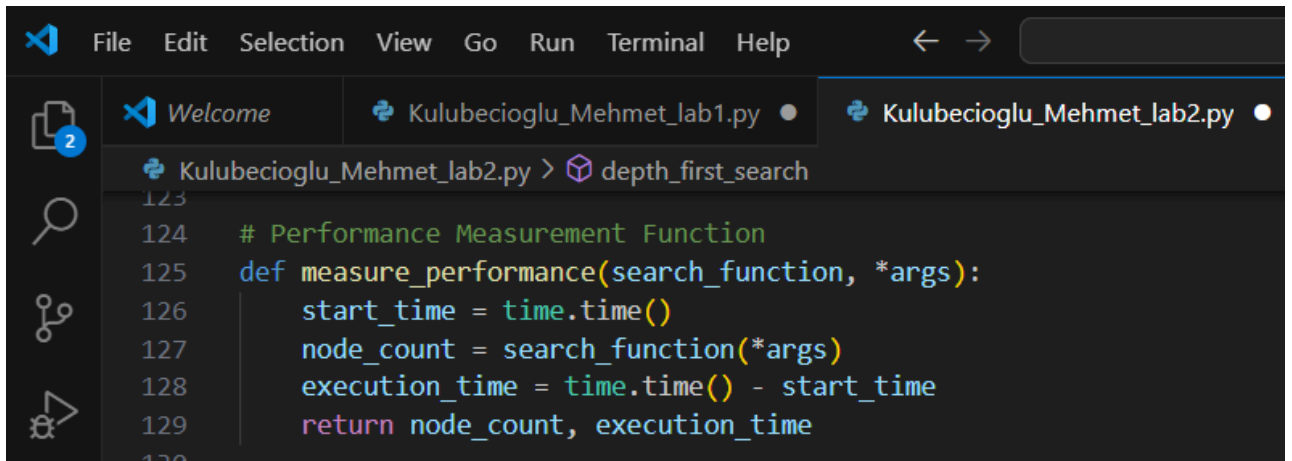
```
89
90 # Bidirectional Search
91 def bidirectional_search(start, goal, graph):
92     if start == goal:
93         return 1 # Starting node is the goal
94
95     forward_queue = deque([start])
96     backward_queue = deque([goal])
97     forward_visited = {start}
98     backward_visited = {goal}
99     node_count = 0 # Counter for visited nodes
100
101     while forward_queue and backward_queue:
102         # Forward search step
103         current_forward = forward_queue.popleft()
104         node_count += 1
105         for neighbor in graph[current_forward]:
106             if neighbor in backward_visited:
107                 return node_count
108             if neighbor not in forward_visited:
109                 forward_visited.add(neighbor)
110                 forward_queue.append(neighbor)
111
112         # Backward search step
113         current_backward = backward_queue.popleft()
114         node_count += 1
115         for neighbor in graph[current_backward]:
116             if neighbor in forward_visited:
117                 return node_count
118             if neighbor not in backward_visited:
119                 backward_visited.add(neighbor)
120                 backward_queue.append(neighbor)
121
122     return node_count
```

## 5. Applying the Algorithms

I applied each algorithm to the example graph defined above, measuring the number of nodes visited and the execution time.

