PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

COURSE CODE CIT 316
Artificial Intelligence Sessional

SUBMITTED TO:

Dr. Md Abdul Masud

Professor,
Department of Computer Science and Information Technology,
Faculty of Computer Science and Engineering

SUBMITTED BY:

Md. Sharafat Karim

ID: **2102024**,
Registration No: **10151**
Faculty of Computer Science and Engineering

Lab 01
Assignment title: Searching Algorithms
Date of submission:

Lab 01 - Searching Algorithms

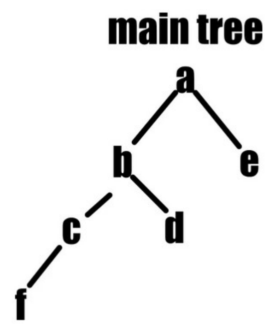
In artificial intelligence, searching problems involve finding a solution from a large space of possible solutions. We can categorize it into,

1. Uninformed searching

These algorithms do not have any additional information about the goal state. Here's some of them are described below,

First, let's consider a tree for our algorithm,
In python, we can define it like,

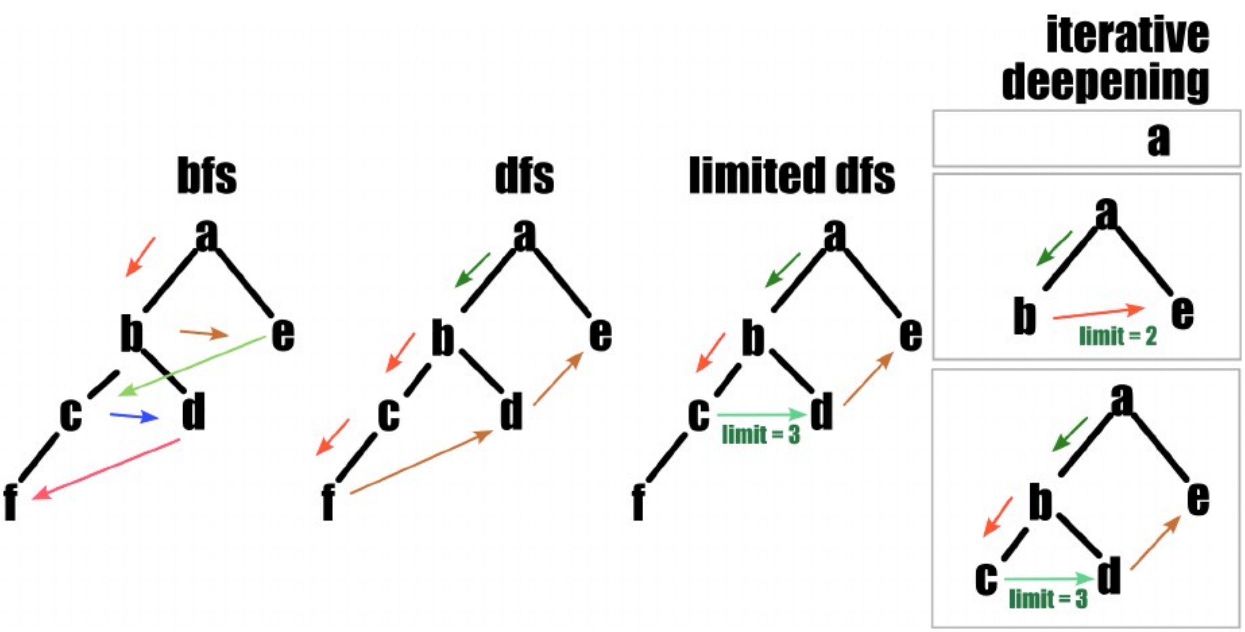
```
tree = {
        'A': ['B', 'E'],
        'B': ['C', 'D'],
        'C': ['F'],
        'D': [],
        'E': [],
        'F': []
}
```



