CS663/362 Artificial Intelligence
Course Instructor: Pratik Shah

- **Group Name : Alpha Elites**
- **Group Members**

No	Student I'D	Student Name
1.	202051021	Aman Kumar
2.	202051022	Aman Singh
3.	202051057	Dagli Kavan Hemantkumar
4.	202051100	Kanani Darpan Ashokbhai

Lab Assignment - 1

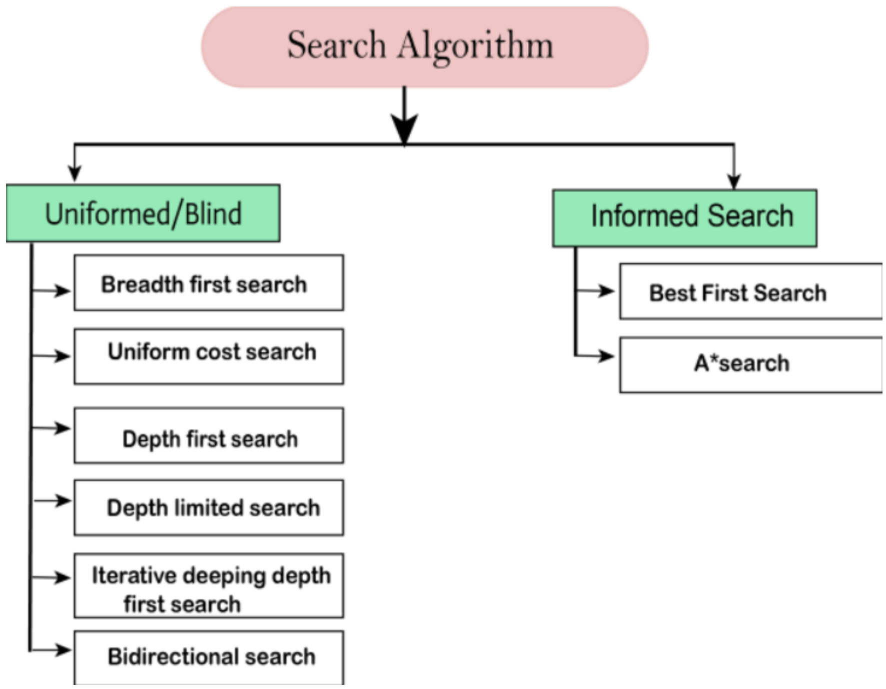
Un informed Search Algorithms

- It is a class of general-purpose search algorithms which operates in brute force way. It does not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

