Lab 3 Problem Formulation

1.1 Representing Search Problem:

The problem-solving agent performs precisely by defining problems and its several solutions. The main components of problem formulation are as follows:

- **Goal Formulation:** It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure.
- **Initial State:** It is the starting state or initial step of the agent towards its goal.
- **Actions:** It is the description of the possible actions available to the agent.
- **Transition Model:** It describes what each action does.
- **Goal Test:** It determines if the given state is a goal state.
- **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**

1.1.1 Explicit Representation of the Search Graph:

The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed). An **explicit graph** consists of

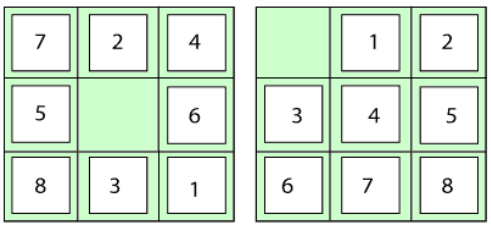
- a list or set of nodes
- a list or set of arcs
- a start node
- a list or set of goal nodes

To define a search problem, we need to define the start node, the goal predicate, and the neighbors function.

1.1.2 Example Problems

a) **8 Puzzle Problem:**

Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space. The objective is to reach a specified goal state similar to the goal state, as shown in the below figure. In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.



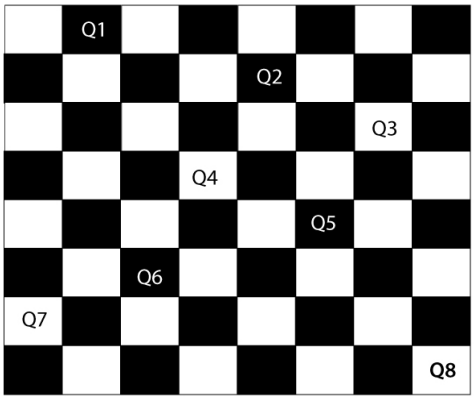
Start State Goal State

In the above figure, our task is to convert the current (Start) state into goal state by sliding digits into the blank space.

The problem formulation is as follows:

- **States:** It describes the location of each numbered tiles and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
- **Transition Model:** It returns the resulting state as per the given state and actions.
- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

b) **8-queens problem:**
The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.** From the following figure, we can understand the problem as well as its correct solution.



It is noticed from the above figure that each queen is set into the chessboard in a position where no other queen is placed diagonally, in same row or column. Therefore, it is one right approach to the 8-queens problem. For this problem, there are two main kinds of formulation:

- **Incremental formulation:** It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

- **States:** Arrangement of any 0 to 8 queens on the chessboard.
- **Initial State:** An empty chessboard
- **Actions:** Add a queen to any empty box.
- **Transition model:** Returns the chessboard with the queen added in a box.
- **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.

- **Path cost:** There is no need for path cost because only final states are counted.

In this formulation, there is approximately **1.8×10^{14}** possible sequence to investigate.

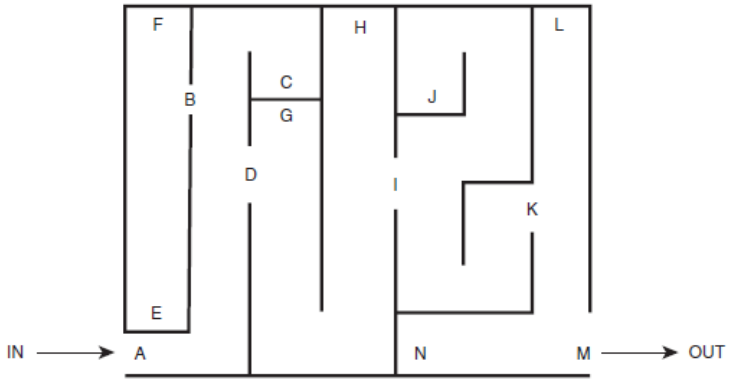
- **Complete-state formulation:** It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

