เอกสารประกอบการอบรม

เรื่อง State Space Search Algorithms

ผู้ช่วยศาสตราจารย์ ดร. สุกรี สินธุภิญโญ

นำมาจาก Slide ของ AMIA Slides Stuart Russell and Peter Norvig

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:

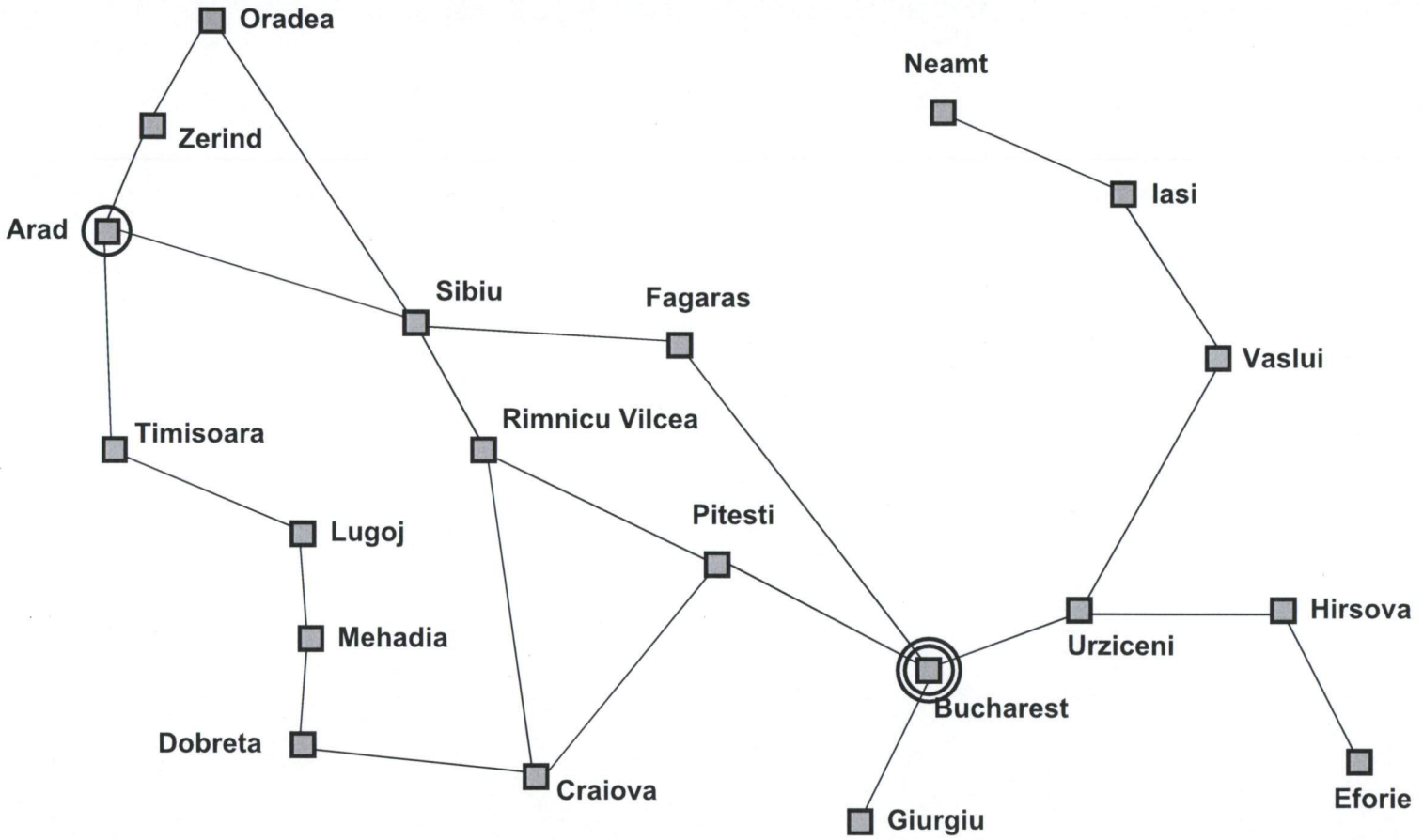
states: various cities

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

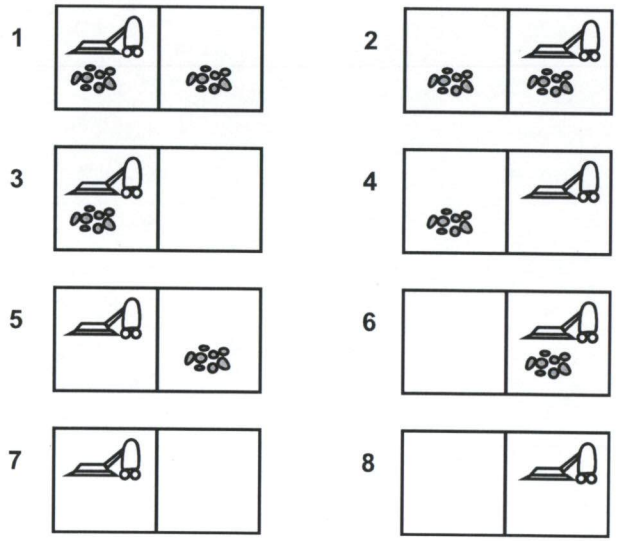
Unknown state space $\Longrightarrow$ *exploration problem* ("online")

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:

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.

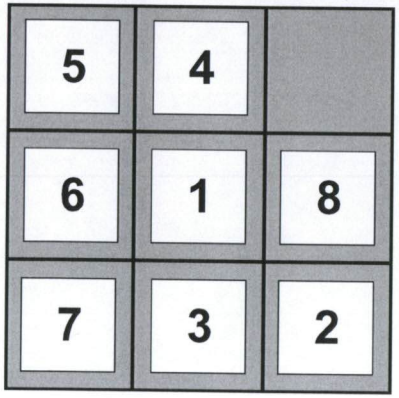
For guaranteed realizability, any real state "in Arad" must get to *some* real state "in Zerind"

(Abstract) solution =

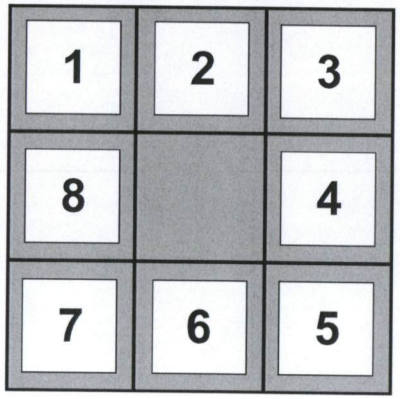
set of real paths that are solutions in the real world