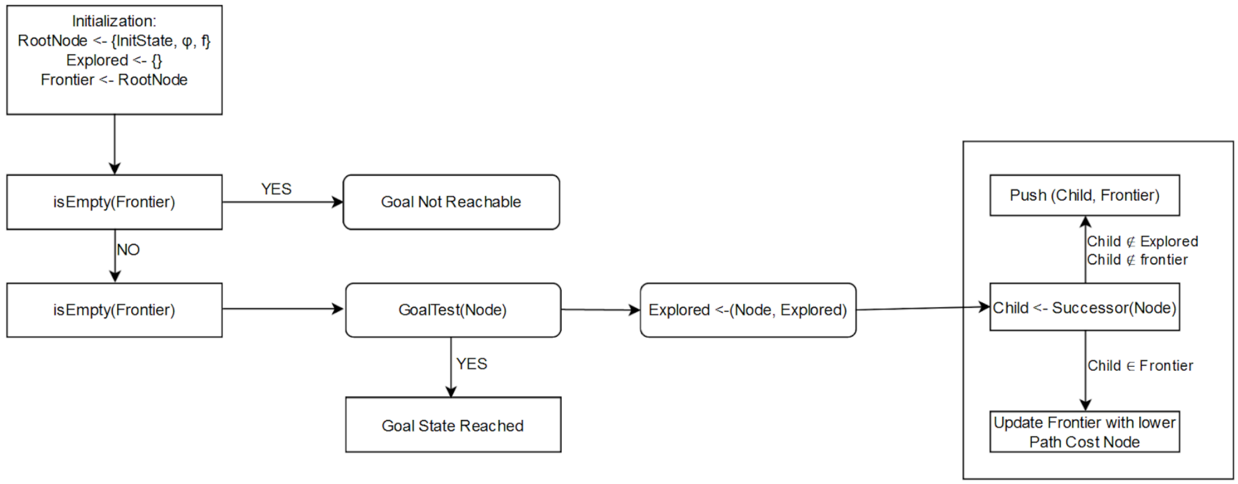
Informed Search Algorithms

- In an informed search algorithm, It stores knowledge such as how far we are from the goal, path cost, how to reach the goal in the form of set, array or table. This knowledge helps agents to explore less of the search space and find the goal node more efficiently.
- The informed search algorithm is more useful for large search space. Informed search algorithms uses the idea of heuristic, so it is also called Heuristic search. Ex. N*N -1 Puzzle
- Heuristics function: It is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and

produces the estimation of how close the agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.



A. Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.



Here is a pseudocode for a graph search agent:

```
# we have the graph in which we want to search, start / root and goal node.
function graph_search(graph, start, goal):
  # initialize an empty list to store the path
  path = []
  # initialize a queue to store the nodes that will be explored
  queue = []
  # add the start node to the queue
  queue.append(start)
  # initialize a set to store the visited nodes
  visited = set()

  # loop until the queue is empty
  while queue:
    # remove the first node in the queue and mark it as visited
    current_node = queue.pop(0)
    visited.add(current_node)
    # if the current node is the goal, return the path
    if current_node == goal:
      return path
    # get the neighbors of the current node
    neighbors = graph[current_node]
    # loop through the neighbors
    for neighbor in neighbors:
   # if the neighbor has not been visited, add it to the queue and update the path
```

```
            if neighbor not in visited:
                queue.append(neighbor)
                path.append((current_node, neighbor))
    # if the queue is empty and the goal has not been reached, return "Goal not found"
    return "Goal not found”
```

