Solving Problems By Searching

Bùi Tiến Lên

2021

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

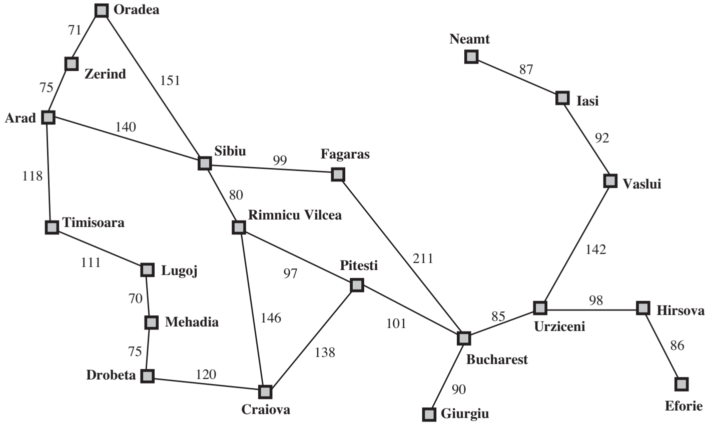
Problem-solving Agents

Problem-solving agents

- A **problem-solving agent** is one kind of goal-based agent
- It uses **atomic** representations

Example: Romania travelling

- You are at Arad
- Your friend's wedding tomorrow at Bucharest



Properties of the environment

- **Observable**
  - Each city has a sign indicating its presence for arriving drivers.
  - The agent always knows the current state.
- **Discrete**
  - Each city is connected to a small number of other cities.
  - There are only finitely many actions to choose from any given state.
- **Known**
  - The agent knows which states are reached by each action.
- **Deterministic**
  - Each action has exactly one outcome.
- *Under these assumptions, the solution to any problem is a fixed sequence of actions*

Solving problem by searching

- **Search**: the process of looking for a sequence of actions that reaches the goal
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- **Execution phase**: once a solution is found, the recommended actions are carried out.
  - While executing the solution, the agent ignores its percepts when choosing an action $\rightarrow$ open-loop system

Solving problem by searching (cont.)