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

Example: The 8-puzzle



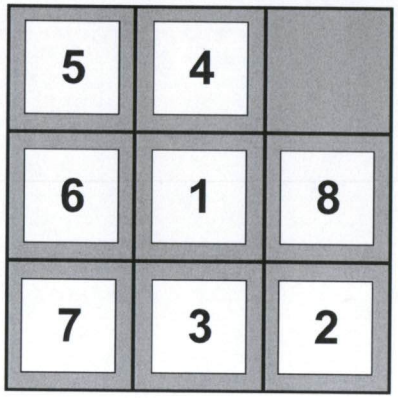
Start State



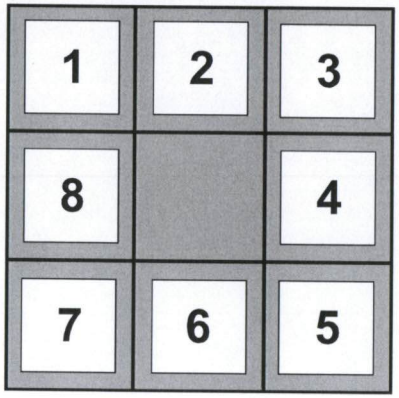
Goal State

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

Example: The 8-puzzle



Start State

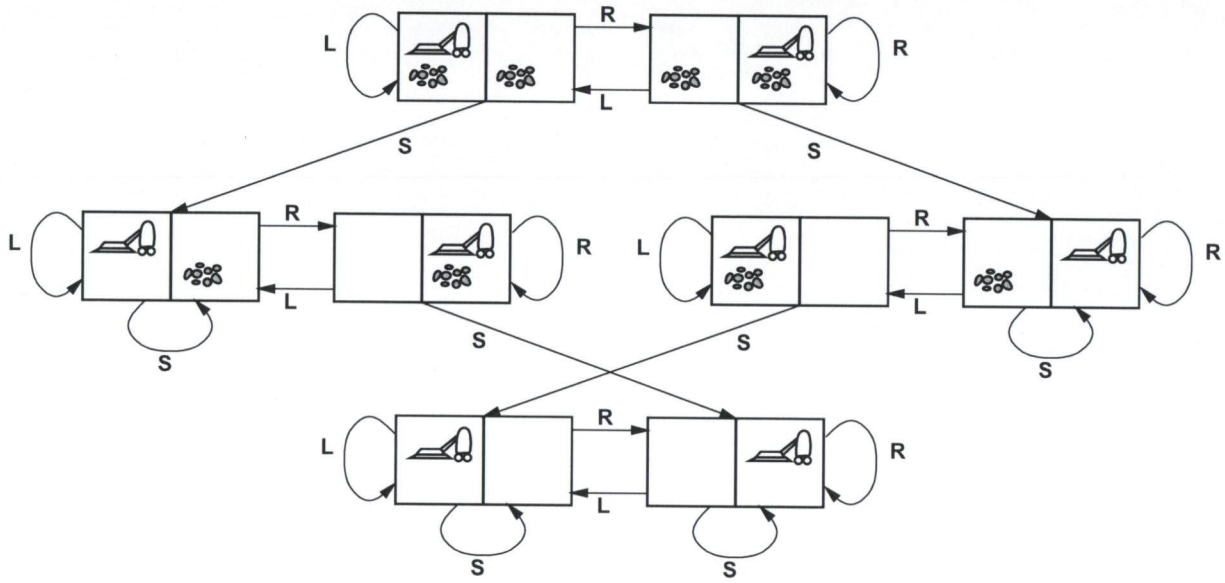


Goal State

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



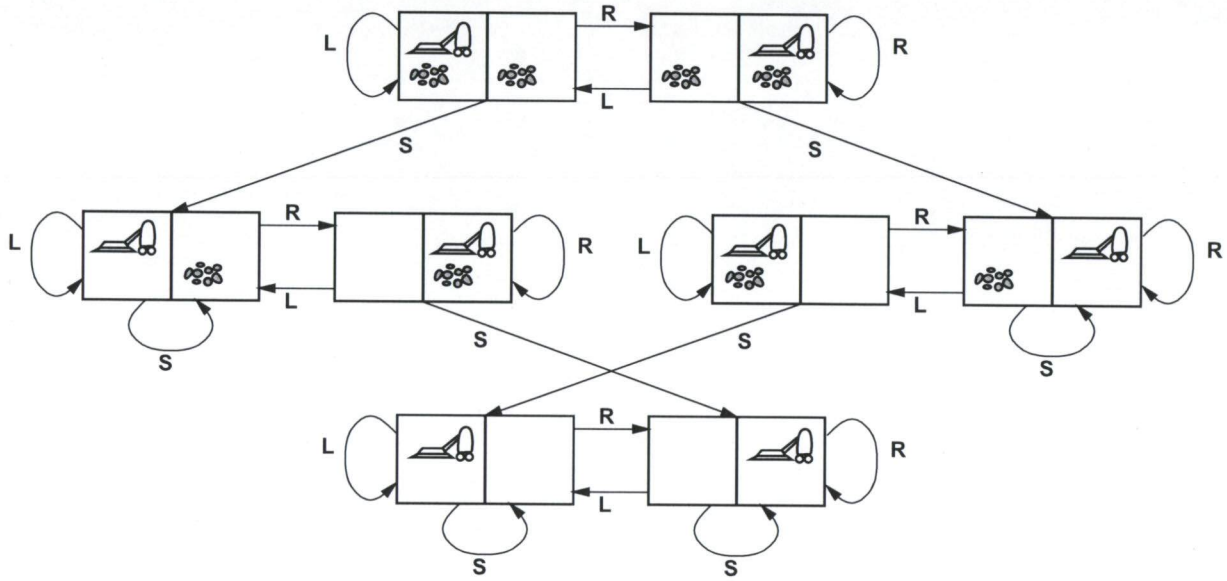
states??

operators??

goal test??

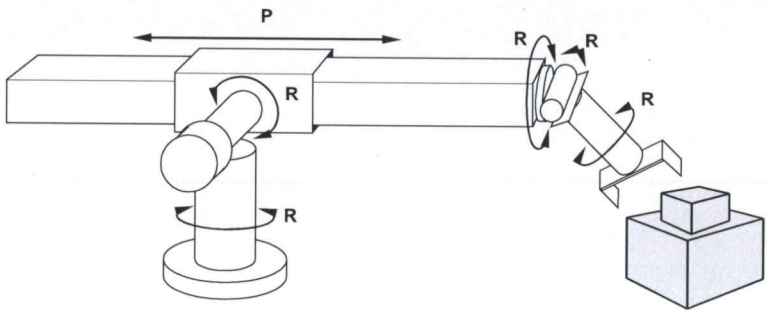
path cost??

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:

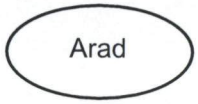
offline, simulated exploration of state space

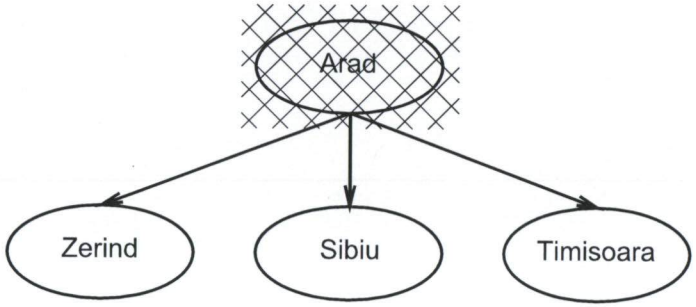
by generating successors of already-explored states