Following steps are involved in this formulation

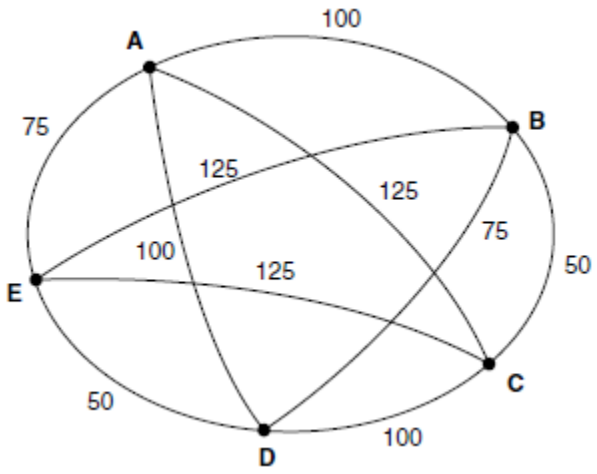
- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.

This formulation is better than the incremental formulation as it reduces the state space from **1.8×10^{14}** to **2057**, and it is easy to find the solutions.

c) **Traversing a Maze:**
When traversing a maze, most people will wander randomly, hoping they will eventually find the exit. This approach will usually be successful eventually but is not the most rational and often leads to what we call "going round in circles." This problem, of course, relates to search spaces that contain loops, and it can be avoided by converting the search space into a search tree. An alternative method that many people know for traversing a maze is to start with your hand on the left side of the maze (or the right side, if you prefer) and to follow the maze around, always keeping your left hand on the left edge of the maze wall. In this way, you are guaranteed to find the exit.



d) **Traveling salesperson problem (TSP):**
It is a **touring problem** where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city. Suppose a salesperson has five cities to visit and then must return home. The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and then returning to the starting city. The nodes of the graph represent cities, and each arc is labeled with a weight indicating the cost of traveling that arc. This cost might be a representation of the miles necessary in car travel or cost of an air flight between the two cities.


A
B
C
D
E
100
75
125
125
75
100
125
50
50
100

1.1.3 Code Structure:

```
#
class Problem:
    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
    __init__, goal_test, and path_cost. Then you will create instances
    of your subclass and solve them with the various search functions."""

    def __init__(self, initial, goal=None):
        """The constructor specifies the initial state, and possibly a goal
        state, if there is a unique goal. Your subclass's constructor can add
        other arguments."""
        self.initial = initial
        self.goal = goal

    def actions(self, state):
        """Return the actions that can be executed in the given
        state. The result would typically be a list, but if there are
        many actions, consider yielding them one at a time in an
        iterator, rather than building them all at once."""
        raise NotImplementedError

    def result(self, state, action):
        """Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state)."""
        raise NotImplementedError

    def goal_test(self, state):
        """Return True if the state is a goal. The default method compares the
        state to self.goal or checks for state in self.goal if it is a
        list, as specified in the constructor. Override this method if
        checking against a single self.goal is not enough."""
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Return the cost of a solution path that arrives at state2 from
        state1 via action, assuming cost c to get up to state1. If the problem
        is such that the path doesn't matter, this function will only look at
        state2. If the path does matter, it will consider c and maybe state1
        and action. The default method costs 1 for every step in the path."""
        return c + 1

    def value(self, state):
        """For optimization problems, each state has a value.
        Some algorithms try to maximize this value."""
        raise NotImplementedError
#
```

1.2 Lab Tasks

Exercise 3.1.

In this task you will consider 8-puzzle problem. The 8-puzzle problem is a puzzle played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in specified order. You are permitted to slide blocks horizontally or vertically into the blank square.

3	4	1
2	5	6
7	8	

Initial State (Randomly Chosen)

1	2	3
4	5	6
7	8	

Goal State

Write a method "Play" which allows the user to play the i.e by moving the blank space in different positions on board. Print the state after each move and show a message if the goal state has been reached.

You output should look something like this:

```
Initial configuration
1-e-2
6-3-4
7-5-8

After moving 2 into the empty space
1-2-e
6-3-4
7-5-8

After moving 4 into the empty space
1-2-4
6-3-e
7-5-8

After moving 3 into the empty space
1-2-4
6-e-3
7-5-8

After moving 6 into the empty space
1-2-4
e-6-3
7-5-8
```

Exercise 3.2

Write a class for 8-queens game and a method to play the game manually.

Exercise 3.3

Write a class for traversing the maze game and a method to play the game manually.

Exercise 3.4

Write a class for TSP game and a method to play the game manually.