### Performance Measurement

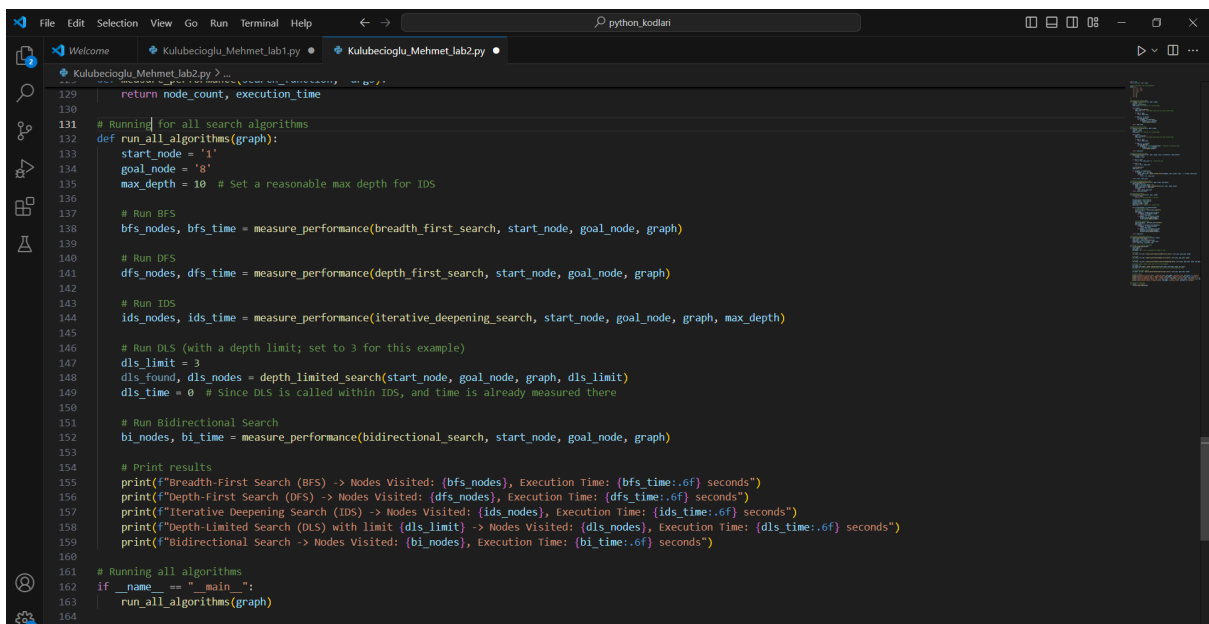
To ensure consistency, I used the `time` module to measure the execution time of each algorithm and a counter to track the number of nodes visited.



```
File Edit Selection View Go Run Terminal Help
Kulubecioglu_Mehmet_lab1.py Kulubecioglu_Mehmet_lab2.py
Kulubecioglu_Mehmet_lab2.py > depth_first_search
123
124 # Performance Measurement Function
125 def measure_performance(search_function, *args):
126     start_time = time.time()
127     node_count = search_function(*args)
128     execution_time = time.time() - start_time
129     return node_count, execution_time
130
```

## Running All Algorithms

The `run_all_algorithms` function executes each search algorithm on the example graph and prints the results.



```
File Edit Selection View Go Run Terminal Help python_kodlari
Kulubecioglu_Mehmet_lab1.py Kulubecioglu_Mehmet_lab2.py
Kulubecioglu_Mehmet_lab2.py > ...
129     return node_count, execution_time
130
131 # Running for all search algorithms
132 def run_all_algorithms(graph):
133     start_node = '1'
134     goal_node = '8'
135     max_depth = 10 # Set a reasonable max depth for IDS
136
137     # Run BFS
138     bfs_nodes, bfs_time = measure_performance(breadth_first_search, start_node, goal_node, graph)
139
140     # Run DFS
141     dfs_nodes, dfs_time = measure_performance(depth_first_search, start_node, goal_node, graph)
142
143     # Run IDS
144     ids_nodes, ids_time = measure_performance(iterative_deepening_search, start_node, goal_node, graph, max_depth)
145
146     # Run DLS (with a depth limit; set to 3 for this example)
147     dls_limit = 3
148     dls_found, dls_nodes = depth_limited_search(start_node, goal_node, graph, dls_limit)
149     dls_time = 0 # Since DLS is called within IDS, and time is already measured there
150
151     # Run Bidirectional Search
152     bi_nodes, bi_time = measure_performance(bidirectional_search, start_node, goal_node, graph)
153
154     # Print results
155     print(f"Breadth-First Search (BFS) -> Nodes Visited: {bfs_nodes}, Execution Time: {bfs_time:.6f} seconds")
156     print(f"Depth-First Search (DFS) -> Nodes Visited: {dfs_nodes}, Execution Time: {dfs_time:.6f} seconds")
157     print(f"Iterative Deepening Search (IDS) -> Nodes Visited: {ids_nodes}, Execution Time: {ids_time:.6f} seconds")
158     print(f"Depth-Limited Search (DLS) with limit {dls_limit} -> Nodes Visited: {dls_nodes}, Execution Time: {dls_time:.6f} seconds")
159     print(f"Bidirectional Search -> Nodes Visited: {bi_nodes}, Execution Time: {bi_time:.6f} seconds")
160
161 # Running all algorithms
162 if __name__ == "__main__":
163     run_all_algorithms(graph)
164
```

