Session 3: Problem Solving

Course Title: Computational Intelligence

Course Code: CSC401A

Vaishali R Kulkarni

Department of Computer Science and Engineering
Faculty of Engineering and Technology
M. S. Ramaiah University of Applied Sciences
Email: *vaishali.cs*.et@msruas.ac.in

Basic Search Strategies

- **Problem formulation** is the process of deciding what actions and states to consider, given a goal. A problem can be defined formally by five components:

1. **The initial state**
2. **Actions**
3. **Transition model**
4. **Goal test**
5. **Path cost**

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions

EXAMPLE PROBLEMS

- The problem-solving approach has been applied to many task environments. They are distinguished as **toy and real-world problems.**
- **A toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.
- **A real-world problem** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but can be described in a general manner.

Toy Problem: Vacuum Cleaner

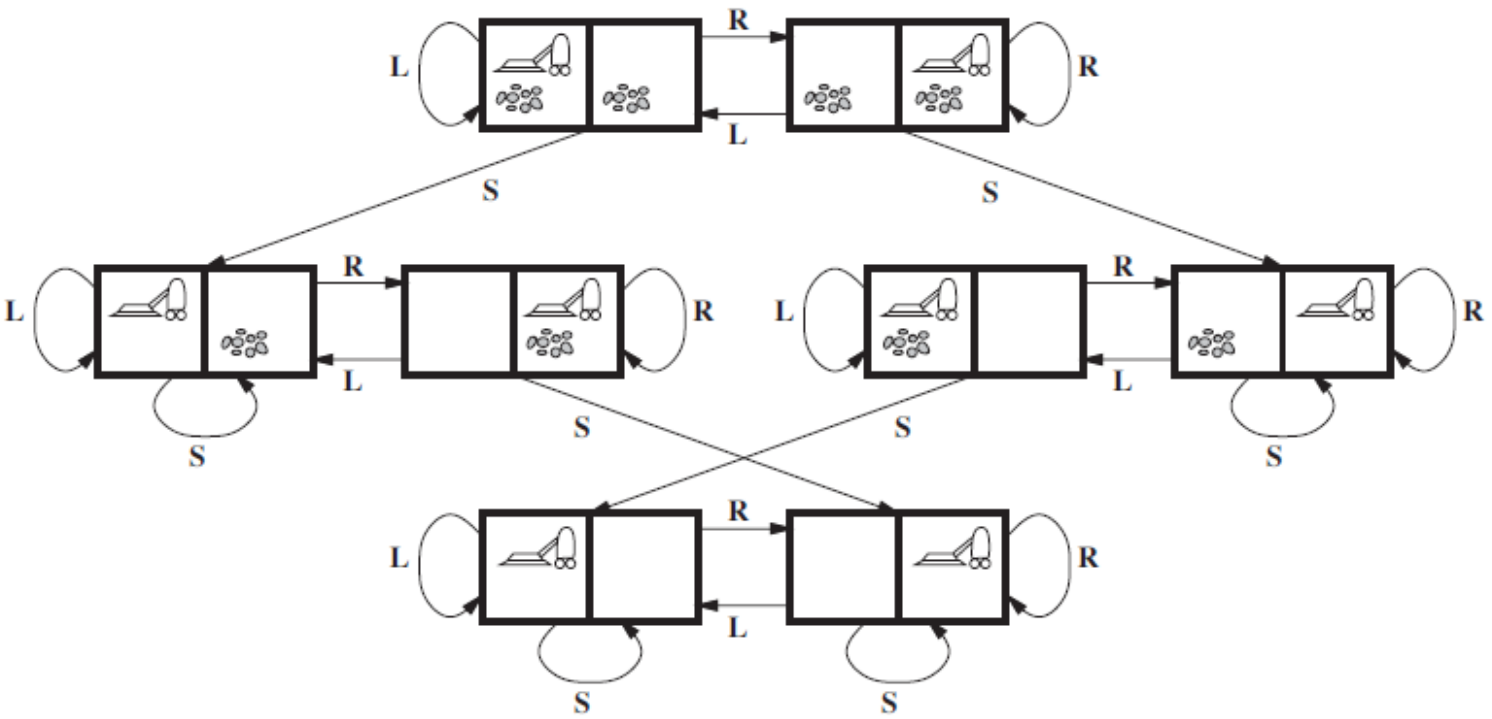


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Toy Problem: Vacuum Cleaner

The vacuum world can be formulated as a problem as follows:

1. 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 $2 \times 22 = 8$ possible world states. A larger environment with n locations has $n \cdot 2n$ states.
2. Initial state: Any state can be designated as the initial state.
3. Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
4. Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
5. Goal test: This checks whether all the squares are clean.
6. Path cost: Each step costs 1, so the path cost is the number of steps in the path.

8-Puzzle

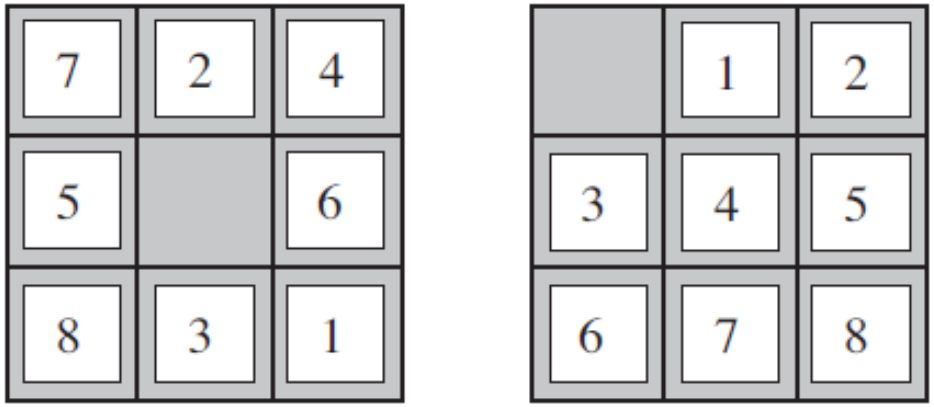


Figure 3.4 A typical instance of the 8-puzzle.

