Artificial Intelligence

Lecture 3: Solving Problems by Searching (Review Before Basic Algorithms)

AI 1

What is Search?

Definition

Search is an **enumeration of a set of potential partial solutions** to a problem so that they can be checked to see if they truly are solutions, or could lead to a solution.

To carry out a search, we need:

- A definition of a potential solution.
- A method of generating the potential solutions (hopefully in a clever way).
- A way to check whether a potential solution is a solution.

Search Problem

Definition

A **search problem** is a tuple $\langle \Sigma, succ, S_0, G \rangle$ where

- Σ is the set of *states*, which represents the set of *all* potential solutions.
- $succ \subseteq \Sigma \times \Sigma$ is a binary relation over the set of states, which denotes a relationship between the potential solutions. Intuitively, $(s, u) \in succ$ means that the potential solution u can be constructed from s. Each element (s, u) in $succ$ is called a **transition**.
- S_0 is a set of states describing the set of *initial* states from which the search starts.
- G is a set of states describing the set of *goal* states where the search can stop.

Search Problem with Actions and Costs

Definition

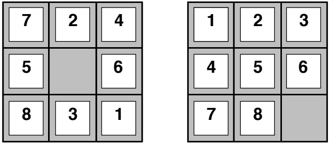
In the book, a **search problem** is a tuple $\langle \Sigma, A, succ, cost, S_0, G \rangle$ where

- Σ, S_0, and G are defined as in Definition **??**.
- A is a set of actions.
- $succ \subseteq \Sigma \times A \times \Sigma$ is a ternary relation over the set of states and actions, which denotes a relationship between the potential solutions. Intuitively, $(s, a, u) \in succ$ means that the potential solution u can be constructed from s *by executing a in* s. Similar to Definition **??**, each element (s, a, u) in $succ$ is called a **transition**.
- $cost$ is a function that maps transitions and actions to positive numbers. i.e., $cost : S \times A \times S \longrightarrow R^+$. Intuitively, $cost(s, a, u) = v$ means that it costs v to go from s to u by the action a.

Example 1: The 8-puzzle problem

The 8-puzzle problem can be defined by the search problem $\langle \Sigma, succ, S_0, G \rangle$ where

- Σ is the set of all possible configurations of the 8-puzzle problem.
- $succ \subseteq \Sigma \times \Sigma$ where $(s, u) \in succ$ iff u can be obtained from s by exchanging the empty tile with one of its neighbors.
- S_0 contains a single initial state that is given to us at the beginning of the game (e.g., the configuration on the left hand side of the figure below).
- G contains a single goal state (e.g., the state in which the numbers are arranged in the order 1, 2, 3, . . . , 8 when read from left to right from top to bottom, the configuration on the right hand side of the figure below.)

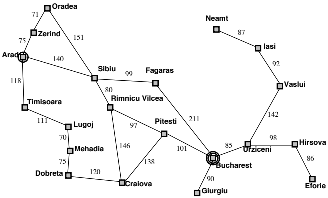


Start State **Goal State**

Example 2: Driving from Arad to Bucharest

The problem of finding a path from Arad to Bucharest can be defined by the search problem $\langle \Sigma, succ, S_0, G \rangle$ where

- Σ is the set of all cities in Romania.
- $succ \subseteq \Sigma \times \Sigma$ where $(s, u) \in succ$ iff there is a direct connection from s to u on the Romania map.
- $S_0 = \{Arad\}$.
- $G = \{Bucharest\}$.



Notes

- Sometime, it is easy to list all elements of Σ. For example, the set of states in Example 2 (Driving ...) has only 19 states. So, with little effort, we can list all elements of Σ.
  Sometime, we just cannot list all elements of Σ. The 8-puzzle problem has 9! (9 factorial) ($> 2^{16}$) elements. It will take some time to list them all!
  The number of states depends on the **number of properties of the world** and the **set of constants**.
- There are different formulations for the same problem. Finding one that is intuitive and enables an efficient search for the solution is not an easy task — **one gets better with more experience**!

Graph: Terminologies

Definition

A graph consists of a set N of **nodes** and a set A of ordered pairs of nodes, called **arcs** (or **edges**).

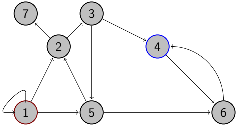
Node n_2 is a **successor** (or a **neighbor**) of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$. An arc may be labeled (by an action or by a cost).

A **path** is a sequence of nodes $\langle n_0, n_1, ..., n_k \rangle$ such that $\langle n_{i+1}, n_i \rangle \in A$.

A **cycle** is a nonempty path such that the end node is the same as the start node. A graph without cycle is called **directed acyclic graph** or DAG.

The **forward branching factor** of a node is the number of arcs going out from the node, and the **backward branching factor** of a node is the number of arcs going into the node. The **forward/backward branching factor** of a graph is the maximal forward/backward branching factor among its nodes.

Graph: Example



In the above graph,

- the set of nodes is $\{1, 2, 3, 4, 5, 6, 7\}$
- The set of arcs/edges is $\{(1, 2), (3, 4), (1, 5), (1, 1), (2, 3), (4, 6), (6, 4), (3, 5), (5, 6), (4, 4), (2, 7)\}$
- 1 is successor of itself; it is not a successor of any other node;
- 2 is successor of 1 or 5; it is not a successor of any other node;
- $\langle 5, 6, 4, 6 \rangle$ and $\langle 1, 2, 7 \rangle$ are paths. $\langle 6, 4, 6 \rangle$ is a cycle.
- Forward branching factor of 1 is 3. Backward branching factor of 1 is 1.

A Generic Graph Searching Algorithm

Given a graph, the set of start nodes, and the set of goal nodes. A path between a start node and a goal node is a solution. Searching algorithms provide us a way to find a solution.

Idea: Incrementally explore paths from start nodes. Maintaining a **frontier** or **fringe** of paths from the start nodes that have been explored.

A Generic Graph Searching Algorithm

```
Input:  a graph,
  a set of start nodes
  Boolean procedure goal(n):  true if n is a goal node.

fringe := ⟨s⟩    % s is a start node;

while fringe is not empty:
  select and remove a path ⟨n_0, ..., n_k⟩ from fringe;
  if goal(n_k)
    return ⟨n_0, ..., n_k⟩;
     for every successor n of n_k
    add ⟨n_0, ..., n_k, n⟩ to fringe;
   end while
```

A Generic Graph Searching Algorithm

```
Input:  a graph,
  a set of start nodes
  Boolean procedure goal(n):  true if n is a goal node.

fringe := ⟨s⟩    % s is a start node;

while fringe is not empty:
  select and remove a path ⟨n0, ..., nk⟩ from fringe;
  if goal(nk) % Check for goal
    return ⟨n0, ..., nk⟩;
  % Expand phase
  for every successor n of nk % use of succ: (nk, n) ∈ succ
    add ⟨n0, ..., nk, n⟩ to fringe;
  % End Expand phase
end while
```

A Generic Graph Searching Algorithm (Avoiding Cycles)

```
Input:  a graph,
  a set of start nodes
  Boolean procedure goal(n):  true if n is a goal node.

fringe := ⟨s⟩   % s is a start node; visited := {}

while fringe is not empty:
  select and remove a path ⟨n_0, ..., n_k⟩ from fringe;
  if n_k is not in visited
    add n_k to visited
    if goal(n_k)
      return ⟨n_0, ..., n_k⟩;
    for every successor n of n_k
      add ⟨n_0, ..., n_k, n⟩ to fringe;
end while
```

A Generic Graph Searching Algorithm (Avoiding Cycles)

```
Input:  a graph,
  a set of start nodes
  Boolean procedure goal(n):  true if n is a goal node.

fringe := ⟨s⟩   % s is a start node; visited := {}

while fringe is not empty:
  select and remove a path ⟨n_0, ..., n_k⟩ from fringe;
  if n_k is not in visited
    add n_k to visited
    if goal(n_k) % Check for goal
      return ⟨n_0, ..., n_k⟩;
    for every successor n of n_k % use of succ: (n_k, n) ∈ succ
      add ⟨n_0, ..., n_k, n⟩ to fringe;
end while
```

A Generic Graph Searching Algorithm (Book)

More formal, assume functions such as create a node, select a node, insert into a data structure, etc.

```
function GRAPH-SEARCH( problem, fringe)
returns a solution, or failure

   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
              add STATE[node] to closed
              fringe ← INSERTALL(EXPAND(node, problem), fringe)
   end
```