Now, we can visualize our algorithms like,

1.1 Breadth First Search (BFS)

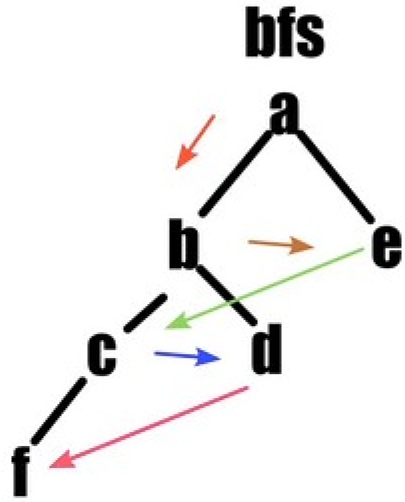
BFS (Breadth First Search) is an algorithm for traversing or searching tree or graph data structures. It



starts at the root node and explores all neighboring nodes at the present depth prior to moving on to nodes at the next depth level.

```
from collections import deque

def bfs(tree, start):
    visited = []
```

```
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.append(node)
            print(node, end=" ")

            for neighbor in tree[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

bfs(tree, 'A')
```

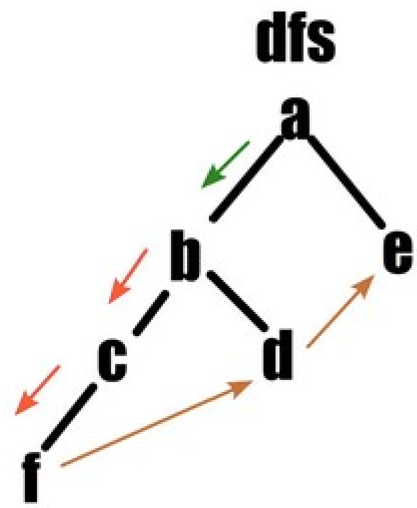


A B E C D F

1.2 Breadth First Search (BFS)

DFS (Depth-First Search) is an algorithm for traversing or searching tree or graph data structures. It starts at the root node and explores as far as possible along each branch before backtracking.

```
def dfs(tree, start, visited = []):
    if start not in visited:
        print(start, end=" ")
        visited.append(start)
    for node in tree[start]:
        dfs(tree, node, visited)

dfs(tree, 'A')
```

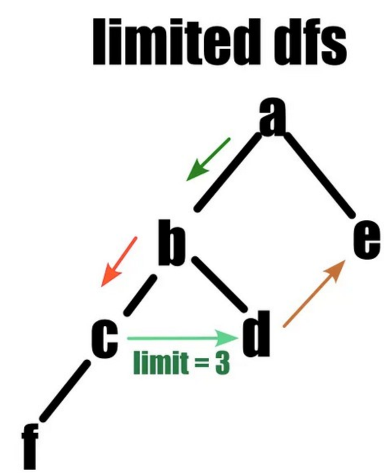


A B C F D E

1.3 DFS Limited

In DFS limited, instead of exploring till the deepest node, we will limit our step number.

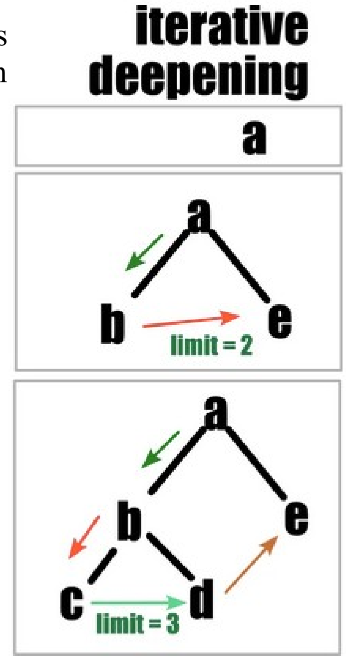
```
def dfs_limited(tree, start, limit, visited=[]):
    if limit <= 0:
        return
    if start not in visited:
        print(start, end=" ")
        visited.append(start)
    for node in tree[start]:
        dfs_limited(tree, node, limit-1, visited)

dfs_limited(tree, 'A', 3)
```