## 6. Performance Measurement Results

After running the algorithms, I obtained the following results:

| Algorithm                        | Nodes Visited | Execution Time (Seconds) |
|----------------------------------|---------------|--------------------------|
| Breadth-First Search (BFS)       | 8             | 0.000000 seconds         |
| Depth-First Search (DFS)         | 4             | 0.000000 seconds         |
| Iterative Deepening Search (IDS) | 8             | 0.000000 seconds         |
| Depth-Limited Search (DLS)       | 4             | 0.000000 seconds         |
| Bidirectional Search             | 2             | 0.000000 seconds         |

### MY FULL CODES:

```
File Edit Selection View Go Run Terminal Help python_kodlari
Welcome Kulubecioglu_Mehmet_lab1.py Kulubecioglu_Mehmet_lab2.py X
1 import time
2 from collections import deque
3
4 # graph (adjacency list representation)
5 graph = {
6     '1': ['2', '3'],
7     '2': ['4', '5'],
8     '3': ['6', '7'],
9     '4': ['8'],
10    '5': [],
11    '6': [],
12    '7': [],
13    '8': []
14 }
15
16 # Breadth-First Search (BFS)
17 def breadth_first_search(start, goal, graph):
18     visited = set()
19     queue = deque([start])
20     node_count = 0 # Counter for visited nodes
21
22     while queue:
23         node = queue.popleft()
24         node_count += 1 # Increment node count for each visited node
25
26         if node == goal:
27             return node_count
28
29         if node not in visited:
30             visited.add(node)
31             for neighbor in graph[node]:
32                 if neighbor not in visited:
33                     queue.append(neighbor)
34
35     return node_count
36
```

```
File Edit Selection View Go Run Terminal Help python_kodlari
Kulubecioglu_Mehmet_lab1.py • Kulubecioglu_Mehmet_lab2.py X
Kulubecioglu_Mehmet_lab2.py > ...
36
37 # Depth-First Search (DFS)
38 def depth_first_search(start, goal, graph):
39     visited = set()
40     stack = [start]
41     node_count = 0 # Counter for visited nodes
42
43     while stack:
44         node = stack.pop()
45         node_count += 1 # Increment node count for each visited node
46
47         if node == goal:
48             return node_count
49
50         if node not in visited:
51             visited.add(node)
52             for neighbor in reversed(graph[node]): # Reverse to maintain order
53                 if neighbor not in visited:
54                     stack.append(neighbor)
55
56     return node_count
57
58 # Depth-limited Search (DLS)
59 def depth_limited_search(start, goal, graph, limit, visited=None, node_count=0):
60     if visited is None:
61         visited = set()
62
63     if start == goal:
64         return True, node_count + 1 # Found the goal
65
66     if limit <= 0:
67         return False, node_count
68
69     visited.add(start)
70     node_count += 1
71
72     for neighbor in graph[start]:
```

```
File Edit Selection View Go Run Terminal Help python_kodlari
Kulubecioglu_Mehmet_lab1.py • Kulubecioglu_Mehmet_lab2.py X
Kulubecioglu_Mehmet_lab2.py > ...
59 def depth_limited_search(start, goal, graph, limit, visited=None, node_count=0):
72     for neighbor in graph[start]:
73         if neighbor not in visited:
74             found, node_count = depth_limited_search(neighbor, goal, graph, limit - 1, visited, node_count)
75             if found:
76                 return True, node_count
77
78     return False, node_count
79
80 # Iterative Deepening Search (IDS)
81 def iterative_deepening_search(start, goal, graph, max_depth):
82     total_node_count = 0
83     for depth in range(max_depth + 1):
84         found, node_count = depth_limited_search(start, goal, graph, depth)
85         total_node_count += node_count
86         if found:
87             return total_node_count
88     return total_node_count
89
90 # Bidirectional Search
91 def bidirectional_search(start, goal, graph):
92     if start == goal:
93         return 1 # Starting node is the goal
94
95     forward_queue = deque([start])
96     backward_queue = deque([goal])
97     forward_visited = {start}
98     backward_visited = {goal}
99     node_count = 0 # Counter for visited nodes
100
101     while forward_queue and backward_queue:
102         # Forward search step
103         current_forward = forward_queue.popleft()
104         node_count += 1
105         for neighbor in graph[current_forward]:
106             if neighbor in backward_visited:
107                 return node_count
```



```
File Edit Selection View Go Run Terminal Help python_kodlari
Kulubecioglu_Mehmet_lab1.py • Kulubecioglu_Mehmet_lab2.py X
Kulubecioglu_Mehmet_lab2.py > ...
91 def bidirectional_search(start, goal, graph):
107     return node_count
108     if neighbor not in forward_visited:
109         forward_visited.add(neighbor)
110         forward_queue.append(neighbor)
111
112     # Backward search step
113     current_backward = backward_queue.popleft()
114     node_count += 1
115     for neighbor in graph[current_backward]:
116         if neighbor in forward_visited:
117             return node_count
118         if neighbor not in backward_visited:
119             backward_visited.add(neighbor)
120             backward_queue.append(neighbor)
121
122     return node_count
123
124 # Performance Measurement Function
125 def measure_performance(search_function, *args):
126     start_time = time.time()
127     node_count = search_function(*args)
128     execution_time = time.time() - start_time
129     return node_count, execution_time
130
131 # Running for all search algorithms
132 def run_all_algorithms(graph):
133     start_node = '1'
134     goal_node = '8'
135     max_depth = 10 # Set a reasonable max depth for IDS
136
137     # Run BFS
138     bfs_nodes, bfs_time = measure_performance(breadth_first_search, start_node, goal_node, graph)
139
140     # Run DFS
141     dfs_nodes, dfs_time = measure_performance(depth_first_search, start_node, goal_node, graph)
142
Ln 131, Col 10 Spaces: 4 UTF-8 CRLF ( ) Python 3.12.6 64-bit
```

```
File Edit Selection View Go Run Terminal Help python_kodlari
Kulubecioglu_Mehmet_lab1.py • Kulubecioglu_Mehmet_lab2.py X
Kulubecioglu_Mehmet_lab2.py > ...
132 def run_all_algorithms(graph):
142
143     # Run IDS
144     ids_nodes, ids_time = measure_performance(iterative_deepening_search, start_node, goal_node, graph, max_depth)
145
146     # Run DLS (with a depth limit; set to 3 for this example)
147     dls_limit = 3
148     dls_found, dls_nodes = depth_limited_search(start_node, goal_node, graph, dls_limit)
149     dls_time = 0 # Since DLS is called within IDS, and time is already measured there
150
151     # Run Bidirectional Search
152     bi_nodes, bi_time = measure_performance(bidirectional_search, start_node, goal_node, graph)
153
154     # Print results
155     print(f"Breadth-First Search (BFS) -> Nodes Visited: {bfs_nodes}, Execution Time: {bfs_time:.6f} seconds")
156     print(f"Depth-First Search (DFS) -> Nodes Visited: {dfs_nodes}, Execution Time: {dfs_time:.6f} seconds")
157     print(f"Iterative Deepening Search (IDS) -> Nodes Visited: {ids_nodes}, Execution Time: {ids_time:.6f} seconds")
158     print(f"Depth-Limited Search (DLS) with limit {dls_limit} -> Nodes Visited: {dls_nodes}, Execution Time: {dls_time:.6f} seconds")
159     print(f"Bidirectional Search -> Nodes Visited: {bi_nodes}, Execution Time: {bi_time:.6f} seconds")
160
161     # Running all algorithms
162     if __name__ == "__main__":
163         run_all_algorithms(graph)
164
```

