جَامعة المَنارة
MANARA UNIVERSITY

Introduction to Artificial Intelligence

Lecture 3: DFS and GBFS Algorithms

Depth First Search (DFS) Algorithm

The concept behind DFS:
Depth-first search (DFS) is a search algorithm that starts at a node S and follows one path as far as it can go. When it can't go any farther (No more children exist), it backtracks to the last point where a different path was available and continues from there. Instead of exploring the graph level by level (like BFS), it explores it by going deep first.
DFS uses a stack (Last In, First Out) to keep track of what to explore next.

Approach:

- Begin with a frontier that holds only the start node (the only available step at first).
- Keep a visited set that starts empty.
- Then repeat:
  - If the frontier is empty, there is no solution.
  - Take the most recently added node from the frontier (treat it as a stack).
  - If this node is the goal, return the solution.
  - Add the node to the explored set.
  - Expand the node; add each new node (neighbor or child node) to the frontier if it's not already in the frontier or the visited set.

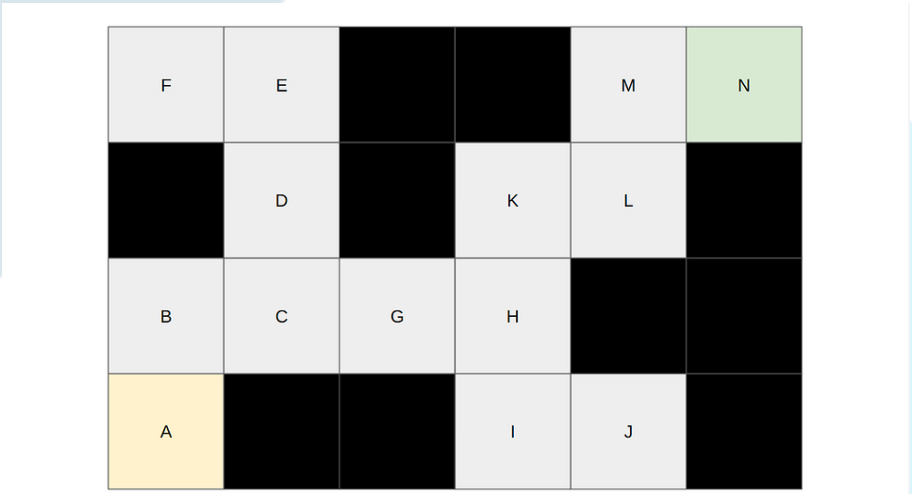
DFS is like following one path all the way down before checking the others.

DFS Code:

The DFS function is almost identical to the BFS version, with one key difference. In BFS, the frontier is treated as a queue, so nodes are removed in the order they were added. For DFS, the frontier must act as a stack, meaning the most recently added node is removed first.
The only required change is replacing frontier.pop(0) with frontier.pop().
Using pop() removes the last inserted element, giving the frontier stack-like behavior and ensuring the algorithm follows a depth-first order.
The following code is the DFS code including comments that explains the purpose behind each line of the code.

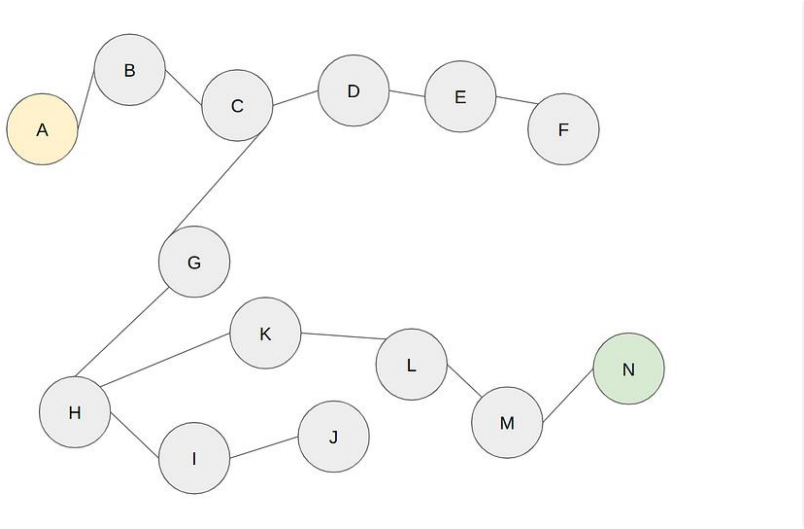
```python
def dfs(graph, start_node, end_node):

    solution = []          # stores path taken from start to the goal nodes
    costs = 0              # counts how many loop iterations happened (number of edges)

    frontier = []          # acts as a stack for DFS
    visited = []           # keeps track of nodes already seen

    frontier.append(start_node)     # start by pushing the start node

    while frontier:                 # continue while there are nodes to explore

        selected_node = frontier.pop()     # pop the last added node (Stack behavior)
        visited.append(selected_node)      # Mark the current node as visited
        solution.append(selected_node)     # Add the current node to the solution path

        if selected_node == end_node:      # if the goal is found
            break                          # stop searching

        # explore all neighbors of the current node
        for neighbour in graph[selected_node]:
            if neighbour not in visited:          # only add new nodes
                frontier.append(neighbour)        # push neighbour onto stack

        costs += 1    # count how many loop cycles were executed

    return solution, costs    # return visit order + cost
```