A B C D E

1.4 Iterative DFS

Iterative deepening DFS is a hybrid of DFS and BFS. It repeatedly applies DFS with increasing depth limits until a goal is found. This approach combines the space efficiency of DFS with the completeness of BFS.




```
def iterative_deepening(tree, start, max_limit):
    for i in range(max_limit):
        print(f"Iteration {i+1} : ", end="")
        dfs_limited(tree, start, i+1, [])
        print()

iterative_deepening(tree, 'A', 4)
```

```
Iteration 1 : A
Iteration 2 : A B E
Iteration 3 : A B C D E
Iteration 4 : A B C F D E
```

1.5 Bidirectional Search

Bidirectional search is a graph search algorithm that simultaneously explores the search space from both the initial state and the goal state. The search continues until the two searches meet, thus finding the shortest path more efficiently than traditional unidirectional search methods.

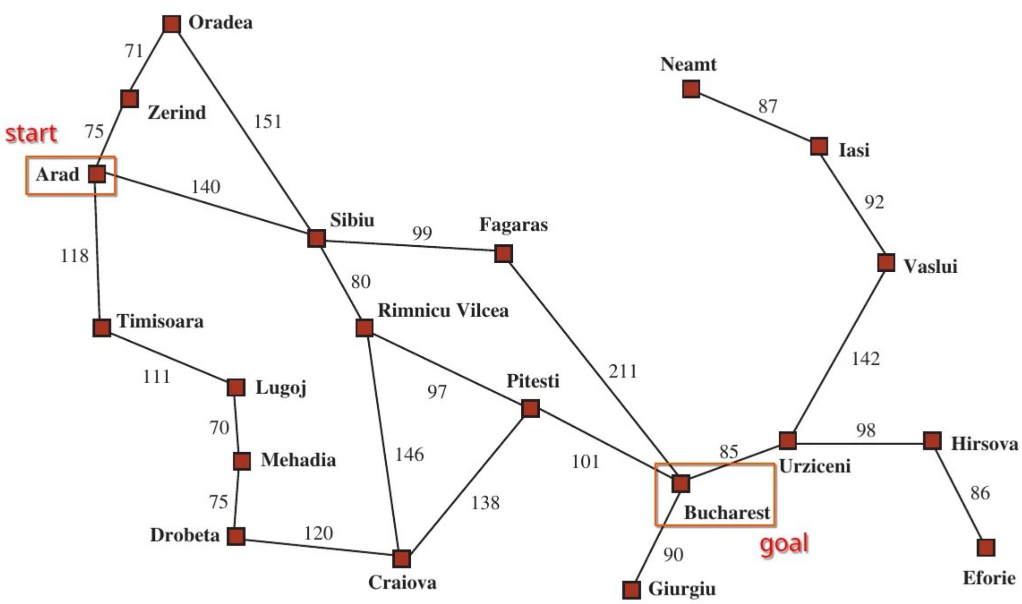
```
# Unidriected Tree
un_tree = {
        'A': ['B', 'E'],
        'B': ['C', 'D', 'A'],
        'C': ['F', 'B'],
        'D': ['B'],
        'E': ['A'],
        'F': ['C']
    }

from collections import deque
def bidirectional(tree, start, goal):
    if start == goal:
        return None, None

    start_visited = []
    goal_visited = []

    start_queue = deque([start])
    goal_queue = deque([goal])

    while start_queue and goal_queue:
        start_node = start_queue.popleft()
        if start_node not in start_visited:
            start_visited.append(start_node)
```

```
            for neighbour in tree[start_node]:
                if neighbour not in start_visited:
                    start_queue.append(neighbour)

        goal_node = goal_queue.popleft()
        if goal_node not in goal_visited:
            goal_visited.append(goal_node)

            for neighbour in tree[goal_node]:
                if neighbour not in goal_visited:
                    goal_queue.append(neighbour)

        if start_node in goal_visited or goal_node in start_visited:
            return start_visited, goal_visited

print(bidirectional(un_tree, 'A', 'F'))
```

```
(['A', 'B', 'E'], ['F', 'C', 'B'])
```