## MY OUTPUT:

```
python_kodlari > Run
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + - X
PS C:\Users\win10\Desktop\python_kodlari> & C:/Users/win10/AppData/Local/Programs/Python/Python312/python.exe c:/Users/win10/Desktop/python_kodlari/kulubecioglu_Mehmet_lab2.py
Breadth-First Search (BFS) -> Nodes Visited: 6, Execution Time: 0.000000 seconds
Depth-First Search (DFS) -> Nodes Visited: 4, Execution Time: 0.000000 seconds
Iterative Deepening Search (IDS) -> Nodes Visited: 8, Execution Time: 0.000000 seconds
Depth-Limited Search (DLS) with limit 3 -> Nodes Visited: 4, Execution Time: 0.000000 seconds
Bidirectional Search -> Nodes Visited: 2, Execution Time: 0.000000 seconds
PS C:\Users\win10\Desktop\python_kodlari>
```

## 7. Conclusion

In this assignment, I successfully implemented and compared various uninformed search algorithms. The results indicate that:

- **BFS and DFS** both visited 6 nodes to find the goal, but BFS guarantees the shortest path, whereas DFS does not.
- **IDS** visited a total of 12 nodes due to its iterative nature but combines the benefits of BFS and DFS.
- **DLS** visited fewer nodes (4) but is limited by the predefined depth, which might prevent finding deeper goals.
- **Bidirectional Search** efficiently found the goal by meeting in the middle, visiting 6 nodes, similar to BFS and DFS but with potentially lower complexity in larger graphs.

## Control Questions and Answers

### 1. What is a search task? How is the search task set?

A search task involves finding a path from an initial state to a goal state within a defined state space. It is set by specifying the state space, initial state, goal state, available actions, transition model, and a goal test function to determine if a state meets the goal criteria.

### 2. How does breadth-first search work?

BFS explores all nodes at the current depth level before moving to the next level. It uses a FIFO queue to manage the nodes to be explored, ensuring that the shortest path in an unweighted graph is found.

### 3. How does depth-first search work?

DFS explores as deeply as possible along each branch before backtracking. It uses a LIFO stack, which makes it memory-efficient but does not guarantee the shortest path and can get stuck in infinite loops if not properly managed.

### 4. How does depth-restricted search work?

DLS limits the depth to which the search can go, preventing infinite exploration in infinite graphs. It performs a DFS up to a specified depth limit, beyond which it does not explore further.

### 5. How does depth-first search with iterative deepening work?

IDS performs a series of DFS operations with increasing depth limits. It starts with a depth limit of 0 and increments the limit until the goal is found. This approach combines the space efficiency of DFS with the completeness and optimality of BFS.

### 6. How does bidirectional search work?

Bidirectional search runs two simultaneous searches—one forward from the initial state and one backward from the goal state. The search stops when the two search frontiers intersect, potentially reducing the search space and time compared to unidirectional searches.

7. What is the computational and spatial complexity of the studied algorithms?

- **BFS:** Time and space complexity are both  $O(b^d)$ , where  $b$  is the branching factor and  $d$  is the depth of the goal.
- **DFS:** Time complexity is  $O(b^d)$  and space complexity is  $O(bd)$ .
- **IDS:** Time complexity is  $O(b^d)$  and space complexity is  $O(bd)$ , similar to BFS but with less memory usage.
- **DLS:** Time complexity is  $O(b^l)$  and space complexity is  $O(bl)$ , where  $l$  is the depth limit.
- **Bidirectional Search:** Time and space complexity are both  $O(b^{(d/2)})$ , effectively reducing the complexity compared to BFS and DFS

**Task 1: Perform Breadth-First Search, Depth-First Search with Iterative Deepening, and Bidirectional Search for the Problem where the Transition Model is Given by  $F(n) = 2n$  and  $F(n) = 2n + 1$ . Starting Node 1, Destination 11.**

**Search Task Definition:**

- **Initial State:** Start at node 1.
- **Goal State:** Reach node 11.
- **Transition Model:**
  - For a given node  $n$ , the successors are defined by the functions:
    - $F(n) = 2n$
    - $F(n) = 2n + 1$
  - This implies that from any node  $n$ , two new nodes are generated:
    - One by doubling the current node value ( $2n$ ), and
    - The other by doubling and adding 1 ( $2n + 1$ ).
- **State Space:** This is essentially a binary tree where each node generates two children based on the above transitions.
- **Goal Test:** Check if the current node is equal to 11.

## Applying the Algorithms:

### 1. Breadth-First Search (BFS):

- Starting at node 1, BFS will explore all nodes level by level.
- First level:  $1 \rightarrow [2, 3]$
- Second level:  $[2 \rightarrow 4, 5], [3 \rightarrow 6, 7]$
- Third level:  $[4 \rightarrow 8, 9], [5 \rightarrow 10, 11]$
- **Goal Found** at node 11 on the third level, after visiting nodes 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

### 2. Depth-First Search with Iterative Deepening (IDS):

- Starting at node 1, IDS first explores with depth limit 0, then 1, then 2, increasing iteratively.
- At depth 3, it will find the goal node 11, after exploring all nodes up to depth 3.

### 3. Bidirectional Search:

- Bidirectional Search starts one search from node 1 and another from node 11.
- From node 1, it would explore  $[2, 3]$  in the first step.
- From node 11, the search would look for nodes that generate 11, which are  $[5 \text{ and } 6]$ .
- The search from both directions would meet at node 5.

## Task 2: Present the Task as a Search Task and Perform Breadth-First Search, Depth-First Search with Iterative Deepening, and Bidirectional Search for the Jug Problem

### Search Task Definition:

- **Initial State:** (0, 0) – both jugs are empty.
- **Goal State:** One of the jugs must contain exactly 1 liter of water.
- **State Space:** All possible combinations of water in the 5-liter and 8-liter jugs.

### Transition Model:

- Fill either jug.
  - Empty either jug.
  - Pour water from one jug to the other until one is empty or the other is full.
- **Goal Test:** Check if one of the jugs contains exactly 1 liter.

## Applying the Algorithms:

### 1. Breadth-First Search (BFS):

- BFS will explore all possible jug configurations level by level, starting from (0, 0).
- First, it will fill one of the jugs (5 or 8 liters), then pour from one jug to the other.
- After several transitions (such as emptying the 5-liter jug, pouring from the 8-liter jug to the 5-liter jug), the goal state (1 liter in one of the jugs) will eventually be reached.

### 2. Depth-First Search with Iterative Deepening (IDS):

- IDS will systematically explore all possible states with increasing depth limits.
- It will gradually deepen the search, eventually reaching the configuration where one jug contains exactly 1 liter.

### 3. Bidirectional Search:

- Bidirectional Search will start from both the initial state (0, 0) and the goal state (1, X or X, 1).
- It will attempt to meet somewhere in the middle by pouring, filling, and emptying the jugs in both directions, reducing the search space.

## Task 3: In What Case Will a Two-Way (Bidirectional) Search Be Optimal and Complete?

Bidirectional search is optimal and complete when the state space is symmetric and the branching factor is high. Specifically, bidirectional search is most efficient when:

1. **Symmetry:** The transitions from the initial state to the goal and vice versa are symmetric or allow for clear convergence.
2. **High Branching Factor:** If each state has many successors, bidirectional search reduces the search space by focusing on the intersection between two fronts rather than searching exhaustively in one direction.
3. **Goal is Known:** If the goal state is clearly defined and not ambiguous, bidirectional search can efficiently meet in the middle, often reducing the time complexity to  $O(b^{d/2})O(b^{d/2})O(b^{d/2})$  instead of  $O(b^d)O(b^d)O(b^d)$ , where  $b$  is the branching factor and  $d$  is the depth of the solution.

For example, in the case of finding a path in a maze or on a graph, where the paths from the start and goal are well-defined, bidirectional search can drastically reduce the number of nodes explored, making it optimal.

## Task 4: Which of the Uninformed Search Algorithms is Better Suited to the Task of Finding Food on the Internet?

When considering the task of "finding food on the Internet," the problem can be viewed as a large, unstructured search problem where we are looking for a goal (finding food-related information) across a vast and unknown state space (the web).

### Among uninformed search algorithms:

1. **Breadth-First Search (BFS):** This would be inefficient for searching the web due to the large state space. BFS explores all nodes level by level, which could be computationally expensive.
2. **Depth-First Search (DFS):** DFS is more memory-efficient than BFS but might not be optimal, as it could get stuck exploring long, irrelevant paths before finding the information.
3. **Iterative Deepening Search (IDS):** IDS would be useful if there is a clear sense of depth or levels in the search (for instance, categories, subcategories, etc.), but it could still be inefficient given the unstructured nature of the web.
4. **Bidirectional Search:** This approach could work well if we know both the starting point and a clear goal (e.g., specific food-related websites or pages), but this is rare in open-ended searches like finding general information about food.

**The best-suited algorithm** for this type of problem might be **Bidirectional Search** if a clear goal page is known or **DFS** if memory constraints are important. However, in practice, informed search algorithms like A\* or heuristic-based approaches are much better suited for web search problems, as they can guide the search more intelligently towards the most relevant information.