Problem solving and search

Chapter 3, Sections 1–5

Outline

◇ Problem-solving agents

◇ Problem types

◇ Problem formulation

◇ Example problems

◇ Basic search algorithms

Problem-solving agents

Restricted form of general agent:

```
function SIMPLE-PROBLEM-SOLVING-AGENT( p) returns an action
   inputs: p, a percept
   static: s, an action sequence, initially empty
           state, some description of the current world state
           g, a goal, initially null
           problem, a problem formulation

   state ← UPDATE-STATE(state, p)
   if s is empty then
       g ← FORMULATE-GOAL(state)
       problem ← FORMULATE-PROBLEM(state, g)
       s ← SEARCH( problem)
   action ← RECOMMENDATION(s, state)
   s ← REMAINDER(s, state)
   return action
```

Note: this is *offline* problem solving.
Online problem solving involves acting without complete knowledge of the problem and solution.

Example: Romania

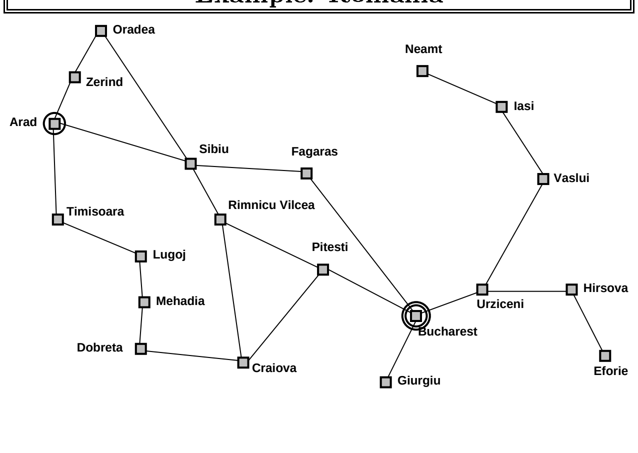
On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:
 be in Bucharest

Formulate problem:
 states: various cities
 operators: drive between cities

Find solution:
 sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

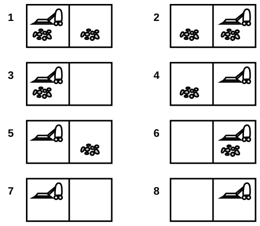
Deterministic, accessible $\Longrightarrow$ *single-state problem*
Deterministic, inaccessible $\Longrightarrow$ *multiple-state problem*

Nondeterministic, inaccessible $\Longrightarrow$ *contingency problem*
 must use sensors during execution
 solution is a *tree* or *policy*
 often *interleave* search, execution

Unknown state space $\Longrightarrow$ *exploration problem* ("online")

Example: vacuum world

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??

Single-state problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

operators (or *successor function* $S(x)$)
e.g., Arad $\rightarrow$ Zerind Arad $\rightarrow$ 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

Selecting a state space

Real world is absurdly complex
$\Rightarrow$ state space must be *abstracted* for problem solving

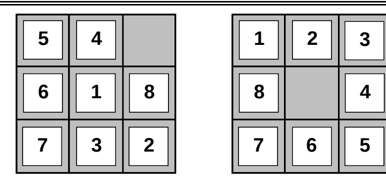
(Abstract) state = set of real states

(Abstract) operator = complex combination of real actions
e.g., "Arad $\rightarrow$ Zerind" represents a complex set
of possible routes, detours, rest stops, etc.
For guaranteed realizability, any real state "in Arad"
must get to *some* real state "in Zerind"

(Abstract) solution =
set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

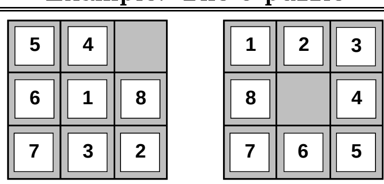
Example: The 8-puzzle



Start State Goal State

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

Example: The 8-puzzle

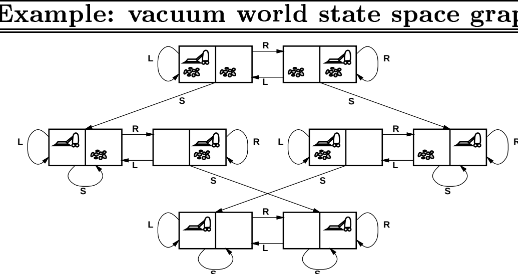


Start State Goal State

states??: integer locations of tiles (ignore intermediate positions)
operators??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

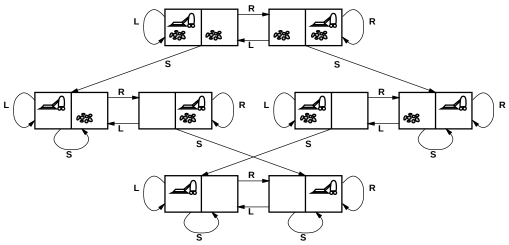
[Note: optimal solution of n-Puzzle family is NP-hard]

Example: vacuum world state space graph



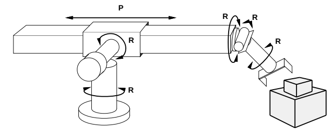
states??
operators??
goal test??
path cost??