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
persistent: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
return action
```

Well-defined problems and solutions

A **problem** can be defined formally by **five** components:

1. The **initial state** that the agent starts in
   - For example, the initial state for our agent in Romania might be described as $In(Arad)$
2. A description of the possible **actions** is available to the agent
   - For example, $Action(In(Arad)) = \{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
3. The **transition model**, which is a description of what each action does
   - For example, $Result(In(Arad), Go(Zerind)) \rightarrow In(Zerind)$
   - The term **successor** to refer to any state *reachable* from a given state by a single action
   - The initial state, actions, and transition model implicitly define the **state space** of the problem

Well-defined problems and solutions (cont.)

- The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions
- A **path** in the state space is a sequence of states connected by a sequence of actions

4. The **goal test**, which determines whether a given state is a goal state
    - For example, the agent's goal in Romania is the singleton set $\{In(Bucharest)\}$
5. A **path cost** function that assigns a numeric cost to each path
    - The **step cost** of taking action a in state s to reach state s' is denoted by $c(s, a, s')$

A **solution** to a problem is an action sequence that leads from the initial state to a goal state

- An **optimal solution** has the lowest path cost among all solutions

Formulating problems by abstraction

- The process of removing detail from a representation: abstract the state description and the actions
- Abstraction is critical for automated problem solving
    - Real-world is too detailed to model exactly
    - Create an approximate, simplified, model of the world for the computer to deal with
- The choice of a good abstraction thus involves
    - Removing as much detail as possible while
    - Retaining validity and ensuring that the abstract actions are easy to carry out

Example Problems

Toy problems vs. Real-world problems

Toy problems	Real-world problems
Illustrate or exercise various problem-solving methods	More difficult
Concise, exact description	No single, agreed-upon description
Can be used to compare performance	
E.g., 8-puzzle, 8-queens problem, Cryptarithmetic, Vacuum world, Missionaries and cannibals, simple route finding	E.g., Route finding, Touring and traveling salesperson problems, VLSI layout, Robot navigation, Assembly sequencing

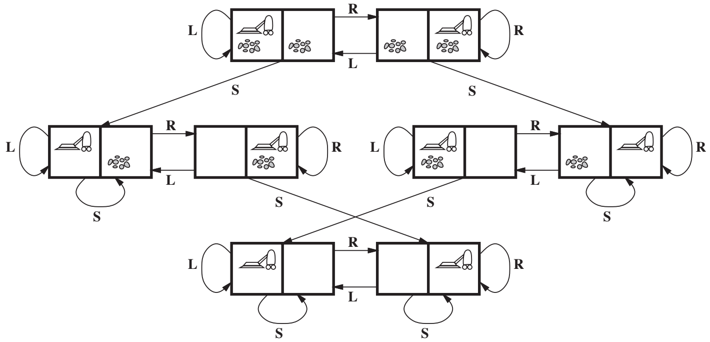
The vacuum world

States: The state is determined by both the agent location and the dirt locations. Thus, there are $2 \times 2^2 = 8$ possible world states ($n \times 2^n$ states in general).

- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1

The vacuum world (cont.)

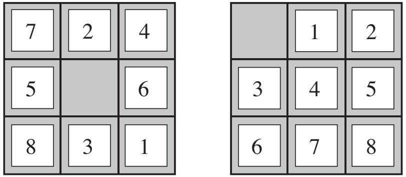
- **Transition model**: state space



The 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
- **Actions**:
  - *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
- **Path cost**: Each step costs 1



Start State Goal State

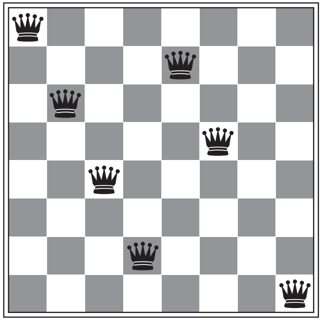
The 8-puzzle (cont.)

- It is a member of the family of sliding-block puzzles, NP-complete
- 8-puzzle: $9!/2 = 181{,}440$ reachable states $\rightarrow$ easily solved.
- 15-puzzle (on board 4×4): 1.3 trillion states $\rightarrow$ optimally solved in a few millisecs
- 24-puzzle (on board 5×5): around 10^{25} states $\rightarrow$ optimally solved in several hours

The 8-queens

There are two main kinds of formulation:

- An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and moves them around



The 8-queens (cont.)

Incremental formulation

- **States**: Any arrangement of 0 to 8 queens on the board is a state
    - The number of states $64 \times 63 \ldots 57 \approx 1.8 \times 10^{14}$
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.
- **Path cost**: no interest

Knuth's 4 problem

Devised by Donald Knuth (1964)

- Illustration of how infinite state spaces can arise
- Knuth's conjecture: Starting with the number 4, a sequence of *factorial* $\cdot!$, *square root* $\sqrt{\cdot}$, and *floor* $\lfloor\cdot\rfloor$ operations will reach any desired positive integer.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

States: any positive number.

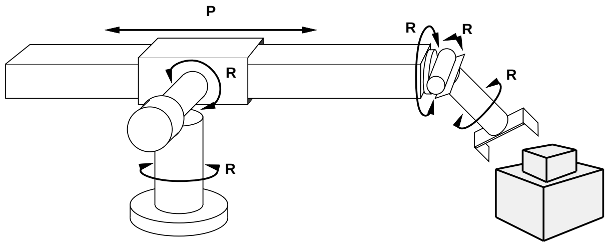
- **Initial state**: 4.
- **Actions**: Apply factorial, square root, or floor operation.
- **Transition model**: As given by the mathematical definitions of the operations (factorial for integers only).
- **Goal test**: State is the desired positive integer.

The route-finding problem

Consider the airline travel problems solved by a travel-planning Web site.
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.

Example: robotic assembly



- **states**: real-valued coordinates of robot joint angles parts of the object to be assembled
- **actions**: continuous motions of robot joints
- **goal test**: complete assembly *with no robot included!*
- **path cost**: time to execute

Searching for Solutions

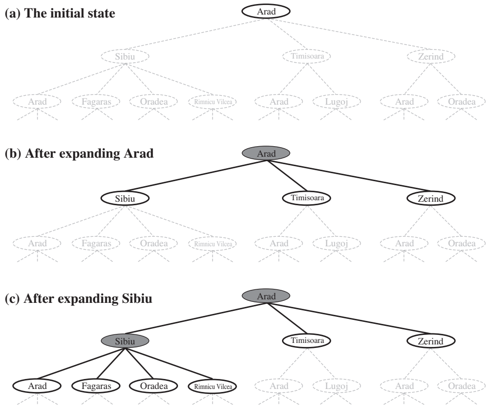
Search tree

A solution is an action sequence, so **search algorithms** work by considering various possible action sequences

- **Search tree**: the possible action sequences starting at the initial state
  - The **branches** are *actions* and the **nodes** correspond to *states* in the state space of the problem
  - The **root node** of the tree corresponds to the initial state
  - Taking actions by **expanding** the current state (**parent node**), thereby **generating** a new set of states (**child nodes**)
  - **Frontier**: the set of all leaf nodes available for expansion at any given point

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**

Search tree (cont.)



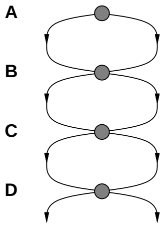
Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

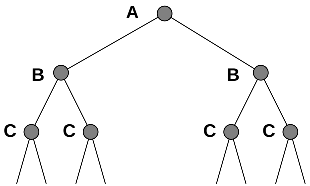
Search tree (cont.)

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then
      return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then
      return the corresponding solution
    expand the chosen node,
      adding the resulting nodes to the frontier
```

Redundant paths

- **Loopy path** cause repeated states in search tree





- **Redundant paths** (general concept) are unavoidable, which exist whenever there is more than one way to get from one state to another
- Following redundant paths can cause a tractable problem to become intractable
    - This is true even for algorithms that know how to avoid infinite loops

Graph search algorithm

- *Algorithms that forget their history are doomed to repeat it*
- Using data structure called the **explored set**, which remembers every expanded node

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then
      return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then
      return the corresponding solution
    add the node to the explored set
    expand the chosen node,
      adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

Graph search algorithm (cont.)

- The algorithm has another nice property: the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier

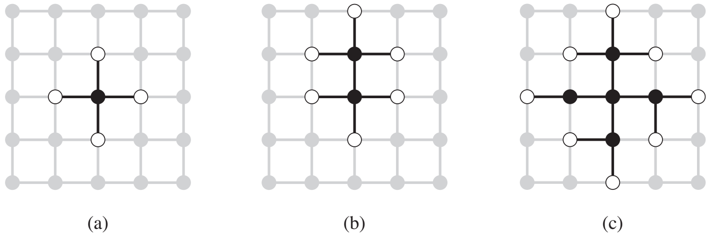


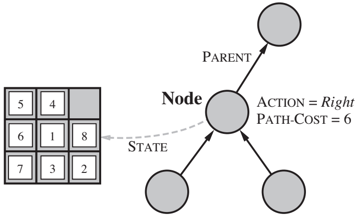
Figure 1: The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes)

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:

- $n.\textsc{State}$: the state in the state space to which the node corresponds;
- $n.\textsc{Parent}$: the node in the search tree that generated this node;
- $n.\textsc{Action}$: the action that was applied to the parent to generate the node;
- $n.\textsc{Path-Cost}$: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

Infrastructure for search algorithms (cont.)



```
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)
```

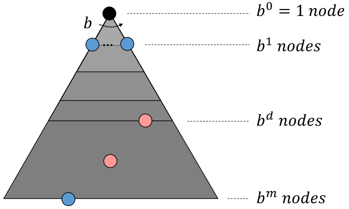
Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Time and space complexity are measured in terms of

- b: maximum **branching factor** of the search tree
- d: **depth** of the least-cost/shallowest solution
- m: **maximum depth** of the state space (may be ∞)



References

Goodfellow, I., Bengio, Y., and Courville, A. (2016).
Deep learning.
MIT press.

Lê, B. and Tô, V. (2014).
Cở sở trí tuệ nhân tạo.
Nhà xuất bản Khoa học và Kỹ thuật.

Nguyen, T. (2018).
Artificial intelligence slides.
Technical report, HCMC University of Sciences.

Russell, S. and Norvig, P. (2016).
Artificial intelligence: a modern approach.
Pearson Education Limited.