Artificial Intelligent

With Python

Lab 9

Supervisor:
Prof.Dr. Wafaa Mohamed
by:
Alyaa Mohamed
Yasser Saad AL-Khazraji

Blind Search & Breadth-First Search (BFS)

What is Blind Search?

Blind Search, also known as uninformed search, refers to search algorithms that explore the search space without any domain-specific knowledge. They rely solely on the problem definition such as the initial state, possible actions, and goal state without using heuristics.

❑ **Examples of Applications:**

- Solving puzzles (e.g., 8-puzzle, maze solving)
- Pathfinding problems
- Game trees exploration

Why a queue for DFS ?

In **Breadth-First Search (BFS)**, the algorithm explores the search space **level by level** — that is, it visits all the nodes at a given depth before moving on to nodes at the next depth.

To achieve this order, BFS uses a **queue data structure** **(FIFO: First In, First Out)**. This means:

- The **first node inserted** into the queue is the **first one processed**.
- New nodes (children) are added to the **end of the queue**, while nodes being explored are **removed from the front**.

Functions queue

Operation	Description	Python Code
Create Queue	Create an empty queue	queue = []
Enqueue	Add element to the **end**	queue.append(item)
Dequeue	Remove element from the **front**	item = queue.pop(0)
Check Empty	Check if queue is empty	if len(queue) == 0:
Size	Get number of elements	len(queue)
Front Element (Peek)	View first element without removing	queue[0]

Queue using a Set

Sets are unordered; cannot represent true FIFO queues.
However, they can be used to store visited elements (like CLOSED in BFS).

```
queue = {'A', 'B', 'C'}
print("Initial set:", queue)
queue.add('D')
print("After adding D:", queue)
queue.remove('A')
print("After removing A:", queue)
```

```
Initial set: {'A', 'B', 'C'}
After adding D: {'A', 'B', 'C', 'D'}
After removing A: {'B', 'C', 'D'}
```

Queue using a Tuple

Tuples are immutable; we create a new tuple after each operation.

```
queue = ('A', 'B', 'C')
print("Initial:", queue)
queue = queue + ('D',)     # enqueue
print("After enqueue:", queue)
queue = queue[1:]          # dequeue
print("After dequeue:", queue)
```

```
Initial: ('A', 'B', 'C')
After enqueue: ('A', 'B', 'C', 'D')
After dequeue: ('B', 'C', 'D')
```

Breadth-First Graph Search Algorithm

Input: State Space

Ouput: *failure* or path from a start state to a goal state.

Assumptions: *Open* is a list of nodes that have not yet been examined.

Closed is the list of states that have been examined.

1. Set *Open* to be a list of the initial nodes in the problem. At any given point in time.
2. While *Open* is not empty
    1. Pick a node n from the front of *Open*.
    2. If n is a goal node
        1. stop and return it and the path from the initial node to *n*.

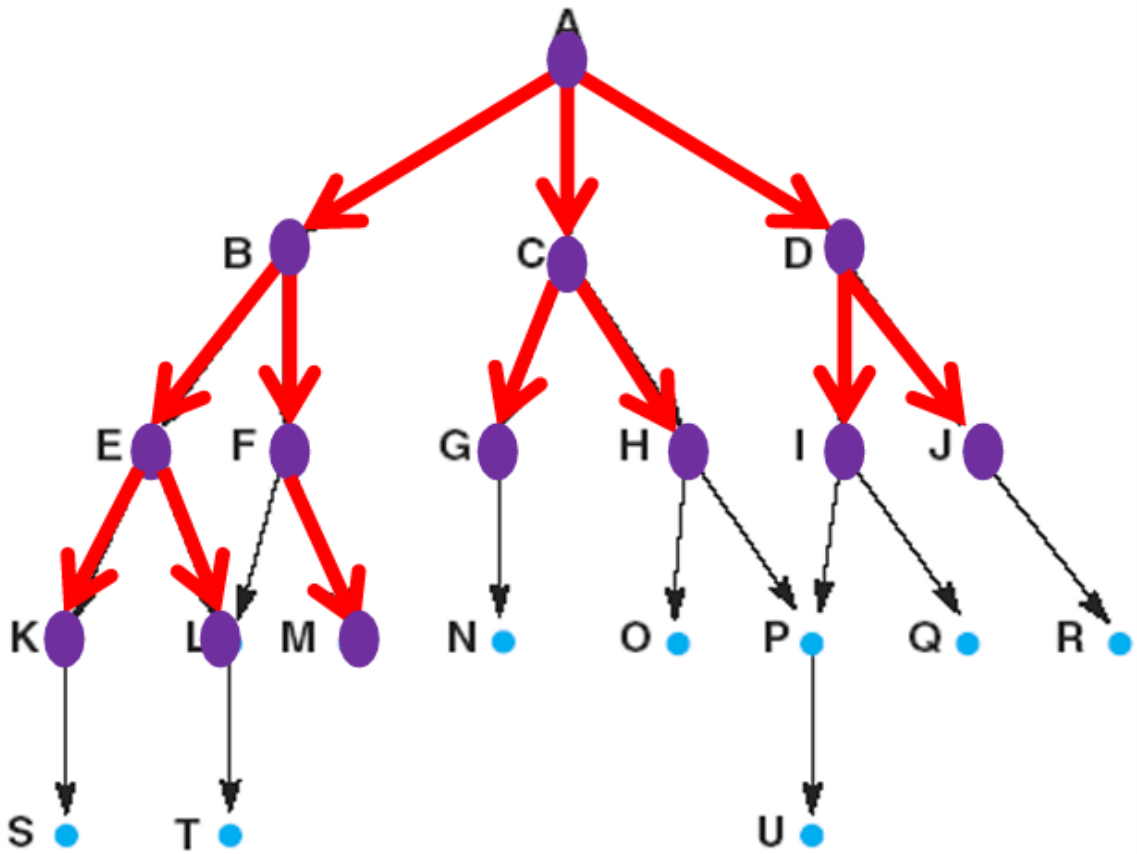
        Else
        1. remove *n* from *Open*
        2. add *n* to *Closed*
        3. get all *n*'s children
        4. discard *n*'s children that are in the *Closed* or *Open* lists.
        5. add the remaining children to the end of *Open* labelling each with its path from the initial node.

