**Session-3**

**Solving Problems by Searching**

**Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation.

**Formal Description of the problem:**

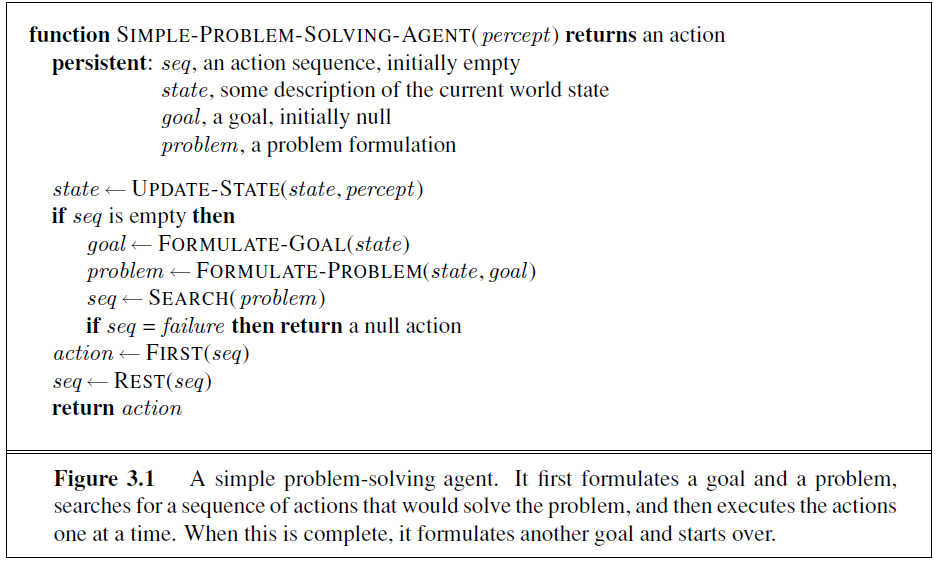
Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

* + **Search Space:** Search space represents a set of possible solutions, which a system may have.
  + **Start State:** It is a state from where agent begins **the search**.
  + **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
  + **Actions:** set of rules that describe the actions (operations) available.

**Well-defined problems and solutions:**

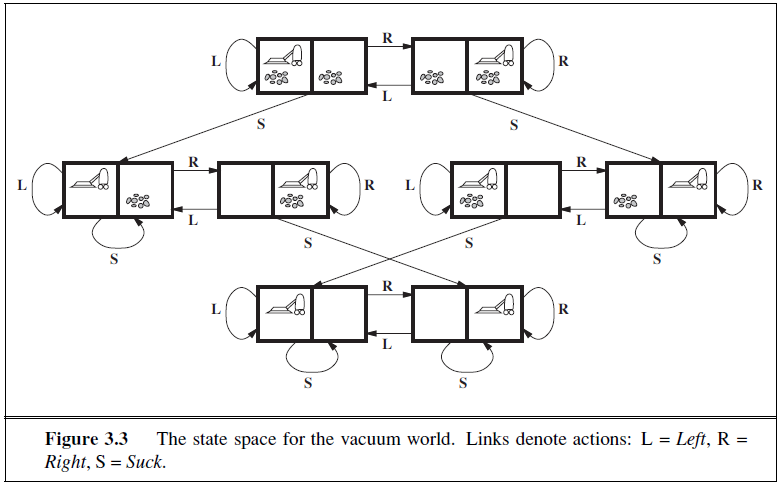
A problem can be defined formally by five components:

* **INITIAL STATE** : The initial state that the agent starts in
* **ACTIONS:** A description of the possible actions available to the agent given a particular state *s,* ACTIONS(s) returns the set of actions that can be executed in *s.*
* **Transition Model:** That represents a description of what each action does.
  + **Successor :** Any state reachable from a given state by a single action.
  + **State Space :** Initial State + Actions + Transition Model.
  + **Graph :** A directed Graph in which the nodes are states and the links between nodes are actions.
* **Goal Test :** Determines whether a given state is a goal state or not.
* **Path Cost :** Its a function that assigns a numeric cost to each path. The problem solving agent chooses a cost function that reflects its own performance measure.
  + **Step Cost :** Taking step ‘a’ in state ‘s’ to reach state ‘s1’. The function can be denoted by **C(s, a, s1).**