1.6 Uniform Cost Search (UCS)

UCS or uniform cost search is a variant of Dijkstra's algorithm and is used to find the least-cost path from a starting node to a goal node. UCS expands the least costly node in the search space, ensuring that the first time it reaches the goal node, it has found the optimal path.




```
from heapq import heappush, heappop

def greedy_best_first(graph, heuristic, start, goal):
    frontier = []
    heappush(frontier, (heuristic[start], 0, start, [start]))
    best_graph = {start: 0}

```

```
    while frontier:
        cost, path_cost, node, path_list = heappop(frontier)
        if node == goal:
            return path_cost, path_list
        for neighbor, neighbor_cost in graph[node]:
            updated_cost = cost + heuristic[neighbor]
            if neighbor not in best_graph or updated_cost < best_graph[neighbor]:
                best_graph[neighbor] = updated_cost
                heappush(frontier, (updated_cost, path_cost + neighbor_cost, neighbor, path_list + [neighbor]))

cost, path= greedy_best_first(G, H, "Arad", "Bucharest")
print("Path : ", " -> ".join(path))
print("Cost : ", cost)
```

```
Path :  Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost :  418
```

2. Informed Searching

In informed search heuristic values are also contained in our input, which we will use to determine the result!

2.1 Greedy Best-First Search

Greedy Best-First Search is an informed search algorithm that uses a heuristic to estimate the cost to reach the goal from a given node.

```
from heapq import heappush, heappop

def greedy_best_first(graph, heuristic, start, goal):
    frontier = []
    heappush(frontier, (heuristic[start], 0, start, [start]))
    best_graph = {start: 0}

    while frontier:
        cost, path_cost, node, path_list = heappop(frontier)
        if node == goal:
            return path_cost, path_list
        for neighbor, neighbor_cost in graph[node]:
            updated_cost = cost + heuristic[neighbor]
            if neighbor not in best_graph or updated_cost < best_graph[neighbor]:
                best_graph[neighbor] = updated_cost
                heappush(frontier, (updated_cost, path_cost + neighbor_cost, neighbor, path_list + [neighbor]))

cost, path= greedy_best_first(G, H, "Arad", "Bucharest")
print("Path : ", " -> ".join(path))
print("Cost : ", cost)
```

```
Path :  Arad -> Sibiu -> Fagaras -> Bucharest
Cost :  450
```

2.2 A* Search

A* Search is an informed search algorithm that combines the strengths of Uniform Cost Search and Greedy Best-First Search. The total cost function is

    f(n) = g(n) + h(n)

```python
from heapq import heappush, heappop

def a_star(graph, heuristic, start, goal):
    frontier = []
    heappush(frontier, (heuristic[start], 0, start, [start]))
    best_graph = {start: 0}

    while frontier:
        cost, path_cost, node, path_list = heappop(frontier)
        if node == goal:
            return path_cost, path_list
        for neighbor, neighbor_cost in graph[node]:
            updated_cost = path_cost + neighbor_cost + heuristic[neighbor]
            if neighbor not in best_graph or updated_cost < best_graph[neighbor]:
                best_graph[neighbor] = updated_cost
                heappush(frontier, (updated_cost, path_cost + neighbor_cost, neighbor, path_list + [neighbor]))

cost, path= a_star(G, H, "Arad", "Bucharest")
print("Path : ", " -> ".join(path))
print("Cost : ", cost)
```

```
Path :  Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost :  418
```

Signature