    End while

    return *failure*

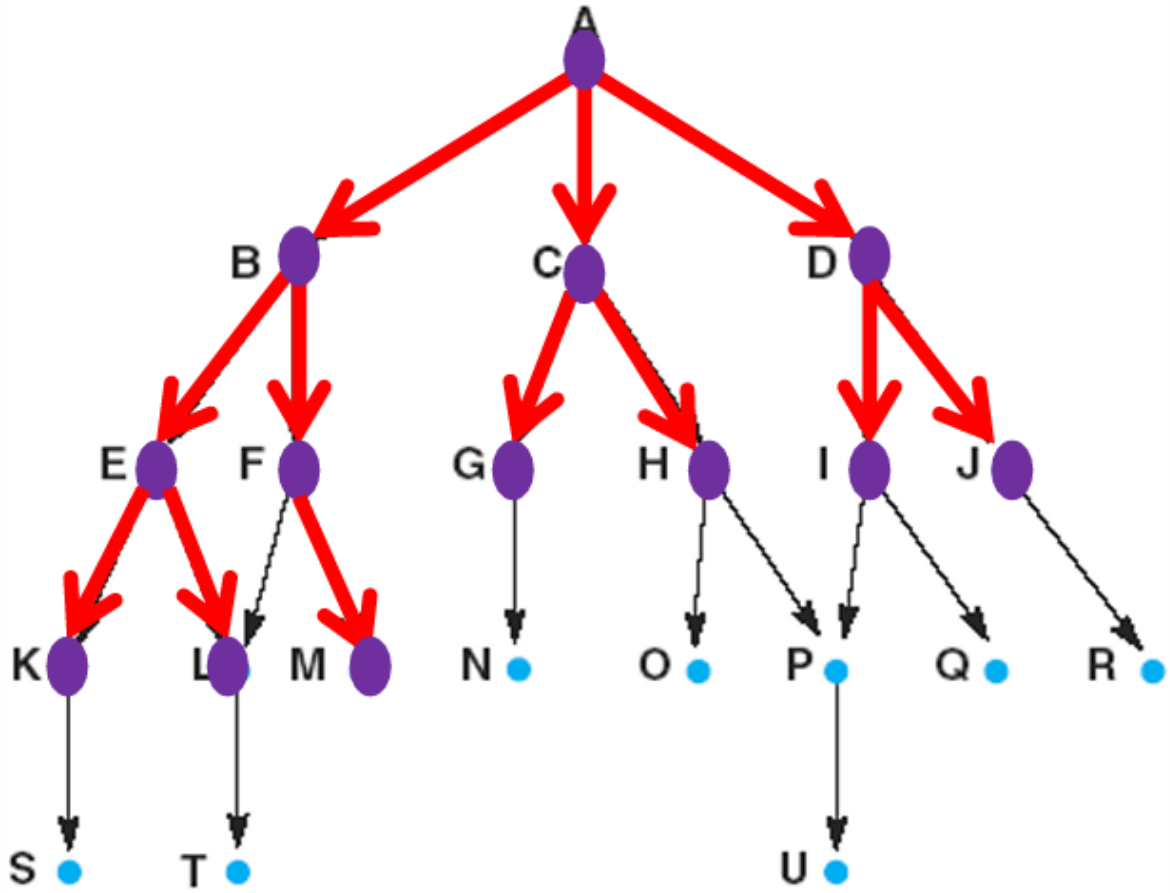
Breadth-First Search in a Graph

Iteratn	n	Open	Closed
0		[A]	[]
1	A	[B,C,D]	[A]
2	B	[C,D,E,F]	[B,A]
3	C	[D,E,F,G,H]	[C,B,A]
4	D	[E,F,G,H,I,J]	[D,C,B,A]
5	E	[F,G,H,I,J,K,L]	[E,D,C,B,A]
6	F	[G,H,I,J,K,L,M]	[F,E,D,C,B,A]



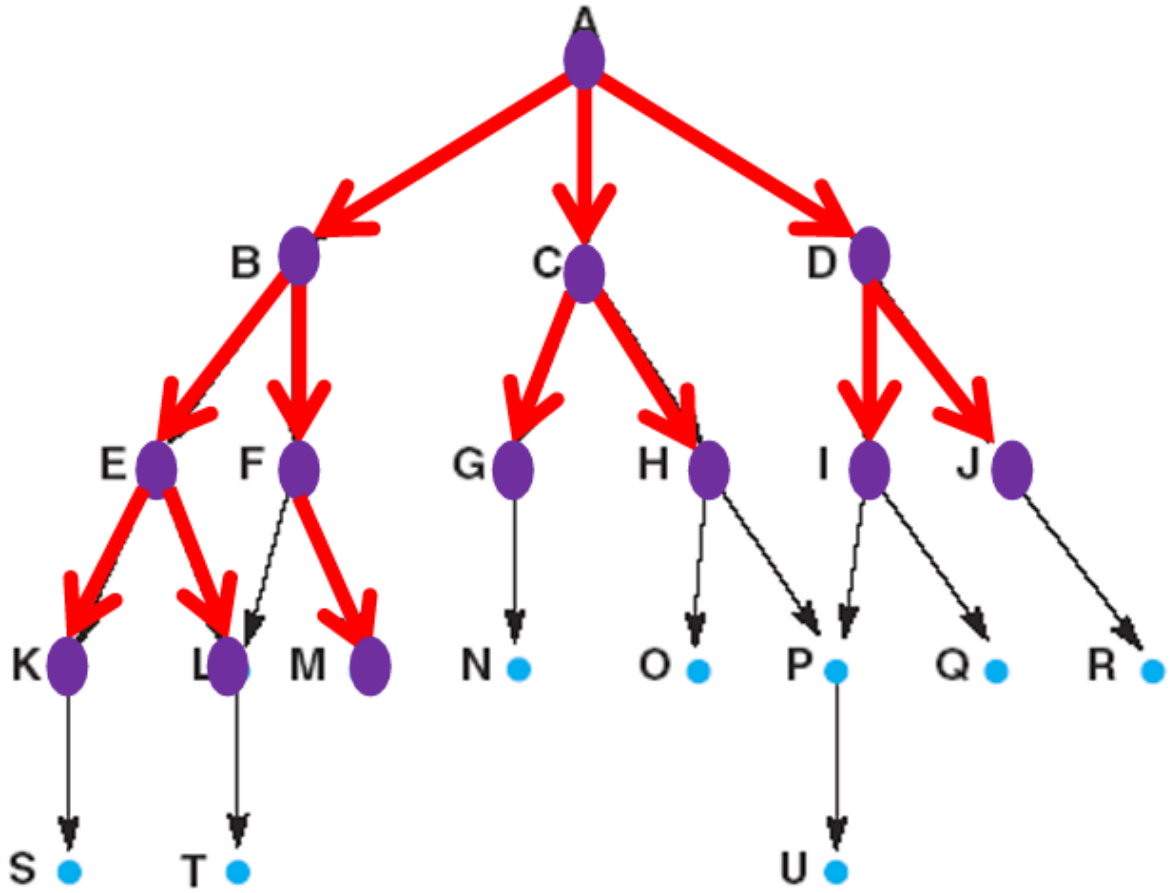
BFS Using Lists

```
graph = [
      ['A', ['B', 'C', 'D']],
      ['B', ['E', 'F']],
      ['C', ['G', 'H']],
      ['D', ['I', 'J']],
      ['E', ['K', 'L']],
      ['F', ['M']],
      ['G', ['N']],
      ['H', ['O']],
      ['I', ['P']],
      ['J', ['Q', 'R']],
      ['K', ['S']],
      ['L', ['T']],
      ['P', ['U']]
]
```



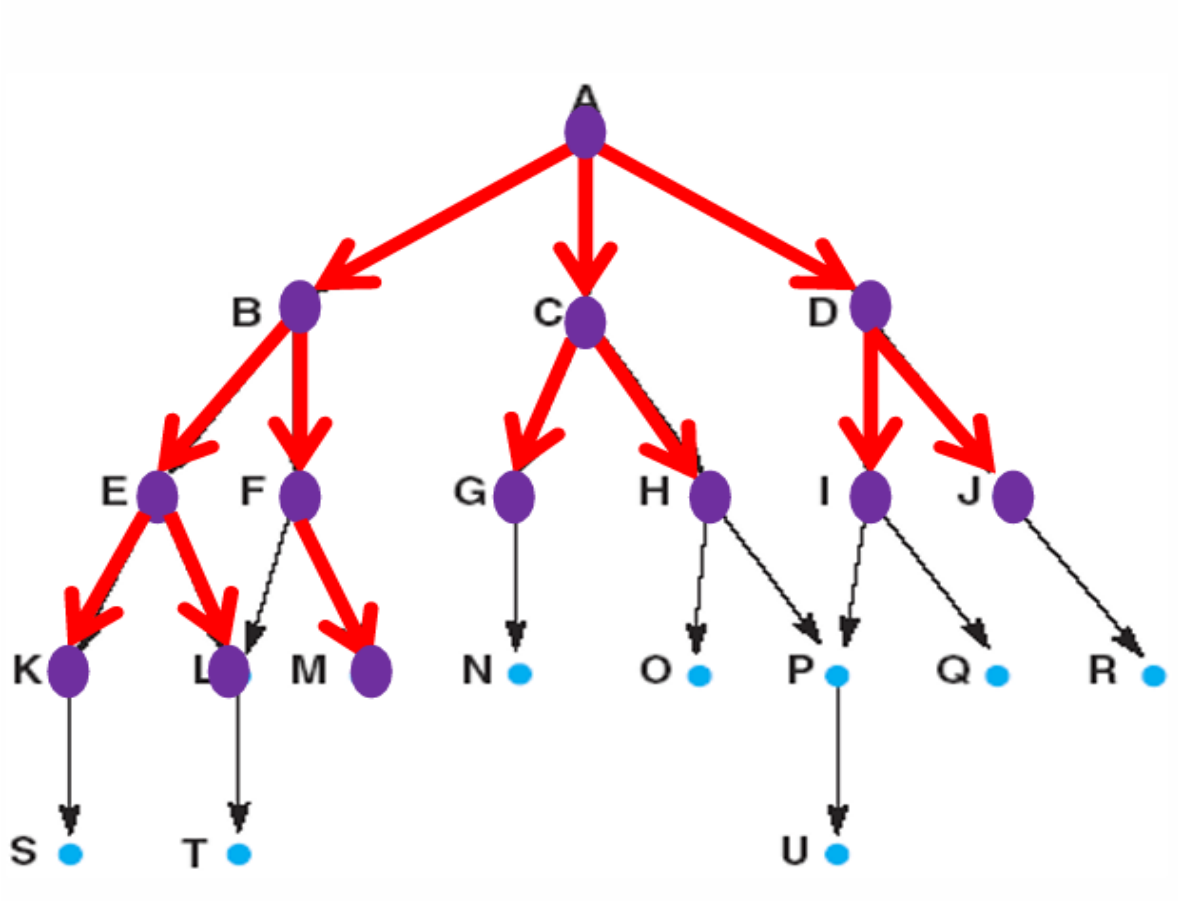
BFS Using Lists

```
graph = {
      'A': ['B', 'C', 'D'],
      'B': ['E', 'F'],
      'C': ['G', 'H'],
      'D': ['I', 'J'],
      'E': ['K', 'L'],
      'F': ['M'],
      'G': ['N'],
      'H': ['O'],
      'I': ['P'],
      'J': ['Q', 'R'],
      'K': ['S'],
      'L': ['T'],
      'P': ['U']
}
```



BFS Using Lists

```
graph_dict_set = {
      'A': {'B', 'C', 'D'},
      'B': {'E', 'F'},
      'C': {'G', 'H'},
      'D': {'I', 'J'},
      'E': {'K', 'L'},
      'F': {'M'},
      'G': {'N'},
      'H': {'O'},
      'I': {'P'},
      'J': {'Q', 'R'},
      'K': {'S'},
      'L': {'T'},
      'P': {'U'}
}
```



BFS Algorithm

```
1. Initialize: OPEN = [Start], CLOSED = [ ]
2. While OPEN ≠ Ø do
        n = first node in OPEN
        If n is Goal then
              return path(Start → n)
        Else
              remove n from OPEN
              add n to CLOSED
              for each child of n do
                  if child not in OPEN and not in CLOSED then
                        add child to end of OPEN
3. End while
4. return Failure
```

BFS Algorithm Simple

```
1. Initialize:
      OPEN ← [Start]
      CLOSED ← [ ]
2. While OPEN is not empty do
      n ← first element of OPEN
      remove n from OPEN
      add n to CLOSED
      print n, OPEN, CLOSED
      If n = Goal then
            print "Goal found" and stop
      Else
            For each child of n in graph do
                  If child not in OPEN and not in CLOSED
                        add child to end of OPEN
      End if
    End while
3. If OPEN becomes empty and Goal not found then
      print "Failure: Goal not found"
4. Print final state of OPEN and CLOSED
```

```
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['E', 'F'],
    'C': ['G', 'H'],
    'D': ['I', 'J'],
    'E': ['K', 'L'],
    'F': ['M'],
    'G': ['N'],
    'H': ['O'],
    'I': ['P'],
    'J': ['Q', 'R']
}
```

Step 0: Represent Inputs

Define the graph, start node, and goal node.

```
graph = [
    ['A', ['B', 'C', 'D']],
    ['B', ['E', 'F']],
    ['C', ['G', 'H']],
    ['D', ['I', 'J']],
    ['E', ['K', 'L']],
    ['F', ['M']],
    ['G', ['N']],
    ['H', ['O']],
    ['I', ['P']],
    ['J', ['Q', 'R']]
]
start = input("Enter start node: ")
goal = input("Enter goal node: ")
```