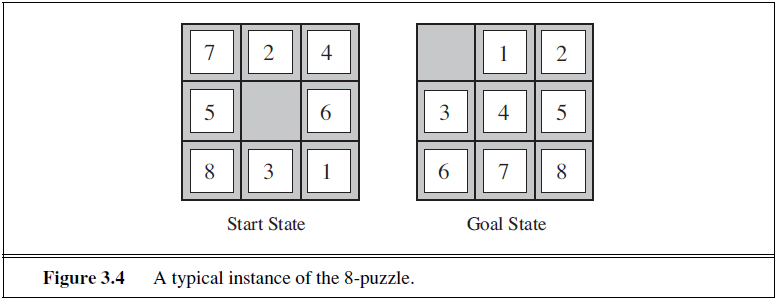
**Example Problems:**

* 1. **Toy Problem(Vacuum Cleaner):**
* **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × 22 = 8 possible world states. A larger environment with n locations has n x 2n states.
* **Initial state:** Any state can be designated as the initial state.
* **Actions:** *Left, Right, and Suck*
* **Transition model:** The complete state space is shown in Figure 3.3.
* **Goal test:** This checks whether all the squares are clean.
* **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

****

* 1. **8-Puzzle Problem:**

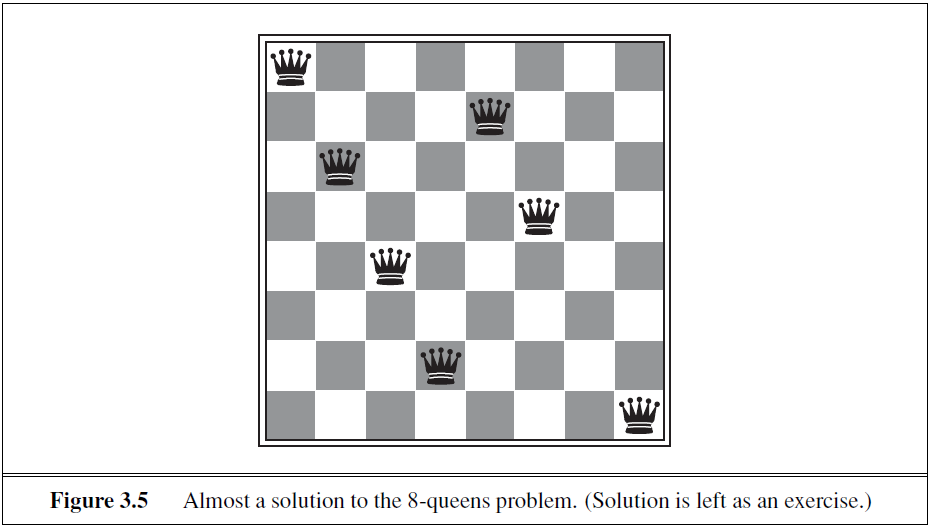
**The below figure 3.4 illustrate 8-puzzle problem and solution to it.**

****

* **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
* **Initial state:** Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
* **Actions:** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
* **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
* **Goal test:** This checks whether the state matches the goal configuration shown in Figure 3.4

**• Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

* 1. **8-Queens Problem:** The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.)

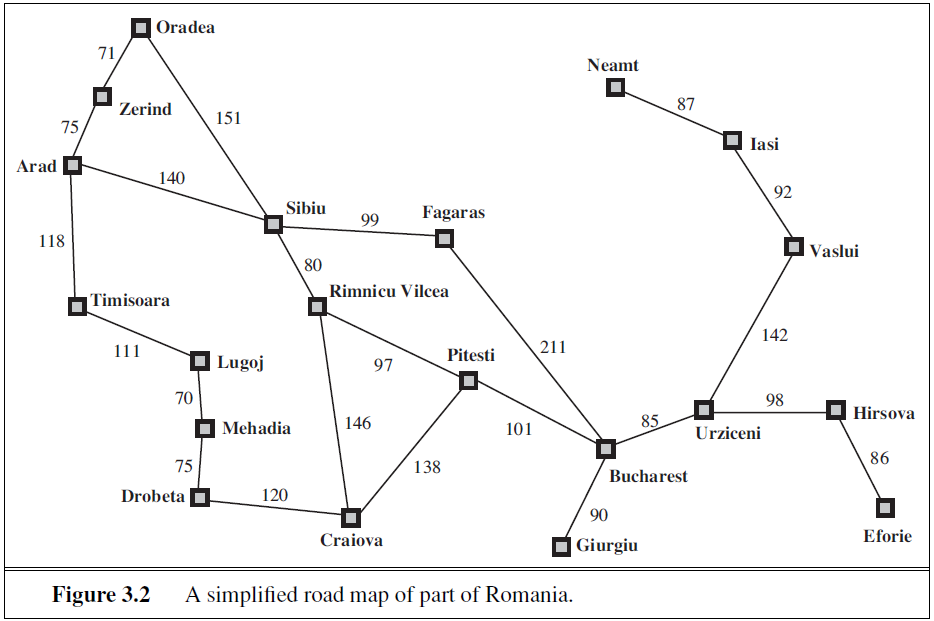
****

* **States:** Any arrangement of 0 to 8 queens on the board is a state. All possible arrangements of n queens (0 ≤ n ≤ 8), one per column in the leftmost n columns, with no queen attacking another.
* **Initial state:** No queens on the board.
* **Actions:** Add a queen to any empty square. Such that it is not attacked by any other queen.
* **Transition model**: Returns the board with a queen added to the specified square.
* **Goal test:** 8 queens are on the board, none attacked.

1. **Route-Finding Problem (Real World Problem):** The **route-finding problem** is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example.

* **States:** Each state obviously includes a location (e.g., an airport) and the current time.
* **Initial state:** This is specified by the user’s query.
* **Actions:** Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
* **Transition model:** The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
* **Goal test:** Are we at the final destination specified by the user?
* **Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

**Eg.** The goal is to reach Bucharest from Arad as shown in Figure 3.2.



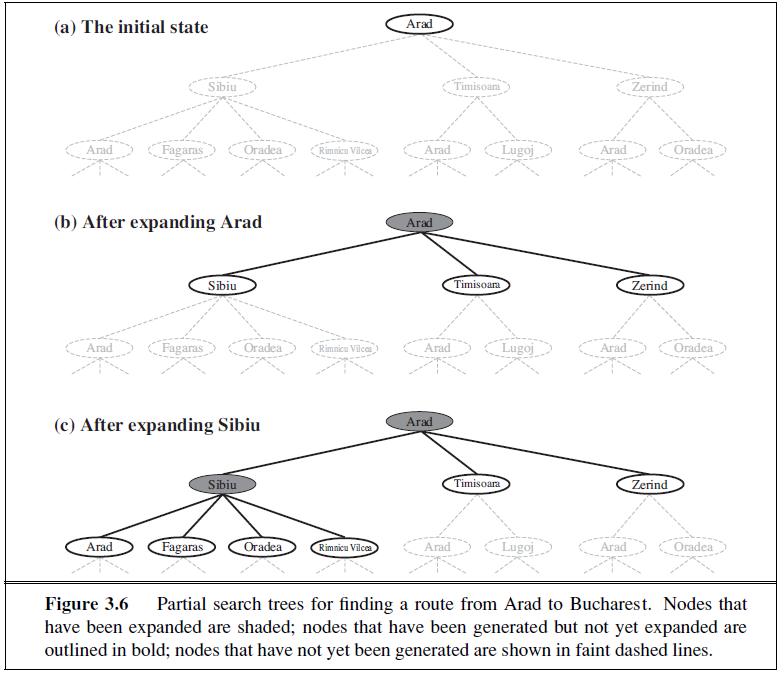
The solution is the sequence of cities: Arad, Sibiu, Fagaras, and Bucharest with path cost of 450.

**SEARCHING FOR SOLUTIONS**

A solution is an action sequence, so search algorithms work by considering various possible action sequences. It can be implemented using **search tree.**

A **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem. Figure 3.6 shows the first few steps in growing the search tree for finding a route from Arad to Bucharest.

We start at Arad and then suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get In (Arad), In (Fagaras), In (Oradea), and In (Rimnicu Vilcea). We can then choose any of these four or go back and choose Timisoara or Zerind. Each of these six nodes is a **leaf node**, that is, a node with no children in the tree. The set of all leaf nodes available for expansion at any given point is called the **frontier (Open List)**.

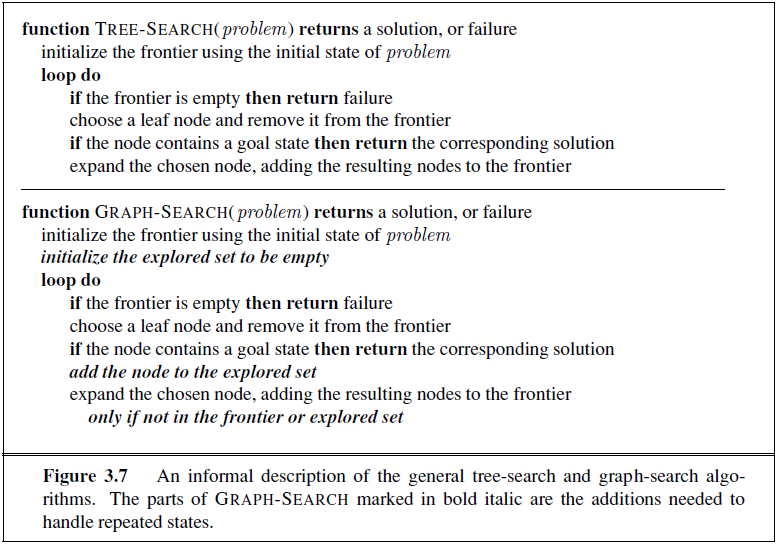


The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. The general TREE-SEARCH algorithm is shown informally in Figure 3.7. Search algorithms all share this basic structure; they vary primarily

according to how they choose which state to expand next—the so-called **search strategy**.

The search tree shown in Figure 3.6: it includes the path from Arad to Sibiu and back to Arad again! We say that In (Arad) is a **repeated state** in the search tree, generated in this case by a **loopy path**. Considering such loopy paths means that the complete search tree for Romania is *infinite* because there is no limit to how often one can traverse a loop.

The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (**closed list**), which remembers every expanded node.

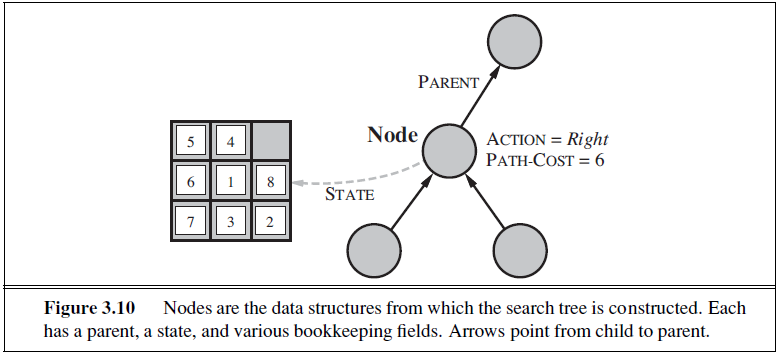
****

**Infrastructure for search algorithms:**

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four Components:

* ***n.STATE:*** the state in the state space to which the node corresponds
* **n.PARENT:** the node in the search tree that generated this node
* **n.Action:** the action that was applied to the parent to generate the node
* **n.PATH-COST:** the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

The node data structure is depicted in Figure 3.10. Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:



Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action

and returns the resulting child node:

**function** CHILD-NODE(problem, parent , action) **returns** a node

**return** a node with

STATE = problem.RESULT(parent.STATE, action),

PARENT = parent, ACTION = action,

PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)

These pointers also allow the solution path to be extracted when a goal node is found; we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.

**Measuring problem-solving performance:**

**Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

**Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

**Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.

**Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.