Toy Problem: 8-Puzzle

- **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;
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Real-world problems

- We have already seen how 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.
- Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.

Airline travel problems Web site

- **States:** Each state includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- **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.

Real-world problems

- Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem “Visit every city at least once, starting and ending in Bangalore.” As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the set of cities the agent has visited.
- So the initial state would be In(Bangalore), Visited({Bangalore}), a typical intermediate state would be In(Chennai), Visited({Bangalore , Hydrabad , Chennai}), and the goal test would check whether the agent is in Bangalore and all 20 cities have been visited.
- The traveling salesperson problem (TSP) is s a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

Real-world problems

- A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**.
- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.
- Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Real-world problems

- **Robot navigation** is a generalization of the route-finding problem. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional.
- Advanced techniques are required just to make the search space finite. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

Real-world problems

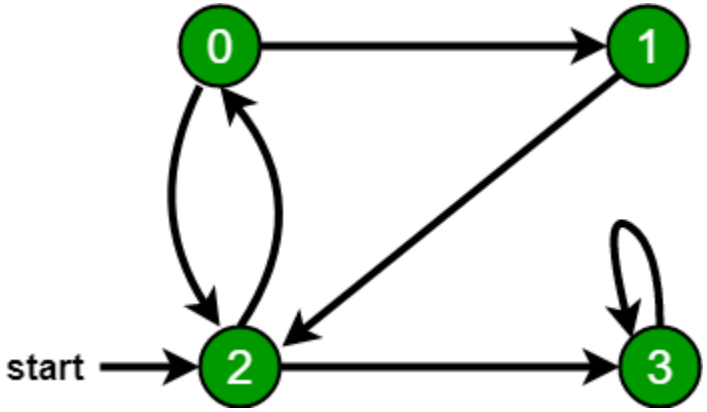
- **Automatic assembly sequencing of complex objects by a robot** was first demonstrated in 1972. Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space.
- Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a 3-dimensional protein with the right properties to cure some disease.

Uninformed search

- **Informed Search:** Informed Search algorithms have information on the goal state which helps in more efficient searching. This information is obtained by a function that estimates how close a state is to the goal state.
  Example: Greedy Search and Graph Search
- **Uninformed Search:** Uninformed search algorithms have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and length of actions.
  Examples: Depth First Search and Breadth-First Search

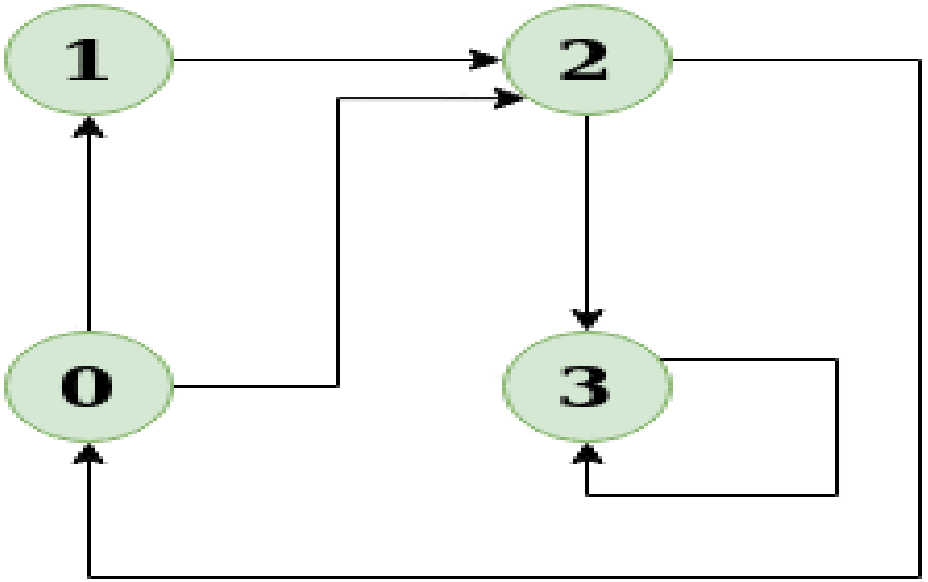
BFS for a Graph

- For example, in the following graph, If we start traversal from vertex 2. then BFS Traversal of the following graph is 2, 0, 3, 1.



DFS for a graph

- ***Output:*** *DFS from vertex 1 : 1 2 0 3*



BFS verses DFS

Sr. No.	Key	BFS	DFS
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitablity for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

BFS Pseudocode

BFS (G, s) //Where G is the graph and s is the source node

let Q be queue.

Q.enqueue(s) //Inserting s in queue until all its neighbor vertices are marked.

mark s as visited.

while (Q is not empty)

//Removing that vertex from queue, whose neighbor will be visited now

v = Q.dequeue( )

//processing all the neighbors of v

for all neighbors w of v in Graph G

if w is not visited

Q.enqueue(w) //Stores w in Q to further visit its neighbor

mark w as visited..

References

- Artificial Intelligence A Modern Approach Third Edition by Stuart J. Russell and Peter Norvig