Exercise:
You are lost in the following maze standing at A square. Use DFS algorithm to determine whether there is a way out ***called N*** or not.



Solution:

In order to figure out whether there is a node called N (my way out) I have to apply DFS to the maze. We first **convert the maze into a graph**, where each open cell becomes a node and edges connect neighboring cells that can be moved between.



Then we should convert this graph to a dictionary in order to use it in the code. Assuming directed graph:

```
Maze = {
    "A": ["B"],
    "B": ["C"],
    "C": ["D", "G"],
    "D": ["E"],
    "E": ["F"],
    "F": [],
    "G": ["H"],
    "H": ["K", "I"],
    "K": ["L"],
    "L": ["M"],
    "M": ["N"],
    "N": [],
    "I": ["J"],
    "J": [],
}
```

Then we define the start and goal nodes, and we call the dfs function defined earlier. Finally, we investigate whether the N node exists in my solution (there is a way out) or not.

```
start_state = "A"
goal_state = "N"

[s, c] = dfs(state_space, start_state, goal_state)
if (s[-1]=="N"):
    print("There is a way out and the solution is ", s)
else:
    print("There is no way out!!")
```

Output is:

```
There is a way out and the solution is ['A', 'B', 'C', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N']
```

Note: The order of adding available nodes to the stack is not important (beginning from the left or right) unless the application requires a specific order. In case I need to change the order of adding neighbor nodes to the stack I can use ***reversed*** command in the for loop of the dfs function as follow:

```
for neighbour in reversed(graph[selected_node])
```

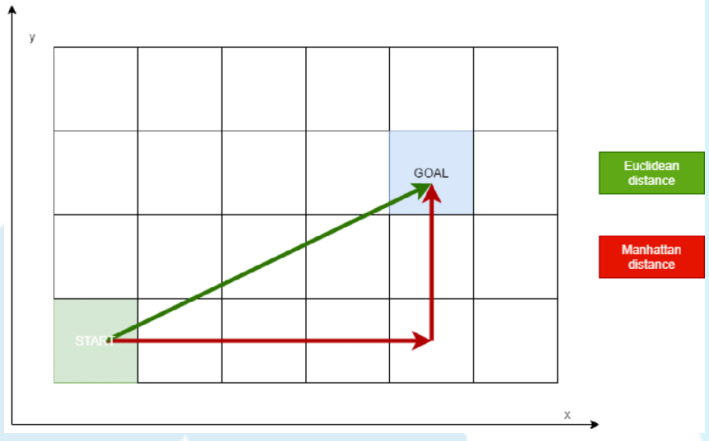
Greedy Best-First Search (GBFS) Algorithm

Greedy Best-First Search is an informed search algorithm that aims to find a solution quickly by always expanding the node that appears closest to the goal based on a heuristic function.

Heuristic Function:

A heuristic is an **approximate** measure of how close the node is to the target. Different heuristics can be employed, such as Manhattan distance for grid-based paths or Euclidean distance for free space. The choice of the heuristic function influences the search efficiency.

- Manhattan Distance: how far apart two points are by summing the absolute differences of their coordinates. For two points P(x1, y1) and Q(x2, y2), the Manhattan distance is: $|x2 - x1| + |y2 - y1|$
- Euclidean distance: the straight-line (shortest) distance between two points. For two points P(x1, y1) and Q(x2, y2), the Euclidean distance is: $\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$



GBFS Approach:

- Start with a frontier that contains the start node.
- Start with an empty explored set.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove the node with the lowest heuristic value from the frontier.
  - If the node is the goal state, return the solution.
  - Add the node to the explored set.
  - Expand node, add resulting nodes to the frontier if they aren't already in the frontier or the explored set.

To achieve this, the frontier in GBFS is represented as a priority queue.

Priority Queue:

A **priority queue** is a type of queue where each element is associated with a priority value, and elements are served based on their priority rather than their insertion order. In other words, elements with higher priority are retrieved or removed before those with lower priority.

So, every element in the priority queue has the shape (p,v) where:

1) p: is a priority value that determines the order in which elements are removed.
2) v: is the value, which is the actual item being stored.

The structure itself does not assign meaning to p, so the interpretation of what counts as "higher" priority is defined by the algorithm that uses the priority queue. In Greedy Best-First Search, for example, the value v represents a node's name, while the priority p is the heuristic estimate h(n). The algorithm interprets smaller heuristic values as higher priority, so the priority queue should always remove the node with the lowest h(n) first.

According to this, there are two types of priority queues:

- **min-priority queue**: the element with the *lowest numeric priority value* p is removed first, because the algorithm treats lower numbers as indicating higher priority.
- **max-priority queue**: the element with the *highest numeric priority value* p is removed first, because the algorithm treats higher numbers as indicating higher priority.

Priority Queue in Python:

In Python, there is a built-in PriorityQueue class that supports basic methods (insertion and removal) in an efficient manner.

- **Importing**
  `from queue import PriorityQueue` (Imports the `PriorityQueue` class).
- **Creating a Priority Queue**
  `pq = PriorityQueue()` (Initializes an empty priority queue).
- **Inserting an Element**
  `pq.put((priority, value))` (Adds an element as a `(priority, value)` pair).

- **Removing the Smallest Item**
  `pq.get()` (Retrieves and removes the element with the lowest numeric priority value).
- **Checking if empty**
  `pq.empty()` (Returns True if the queue has no elements)
- **Inspecting the Queue**
  `print(pq.queue)` (Displays the internal list used by the priority queue (for debugging purposes)).

GBFS Code:

The main difference between the BFS and the GBFS codes is implementing the frontier as a priority queue in GBFS, which implies:

- Passing the heuristic as a parameter to the GBFS function. `heuristic` is defined as a dictionary, where keys are node names and values are numeric heuristic estimates.
- Inserting each element into the priority queue as a **tuple** `(heuristic_value, node_name)`, where the heuristic value determines the node's priority.
- Changing the insertion method from `append` to `put`.
- Changing the removal method from `pop` to `get`.

```
from queue import PriorityQueue # Importing the built-in PriorityQueue class

def greedy_best_first_search(graph, start_node, end_node, heuristic): # Passing the heuristic as a parameter

    solution = []
    costs = 0

    frontier = PriorityQueue()  # Define the frontier as a priority queue
    visited = []

    frontier.put((heuristic[start_node], start_node))   # Push the start node as (heuristic_value, node_name)

    while not frontier.empty(): # Continue while there are nodes in the frontier

        _, selected_node = frontier.get()   # Remove the node with the smallest heuristic
        visited.append(selected_node)
        solution.append(selected_node)

        if selected_node == end_node:
            break

        for neighbour in graph[selected_node]:
            if neighbour not in visited:
                frontier.put((heuristic[neighbour], neighbour))#Push each neighbour node as(heuristic_value, node_name)

        costs += 1

    return solution, costs
```

Notes:

- `frontier.put((heuristic[start_node],start_node))`
  - Since `heuristic` is a dictionary, `heuristic[start_node]` retrieves the heuristic value associated with the node `start_node`.
  - The parentheses ( ) are crucial because the priority queue is expecting tuples as elements.

- `_, selected_node = frontier.get()`

- Each element in the queue is stored as a tuple `(priority, node)`, so `frontier.get()` returns `(heuristic_value, node)`. The underscore _ is used to **ignore the heuristic value**, since at this point, we only need the node itself (`selected_node`) to expand it.

Solving the maze exercise with GBFS:

To solve the previous maze exercise with GBFS, we need to first convert the maze into a graph (same as before). Then, we need to define a heuristic function and calculate its value for each node in the graph. Since this maze is grid-based (the possible actions are: moving left, moving right, moving up, moving down), we will use the Manhattan Distance as our heuristic.

After calculating the Manhattan Distance for each node, we define the heuristic dictionary, where the keys are the node names, and the values are the Manhattan Distances:

```
heuristic = {
    "A": 8,
    "B": 7,
    "C": 6,
    "D": 5,
    "E": 4,
    "F": 5,
    "G": 5,
    "H": 4,
    "K": 3,
    "L": 2,
    "M": 1,
    "N": 0,
    "I": 5,
    "J": 4,
}
```

Then we define the start and goal nodes, and we call the greedy_best_first_search function defined earlier.

```
start_node = "A"
goal_node = "N"

[solution, cost] = greedy_best_first_search (Maze, start_node, goal_node,
heuristic)
```

Finally, we investigate whether the N node exists in my solution (there is a way out) or not as we did in the DFS exercise. The output is:

```
There is a way out and the solution is ['A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'K', 'L', 'M', 'N']
```