B. Write a collection of functions imitating the environment for Puzzle-8.

```
# initialize the puzzle with a given state
def initialize_puzzle(state):
    # state is a list of integers representing the puzzle, with 0 representing the empty square
    # for example, state = [1, 2, 3, 4, 5, 6, 7, 8, 0] represents the puzzle in the solved state
    # store the state in a global variable
    global puzzle
    puzzle = state

# return the current state of the puzzle
def get_state():
    return puzzle

# return a list of possible actions given the current state
def get_actions():
    # get the index of the empty square
    empty_index = puzzle.index(0)
    # initialize a list of actions
    actions = []
    # if the empty square is not in the first column, the "left" action is possible
    if empty_index % 3 > 0:
        actions.append("left")
    # if the empty square is not in the last column, the "right" action is possible
    if empty_index % 3 < 2:
        actions.append("right")
    # if the empty square is not in the first row, the "up" action is possible
    if empty_index > 2:
        actions.append("up")
    # if the empty square is not in the last row, the "down" action is possible
    if empty_index < 6:
        actions.append("down")
```

```
    return actions

# apply an action to the puzzle and return the resulting state
def apply_action(action):
    # get the index of the empty square
    empty_index = puzzle.index(0)
    # initialize a new state as a copy of the current state
    new_state = puzzle.copy()
     # perform the action by swapping the empty square with the appropriate neighboring square
    if action == "left":
          new_state[empty_index], new_state[empty_index - 1] = new_state[empty_index - 1], new_state[empty_index]
    elif action == "right":
         new_state[empty_index], new_state[empty_index + 1] = new_state[empty_index + 1], new_state[empty_index]
    elif action == "up":
          new_state[empty_index], new_state[empty_index - 3] = new_state[empty_index - 3], new_state[empty_index]
    elif action == "down":
         new_state[empty_index], new_state[empty_index + 3] = new_state[empty_index + 3], new_state[empty_index]
    # update the puzzle with the new state
    initialize_puzzle(new_state)
    # return the new state
    return new_state

# check if the puzzle is in the solved state
def is_solved():
    return puzzle == [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

C. Describe what is Iterative Deepening Search.

BFS: consumes more memory and less time

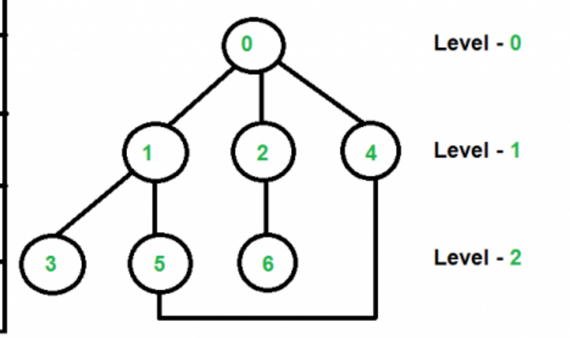
DFS: consumes less memory and more time

Iterative Deepening Search (IDS) is an iterative graph searching strategy that takes advantage of the completeness of the Breadth-First Search (BFS) strategy but uses much less memory in each iteration.

IDS achieves the desired completeness by enforcing a depth-limit on DFS that mitigates the possibility of getting stuck in an infinite or a very long branch.

It searches each branch of a node from left to right until it reaches the required depth. Once it has, IDS goes back to the root node and explores a different branch that is similar to DFS.

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



Time complexity is: $O(b^d)$
Space complexity is: $O(bd)$

When to use iterative deepening:

As a general rule of thumb, we use iterative deepening when we do not know the depth of our solution and have to search a very large state space.

Iterative deepening may also be used as a slightly slower substitute for BFS if we are constrained by memory or space.

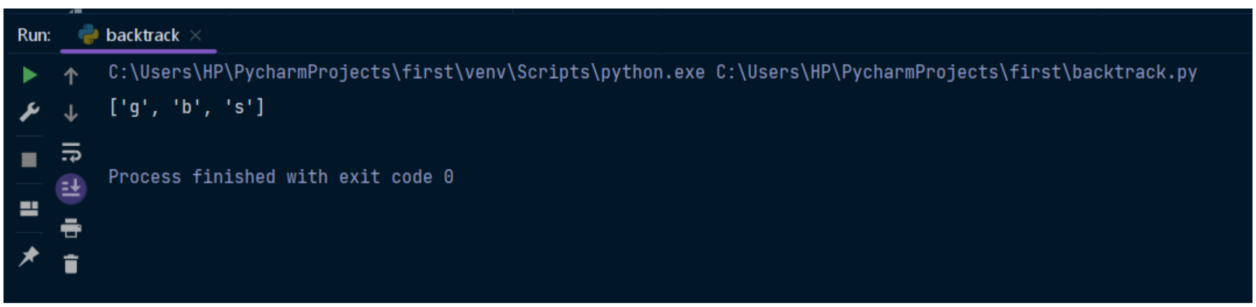
D. Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.

```
path = [('s', 'a'), ('s', 'b'), ('b', 'g')]

def func(startState, finalState, path):
   curr = path.pop()[0]
   finalPath = [finalState, curr]
   while curr != startState:
       for i in path:
           if i[1] == curr:
               curr = i[0]
               finalPath.append(curr)
               break
   return finalPath

print(func('s', 'g', path))
```

Output:



E. Generate Puzzle-8 instances with the goal state at depth “d”.

```
1 2 3
5 6 0
7 8 4
```

At depth 1 following are the instances:

```
1 2 3
5 0 6
7 8 4

1 2 0
```

```
5 6 3
7 8 4

1 2 3
5 6 4
7 8 0
```

To generate Puzzle-8 instances with the goal state at depth "d", we can use the following approach:

1. Initialize the puzzle with the solved state [1, 2, 3, 4, 5, 6, 7, 8, 0].
2. Randomly apply a series of actions to the puzzle, ensuring that the empty square is not moved out of the grid. The number of actions should be equal to "d".
3. Return the resulting puzzle state.

```
import random
def generate_puzzle(d):
    # initialize the puzzle with the solved state
    initialize_puzzle([1, 2, 3, 4, 5, 6, 7, 8, 0])
    # apply a series of random actions to the puzzle
    for i in range(d):
        actions = get_actions()
        action = random.choice(actions)
        apply_action(action)
    # return the resulting puzzle state
    return get_state()
```

F. Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

The space requirement for BFS search agent is $O(b^d)$
The time requirement for BFS search agent is $O(b^d)$
b - branch factor
d - depth factor