Example: vacuum world state space graph



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!*

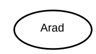
path cost??: time to execute

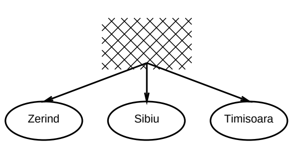
Search algorithms

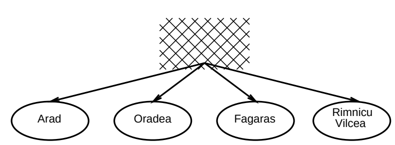
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

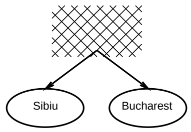
```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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


General search example







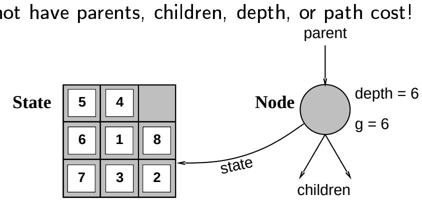


Implementation of search algorithms

function GENERAL-SEARCH(*problem*, QUEUING-FN) **returns** a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
  loop do
    if *nodes* is empty **then return** failure
    node ← REMOVE-FRONT(*nodes*)
    if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
    nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))
  end

Implementation contd: states vs. nodes

A *state* is a (representation of) a physical configuration
A *node* is a data structure constituting part of a search tree
  includes *parent*, *children*, *depth*, *path cost* $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

Search strategies

A strategy is defined by picking the *order of node expansion*

Strategies are evaluated along the following dimensions:
  completeness—does it always find a solution if one exists?
  time complexity—number of nodes generated/expanded
  space complexity—maximum number of nodes in memory
  optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of
  b—maximum branching factor of the search tree
  d—depth of the least-cost solution
  m—maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

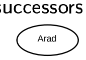
Depth-limited search

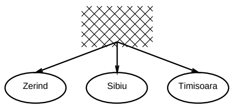
Iterative deepening search

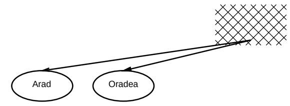
Breadth-first search

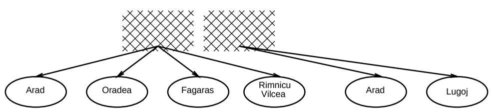
Expand shallowest unexpanded node

Implementation:
  QUEUEINGFN = put successors at end of queue









Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

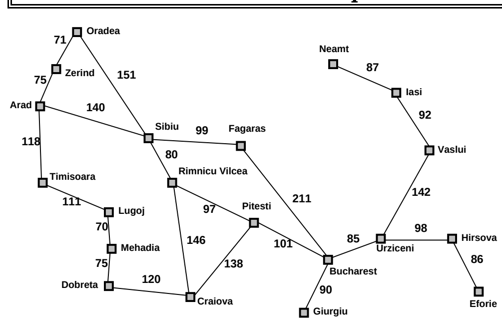
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 1MB/sec so 24hrs = 86GB.

Romania with step costs in km



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

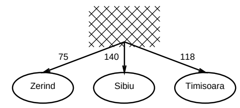
Uniform-cost search

Expand least-cost unexpanded node

Implementation:
QueueingFn = insert in order of increasing path cost

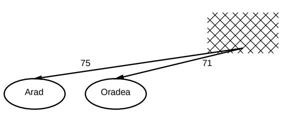
.

.
.



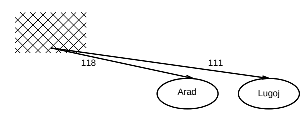
.

.
.



.

.
.



Properties of uniform-cost search

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution

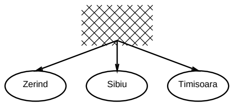
Space?? # of nodes with $g \leq$ cost of optimal solution

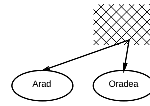
Optimal?? Yes

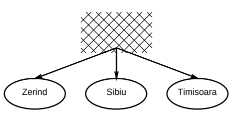
Depth-first search

Expand deepest unexpanded node

Implementation:
QueueingFn = insert successors at front of queue

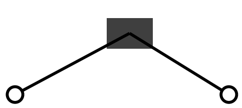


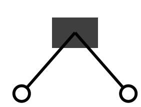


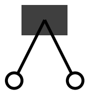


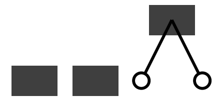
I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

DFS on a depth-3 binary tree

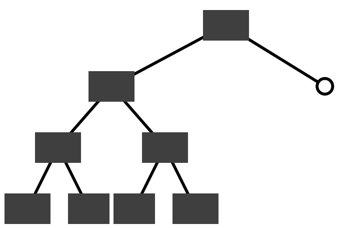


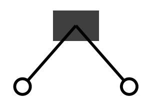


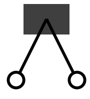


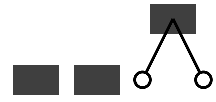


DFS on a depth-3 binary tree, contd.









Properties of depth-first search

Complete??

Time??

Space??

Optimal??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Depth-limited search

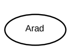
= depth-first search with depth limit l

Implementation:
Nodes at depth l have no successors

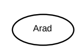
Iterative deepening search

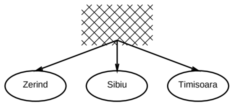
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH( problem, depth)
      if result ≠ cutoff then return result
   end
```


Iterative deepening search $l = 0$

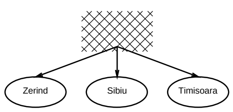


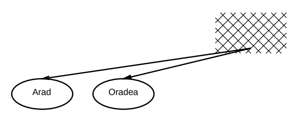
Iterative deepening search $l = 1$

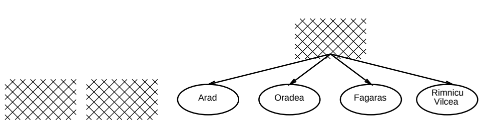


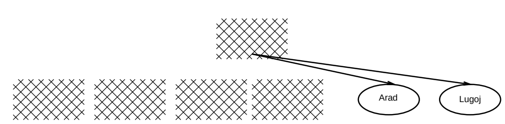


Iterative deepening search $l = 2$









Properties of iterative deepening search

Complete??

Time??

Space??

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1
Can be modified to explore uniform-cost tree

Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms