Problem Solving and Search

Outline

- Introduction to Problem Solving
- Complexity
- Uninformed search
  - Problem formulation
  - Search strategies: depth-first, breadth-first
- Informed search
  - Search strategies: best-first, memory bounded
  - Heuristic functions

Building a Problem Solving Program

- Define the problem precisely
- Analyze the problem
- Represent the task knowledge
- Choose and apply representation and reasoning techniques

To Specify a Problem

- Define the state space
- Specify the initial states
- Specify the goal states
- Specify the operations

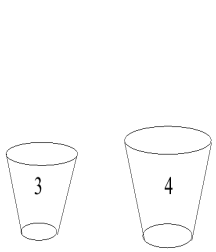
Production Systems

- A set of rules
- Knowledge/databases
- A control strategy
- A rule applier

Problem Characteristics

- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the problem's universe predictable?
- Is a good solution absolute or relative?
- Is the desired solution a state of the world or a path to a state?
- Is a large amount of knowledge absolutely required to solve the problem, or is it important only to constrain the search?
- Must problem-solving be interactive?

The Water Jug Problem

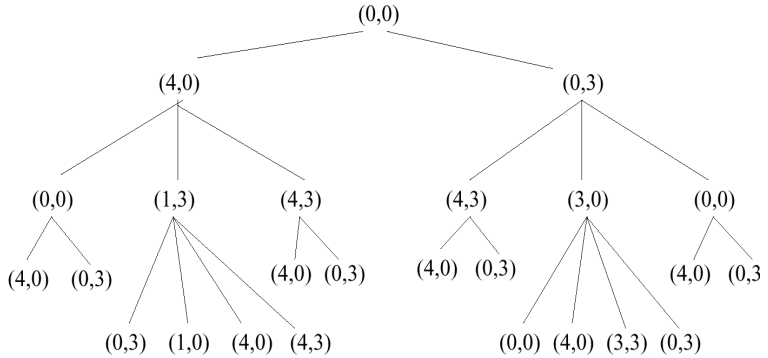


- **Given:**
  - one three- liter jug,
  - one four- liter jug
  - tap that can fill the jugs with water
- **Goal:** exactly two liters of water in the four- liter jug

Water Jug Problem

1. (X,Y: X < 4) -> (4,Y) Fill the 4-liter jug
2. (X,Y: Y < 3) -> (X,3) Fill the 3-liter jug
3. (X,Y: X > 0) -> (0,Y) Empty the 4-liter jug on the ground
4. (X,Y: Y > 0) -> (X,0) Empty the 3-liter jug on the ground
5. (X,Y: X+Y >= 4 and Y > 0) -> (4,Y-(4-X))
   Fill the 4-liter jug from the 3-liter jug
6. (X,Y: X+Y >= 3 and X > 0) -> (X-(3-Y),3))
   Fill the 3-liter jug from the 4-liter jug
7. (X,Y: X+Y <= 4 and Y > 0) -> (X+Y,0)
   Pour all water from the 3-liter jug into the 4-liter jug
8. (X,Y: X+Y <= 3 and X > 0) -> (0,X+Y))
   Pour all water from the 4-liter jug into the 3-liter jug
9. (X,Y: X > 0) -> (X-D,Y)
10. (X,Y: Y > 0) -> (X,Y-D)

Water Jug Problem



Examples of task environments and their characteristics.

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept
  static: *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 do**
    goal ← FORMULATE-GOAL(*state*)
    problem ← FORMULATE-PROBLEM(*state*, *goal*)
    seq ← SEARCH(*problem*)
  action ← FIRST(*seq*)
  seq ← REST(*seq*)
  return *action*

Note: This is *offline* problem-solving. *Online* problem-solving involves acting without complete knowledge of the problem and environment

Problem types

- **Single-state problem:** deterministic, accessible
  Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.
- **Multiple-state problem:** deterministic, inaccessible
  Agent must reason about sequences of actions and states assumed while working towards goal state.
- **Contingency problem:** nondeterministic, inaccessible
  - *Must use sensors during execution*
  - *Solution is a tree or policy*
  - *Often interleave search and execution*
- **Exploration problem:** unknown state space
  Discover and learn about environment while taking actions.

Well-defined problems

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent.
  - Given a particular state , SUCCESSOR-FN(x) returns a set of *action successor* ordered pairs, where each successor is a state that can be reached from x.
  - A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal state.
- A **path cost** function that assigns a numeric cost to each path.
- A **solution** to a problem is a path from the initial state to a goal state.
  - Solution quality is measured by the path cost function
  - **An optimal solution** has the lowest path cost among all solutions.

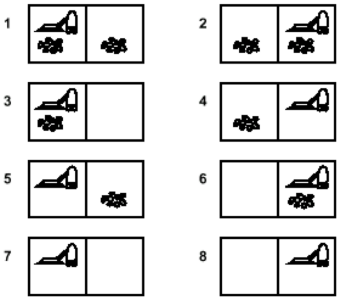
Example: Vacuum world

Simplified world: 2 locations, each may or not contain dirt, each may or not contain vacuuming agent.
Goal of agent: clean up the dirt.

Single-state, start in #5. Solution??

Multiple-state, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., *Right* goes to $\{2, 4, 6, 8\}$. Solution??

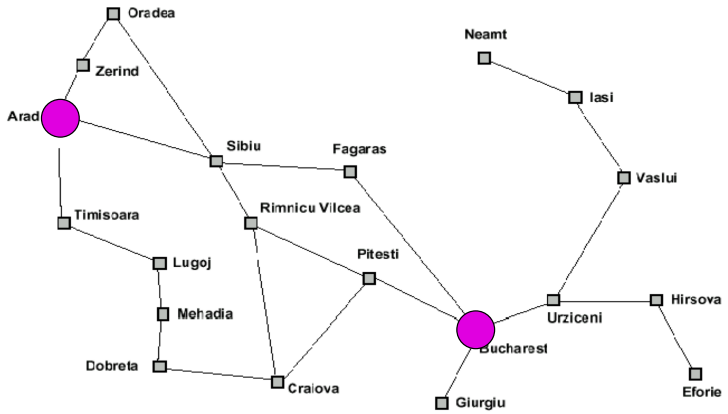
Contingency, start in #5
Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.
Solution??



Example: Romania

- In Romania, on vacation. Currently in Arad.
- Flight leaves tomorrow from Bucharest.
- **Formulate goal:**
  - be in Bucharest
- **Formulate problem:**
  - states: various cities
  - operators: drive between cities
- **Find solution:**
  - sequence of cities, such that total driving distance is minimized.

Example: Traveling from Arad To Bucharest



Problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

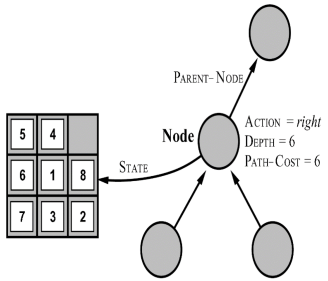
operators (or *successor function* $S(x)$)
e.g., Arad → Zerind Arad → Sibiu etc.

goal test, can be
explicit, e.g., x = "at Bucharest"
implicit, e.g., $NoDirt(x)$

path cost (additive)
e.g., sum of distances, number of operators executed, etc.

A *solution* is a sequence of operators
leading from the initial state to a goal state

Encapsulating *state* information in *nodes*

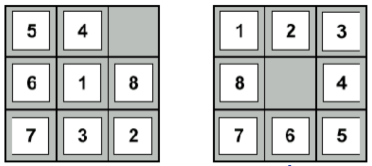


- STATE: the state in the state space to which the node corresponds.
- PARENT-NODE: the node in the search tree that generated this node.
- ACTION: the action that was applied to the parent to generate the node.
- PATH-COST: the cost of the path from the initial state to the node, as indicated by the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

Selecting a state space

- Real world is absurdly complex; some abstraction is necessary to allow us to reason on it…
- Selecting the correct abstraction and resulting state space is a difficult problem!
- Abstract states ⇔ real-world states
- Abstract operators ⇔ sequences or real-world actions
  (e.g., going from city i to city j costs Lij ⇔ actually drive from city i to j)
- Abstract solution ⇔ set of real actions to take in the real world such as to solve problem

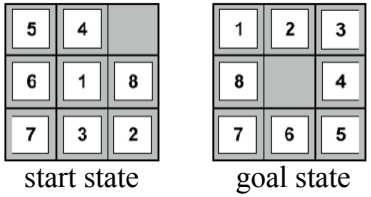
Example: 8-puzzle



start state goal state

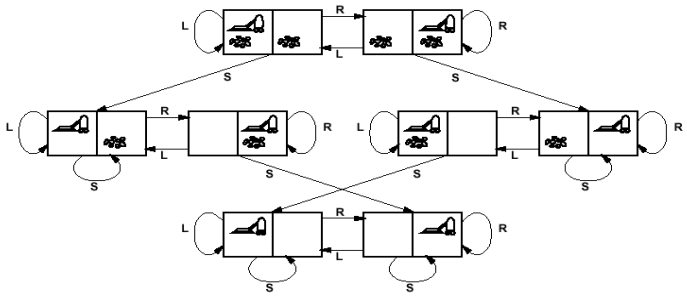
- State:
- Operators:
- Goal test:
- Path cost:

Example: 8-puzzle



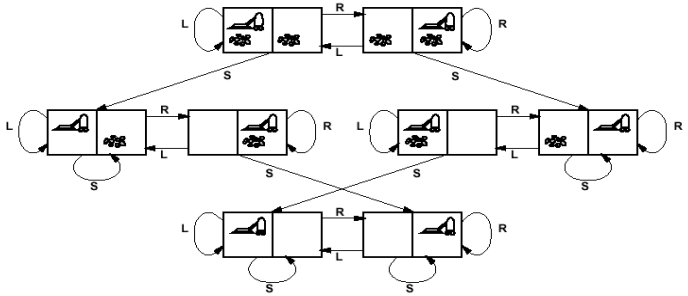
- State: integer location of tiles (ignore intermediate locations)
- Operators: moving blank left, right, up, down (ignore jamming)
- Goal test: does state match goal state?
- Path cost: 1 per move

Back to Vacuum World



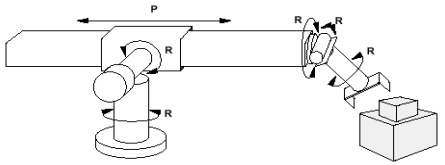
states??
operators??
goal test??
path cost??

Back to Vacuum World



states??: integer dirt and robot locations (ignore dirt *amounts*)
operators??: *Left*, *Right*, *Suck*
goal test??: no dirt
path cost??: 1 per operator

Example: Robotic Assembly



states??: real-valued coordinates of
robot joint angles
parts of the object to be assembled

operators??: continuous motions of robot joints

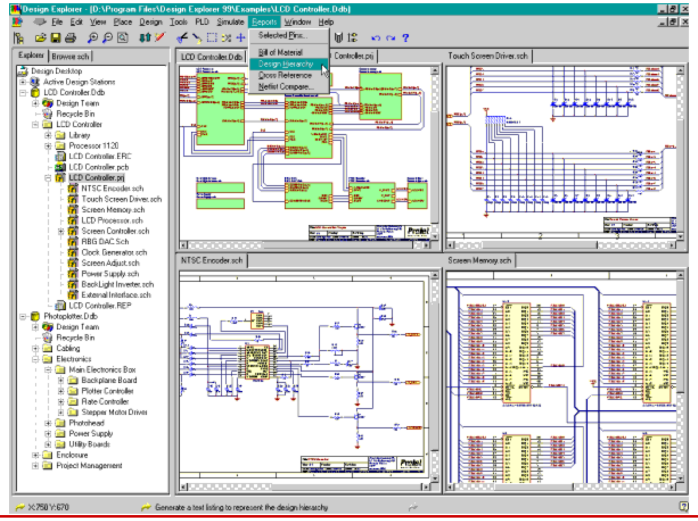
goal test??: complete assembly *with no robot included!*

path cost??: time to execute

Real-life example: VLSI Layout

- Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)
- "optimal way"??
  - minimize surface area
  - minimize number of signal layers
  - minimize number of connections from one layer to another
  - minimize length of some signal lines (e.g., clock line)
  - distribute heat throughout board
  - etc.

A Design Tool



Search algorithms

Basic idea: offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

Function General-Search(*problem*, *strategy*) returns a *solution*, or failure
  initialize the search tree using the initial state problem
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add resulting nodes to the search tree
  end

Tree Search Algorithms

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

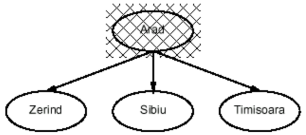
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
  loop do
    if EMPTY?(*fringe*) **then return** failure
    node ← REMOVE-FRONT(*fringe*)
    if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds
      then return SOLUTION(*node*)
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
  end

function EXPAND(*node*, *problem*) **returns** a set of nodes

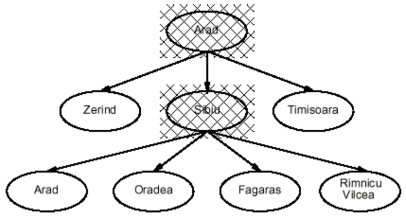
  successors ← the empty set
  for each ⟨*action*, *result*⟩ **in** SUCCESSOR-FN[*problem*](STATE[*node*]) **do**
    successor ← a new NODE
    STATE[*successor*] ← *result*
    PARENT-NODE[*successor*] ← *node*
    ACTION[*successor*] ← *action*
    PATH-COST[*successor*] ← PATH-COST[*node*] + STEP-COST(*node*, *action*, *successor*)
    DEPTH[*successor*] ← DEPTH[*node*] + 1
    add *successor* to *successors*
  end
  return *successors*
end

General search example

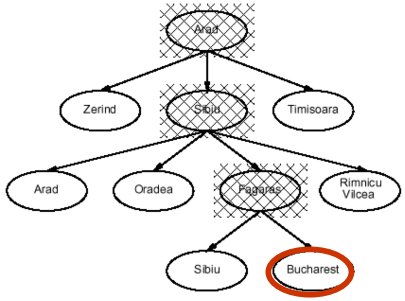
General search example



General search example



General search example



Evaluation of search strategies

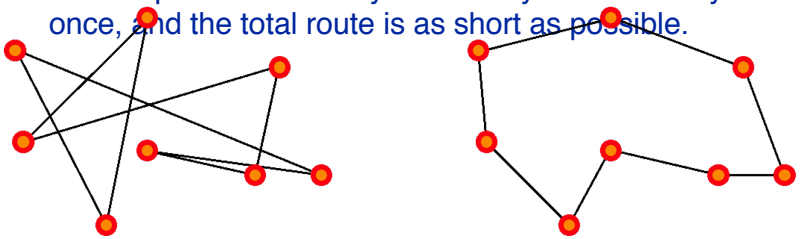
- A search strategy is defined by picking the order of node expansion.
- Search algorithms are commonly evaluated according to the following four criteria:
  - Completeness: is the algorithm guaranteed to find a solution when there is one?
  - Optimality: does the strategy find the optimal solution
  - Time complexity: how long does it take to find a solution?
  - Space complexity: how much memory does it need to perform the search?
- Time and space complexity are measured in terms of:
  - *b* – max branching factor of the search tree
  - *d* – depth of the least-cost solution
  - *m* – max depth of the state-space (may be infinity)

Complexity

- Why worry about complexity of algorithms?
  - because a problem may be solvable in principle but may take too long to solve in practice
- How can we evaluate the complexity of algorithms?
- through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size n of a problem when n tends towards infinity

Complexity example: Traveling Salesman Problem

- There are n cities, with a road of length Lij joining city i to city j.
- The salesman wishes to find a way to visit all cities that is optimal in two ways: each city is visited only once, and the total route is as short as possible.



- This is a *hard* problem: the only known algorithms (so far) to solve it have exponential complexity, that is, the number of operations required to solve it grows as e^n for *n* cities.

Why is exponential complexity “hard”?

It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).

- e^{1} = 2.72
- e^{10} = $2.20\ 10^{4}$ (daily salesman trip)
- e^{100} = $2.69\ 10^{43}$ (monthly salesman planning)
- e^{500} = $1.40\ 10^{217}$ (music band worldwide tour)
- $e^{250,000}$ = $10^{108,573}$ (postal services)
- Fastest computer = 10^{12} operations/second

So…

In general, exponential-complexity problems *cannot be solved for any but the smallest instances!*

Complexity

- Polynomial-time (P) problems: we can find algorithms that will solve them in a time (=number of operations) that grows polynomially with the size of the input.
- **for example:** sort n numbers into increasing order: poor algorithms have n2 complexity, better ones have n log(n) complexity.
- Since we did not state what the order of the polynomial is, it could be very large! Are there algorithms that require more than polynomial time?
- Yes (until proof of the contrary); for some algorithms, we do not know of any polynomial-time algorithm to solve them. These are referred to as non-polynomial-time (NP) algorithms.
- **for example**: traveling salesman problem.
- In particular, exponential-time algorithms are believed to be NP.

Note on NP-hard problems

- The formal definition of NP problems is:
  - A problem is nondeterministic polynomial if there exists some algorithm that can guess a solution and then verify whether or not the guess is correct in polynomial time.
  - (one can also state this as these problems being solvable in polynomial time on a nondeterministic Turing machine.)
- In practice, until proof of the contrary, this means that known algorithms that run on known computer architectures will take more than polynomial time to solve the problem.

Complexity and the human brain

- Are computers close to human brain power?
- Current computer chip (CPU):
  - 10^3 inputs (pins)
  - 10^7 processing elements (gates)
  - 2 inputs per processing element (fan-in = 2)
  - processing elements compute boolean logic (OR, AND, NOT, etc)
- Typical human brain:
  - 10^7 inputs (sensors)
  - 10^{10} processing elements (neurons)
  - fan-in = 10^3
  - processing elements compute complicated functions

Still a lot of improvement needed for computers; but computer clusters come close!