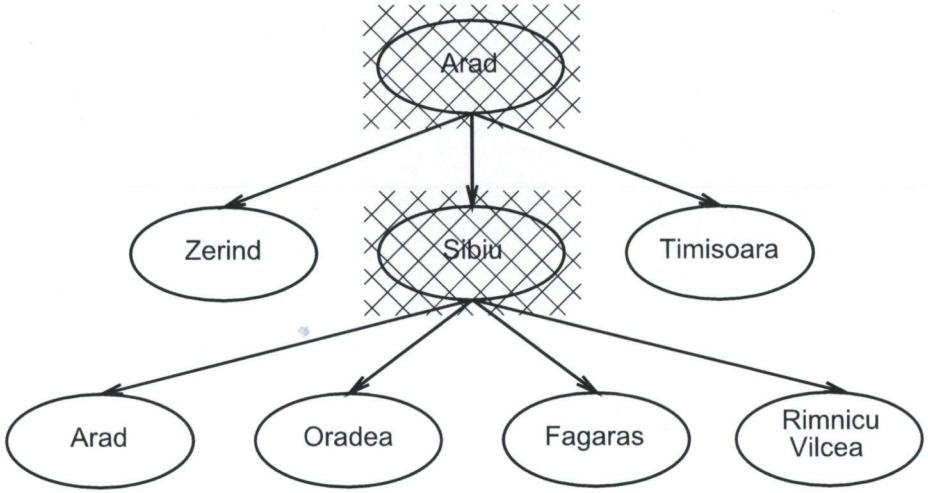
(a.k.a. *expanding* states)

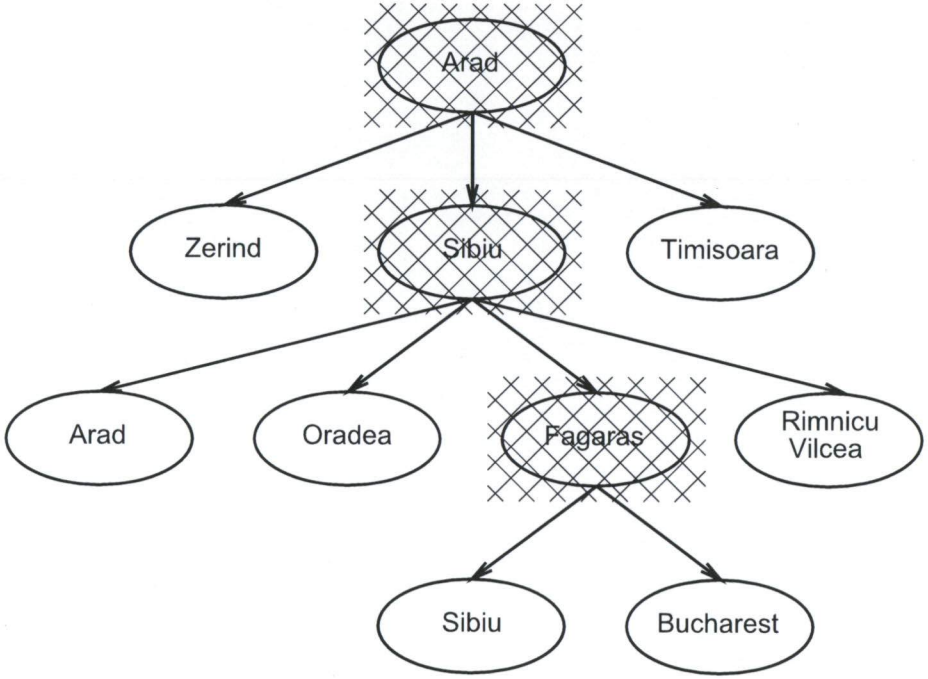
```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains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




Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Fagaras
Rimnicu
Vilcea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Fagaras
Rimnicu
Vilcea
Sibiu
Bucharest

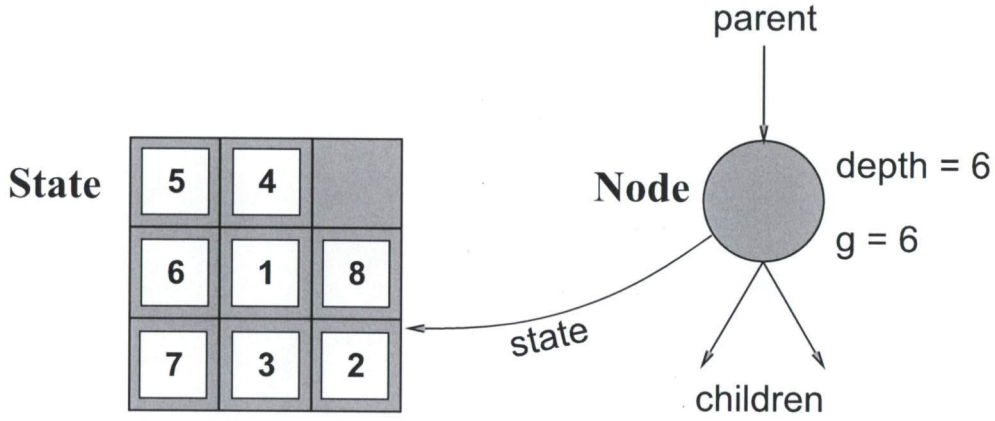
Implementation of search algorithms

```
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

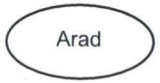
Uninformed

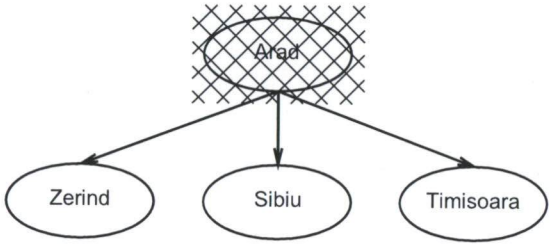
Breadth-first search

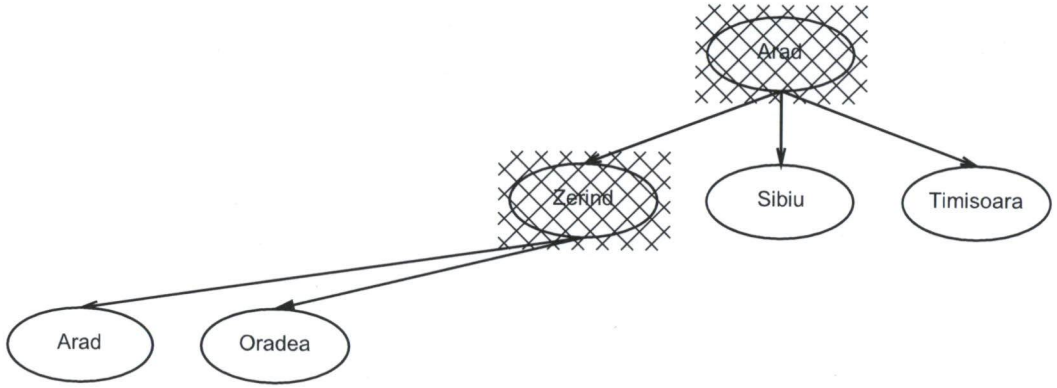
Expand shallowest unexpanded node

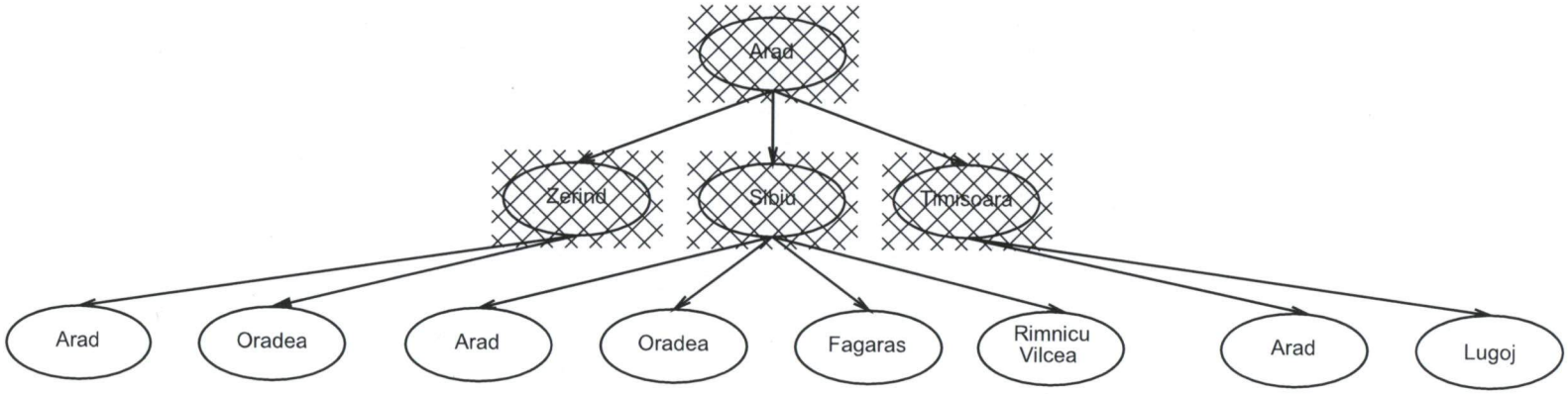
Implementation:

QUEUEINGFN = put successors at end of queue




Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu
Vilcea
Arad
Lugoj

Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

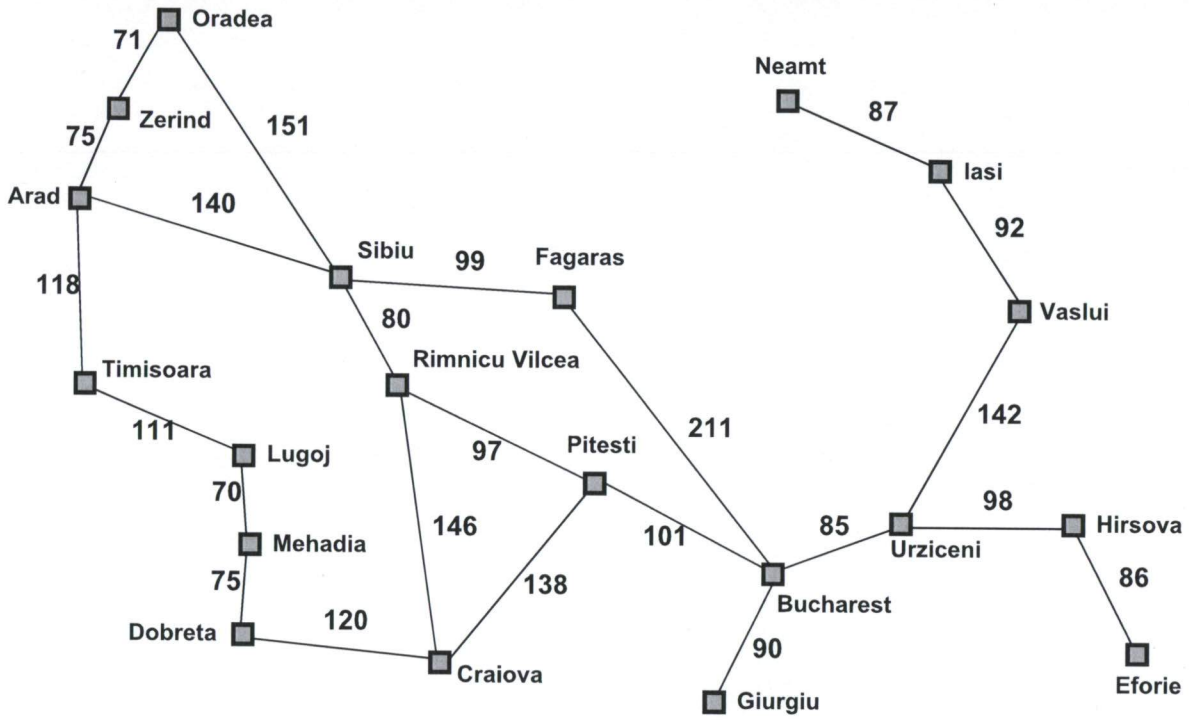
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d

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

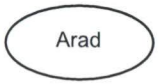
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

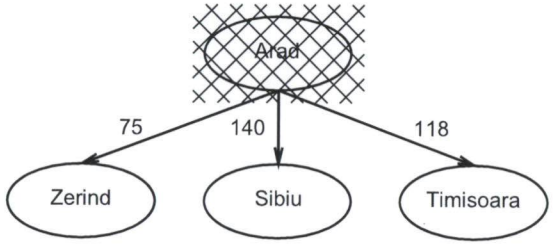
Uniform-cost search

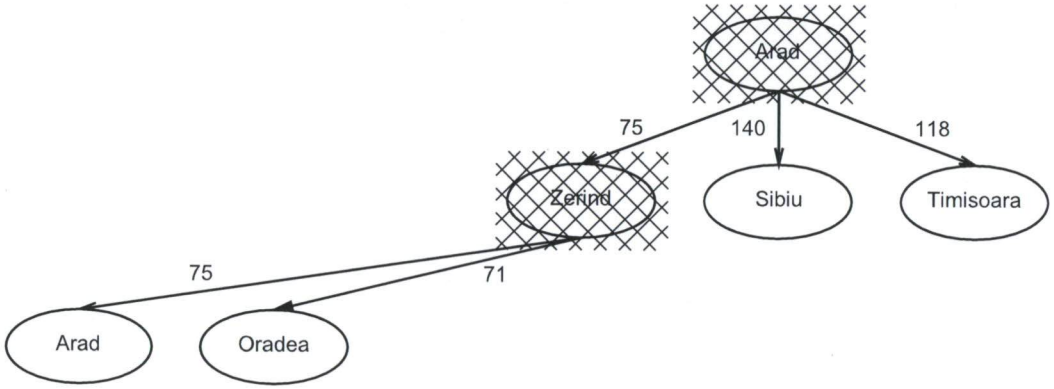
Expand least-cost unexpanded node

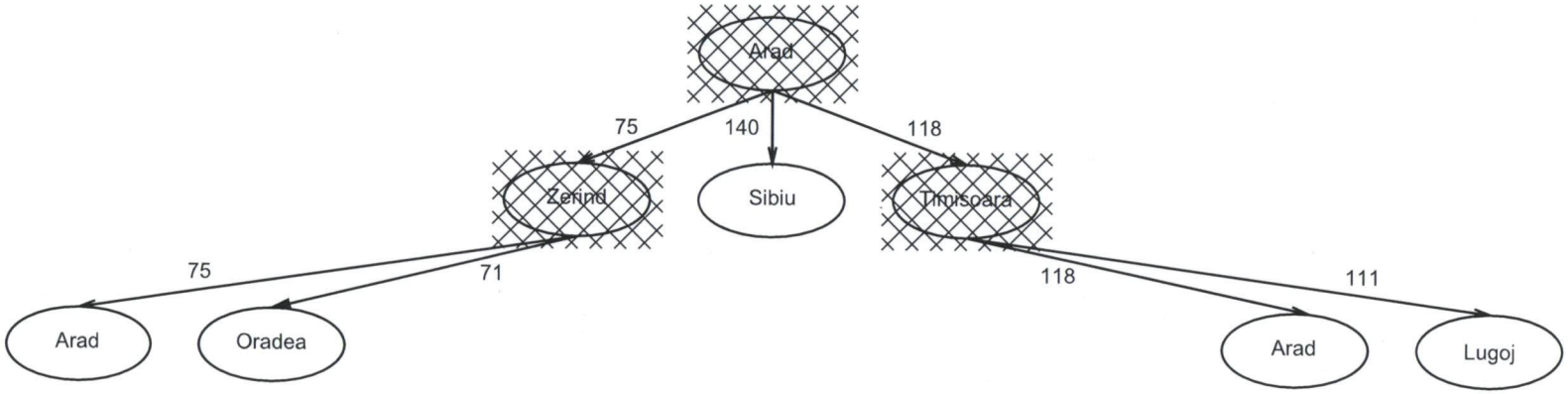
Implementation:

QUEUEINGFN = insert in order of increasing path cost




Arad
75
140
118
Zerind
Sibiu
Timisoara


Arad
75
140
118
Zerind
Sibiu
Timisoara
75
71
Arad
Oradea


Arad
75
140
118
Zerind
Sibiu
Timisoara
75
71
118
111
Arad
Oradea
Arad
Lugoj

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

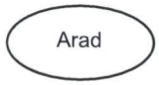
Optimal?? Yes

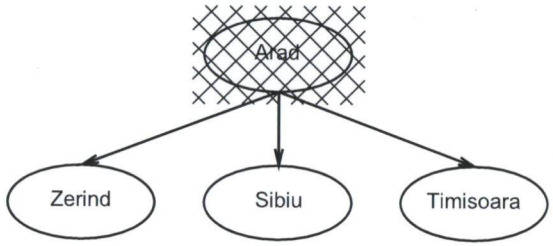
Depth-first search

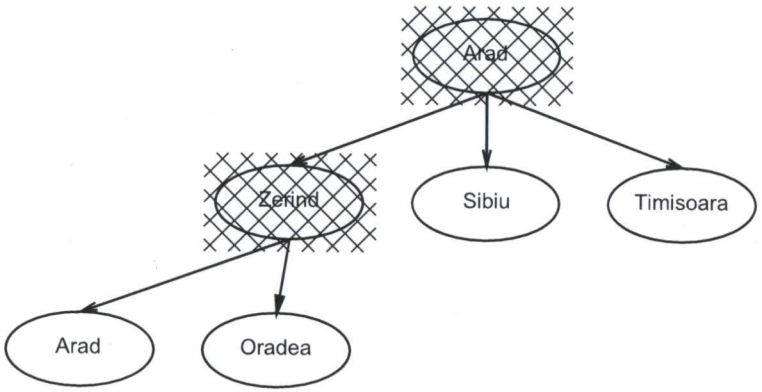
Expand deepest unexpanded node

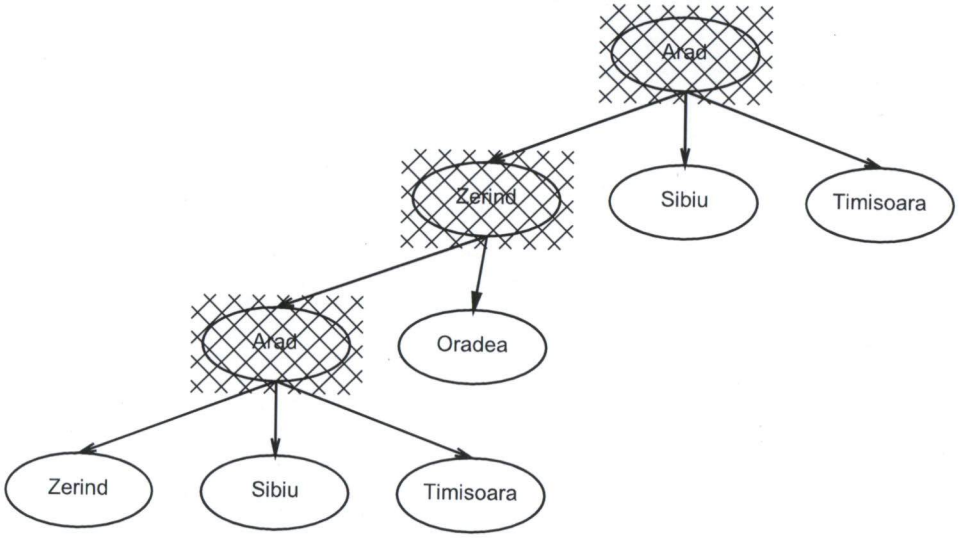
Implementation:

QUEUEINGFN = insert successors at front of queue



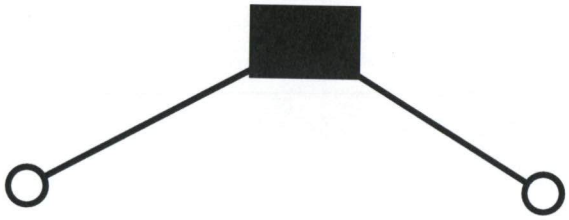

Arad
Zerind
Sibiu
Timisoara

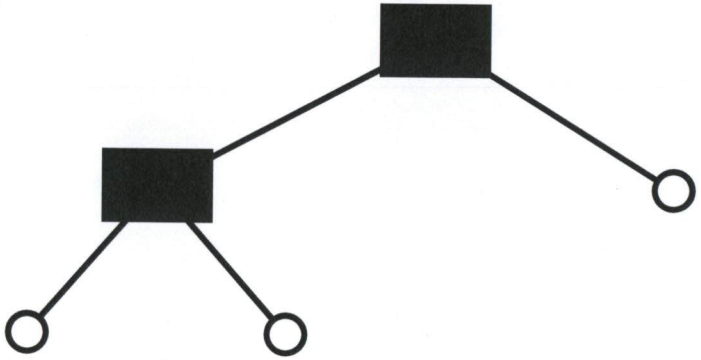

Arad
Zerind
Sibiu
Timisoara
Arad
Oradea

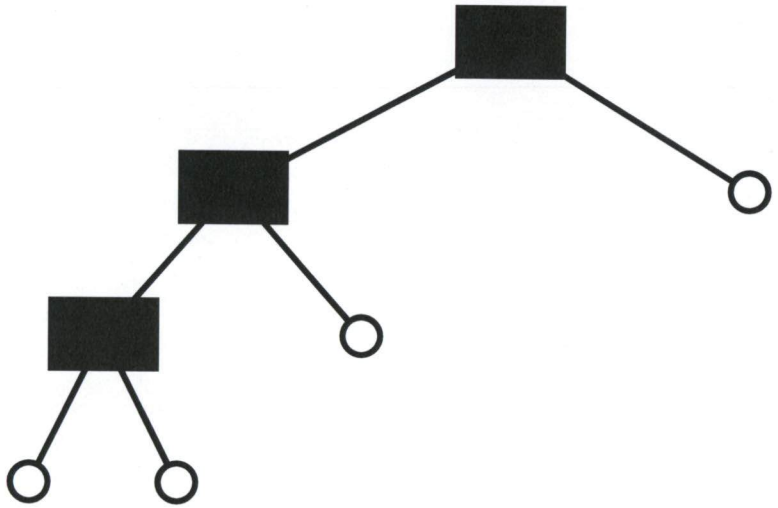


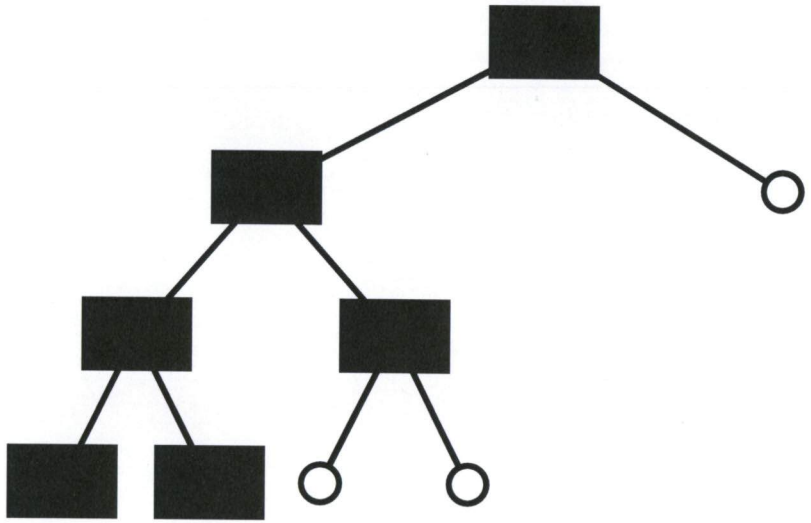
I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

DFS on a depth-3 binary tree

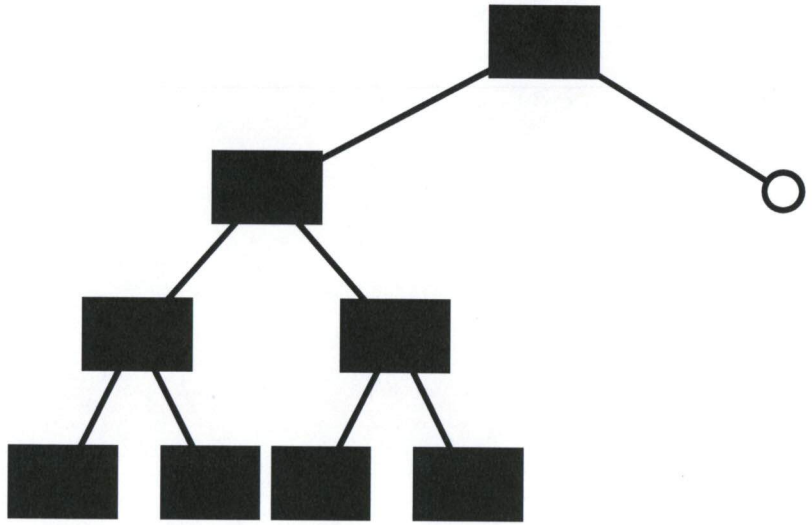


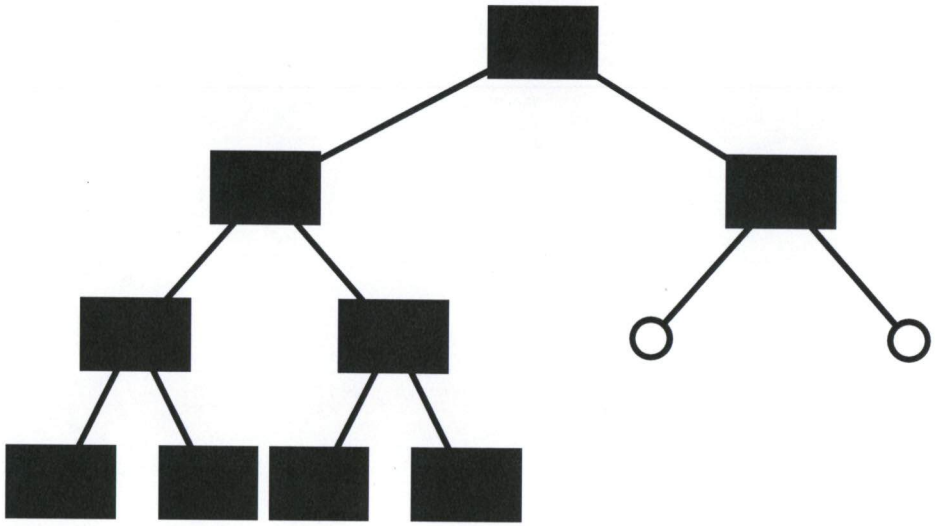


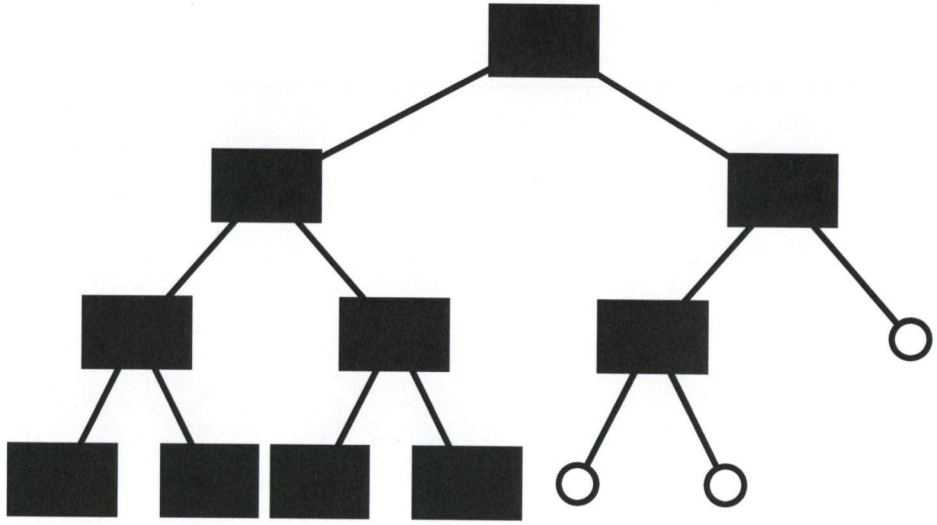


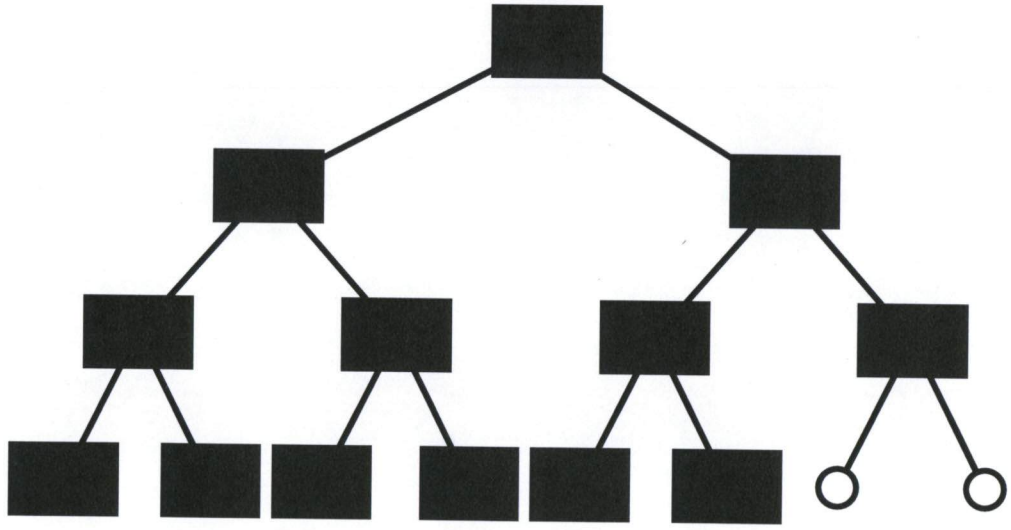


DFS on a depth-3 binary tree, contd.









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!

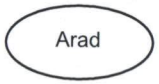
Depth-limited search

= depth-fir

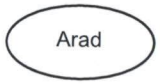
Iterative deepening search

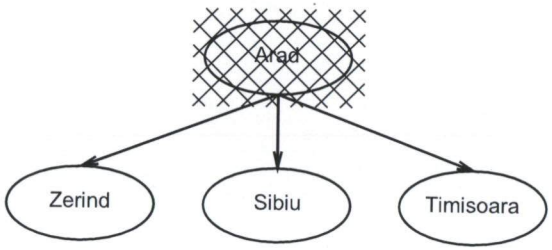
function ITERATIVE-DEEPENING-SEARCH(*pro*

Iterative deepening search $l = 0$

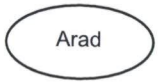


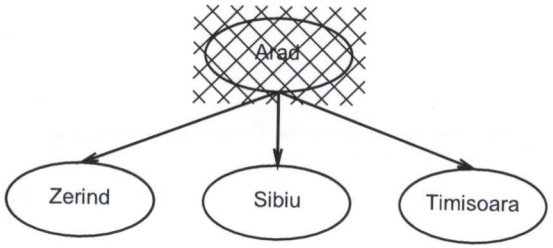
Iterative deepening search $l = 1$

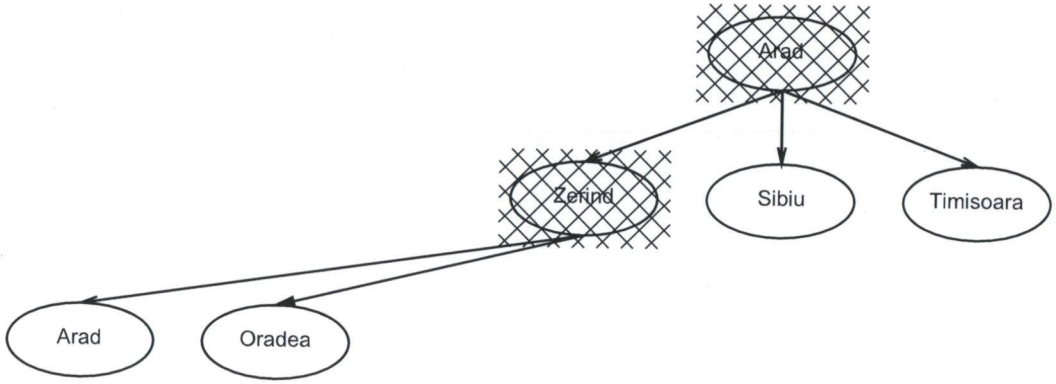


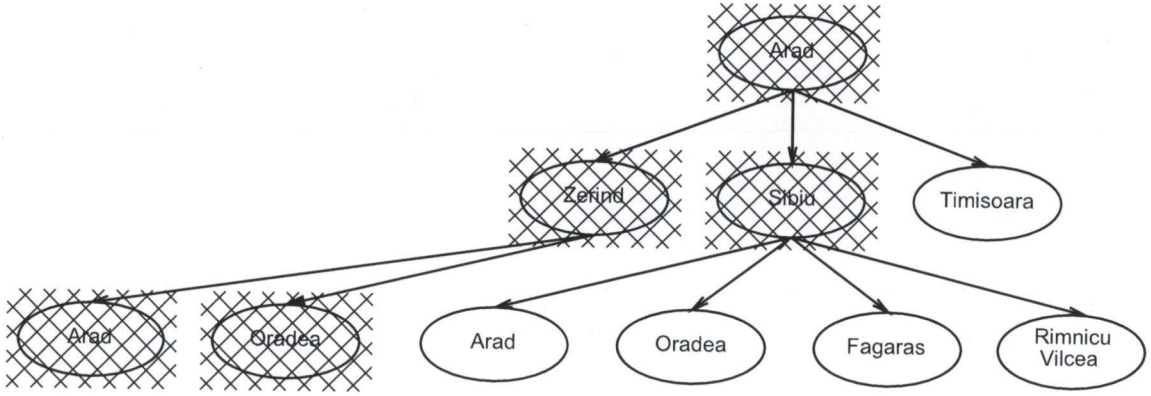

Arad
Zerind
Sibiu
Timisoara

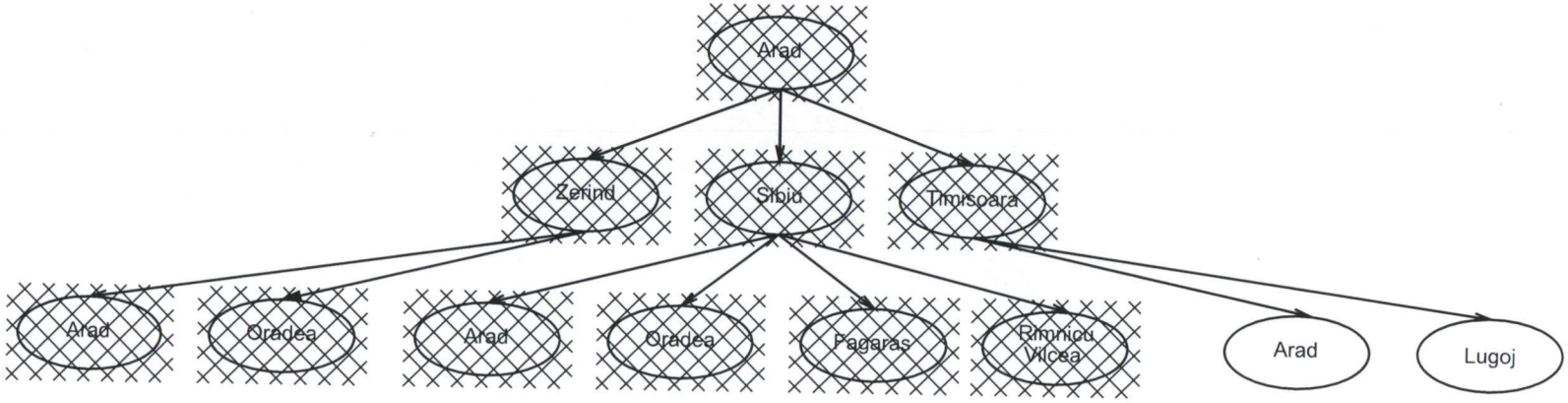
Iterative deepening search $l = 2$




Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu
Vilcea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu Vilcea
Arad
Lugoj

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

Informed search algorithms

Chapter 4, Sections 1–2, 4

Outline

- ♢ Best-first search
- ♢ A^* search
- ♢ Heuristics
- ♢ Hill-climbing
- ♢ Simulated annealing

Best-first search

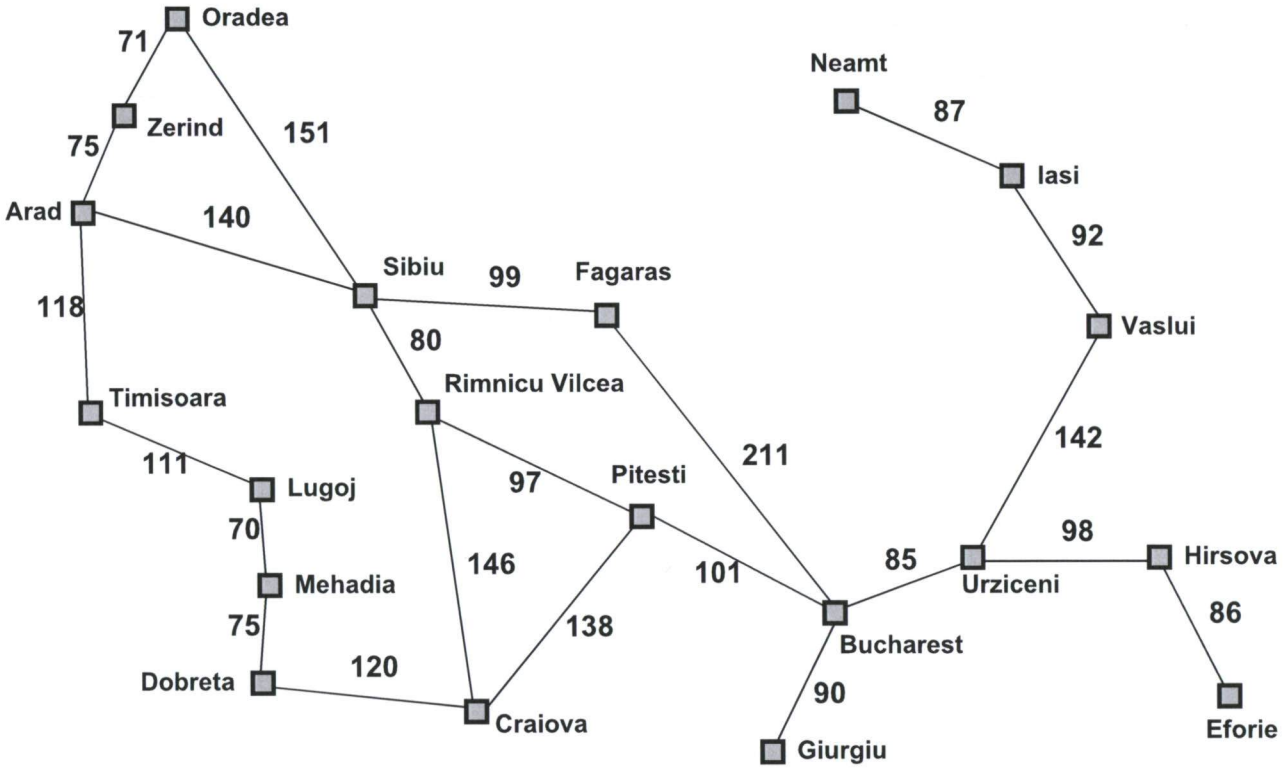
Idea: use an *evaluation function* for each node
– estimate of "desirability"

⇒ Expand most desirable unexpanded node

Implementation:
QUEUEINGFN = insert successors in decreasing order of desirability

Special cases:
greedy search
A* search

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

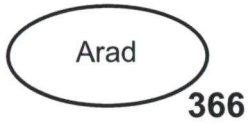
Greedy search

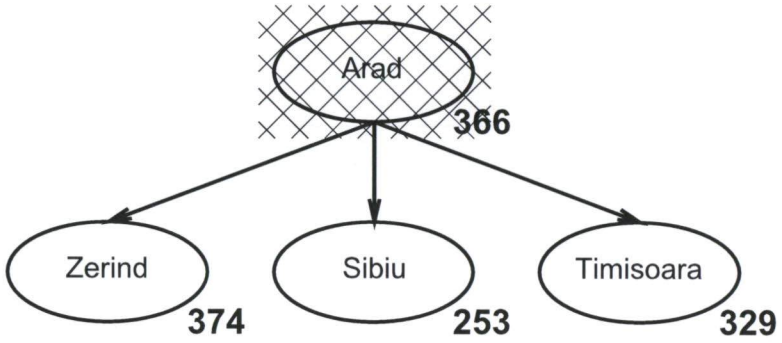
Evaluation function $h(n)$ (heuristic)
= estimate of cost from n to $goal$

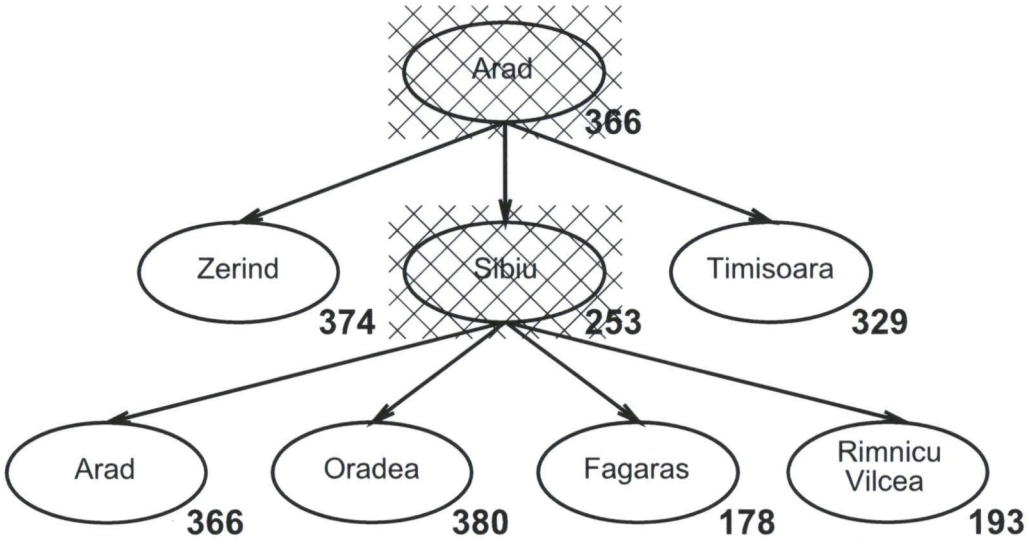
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

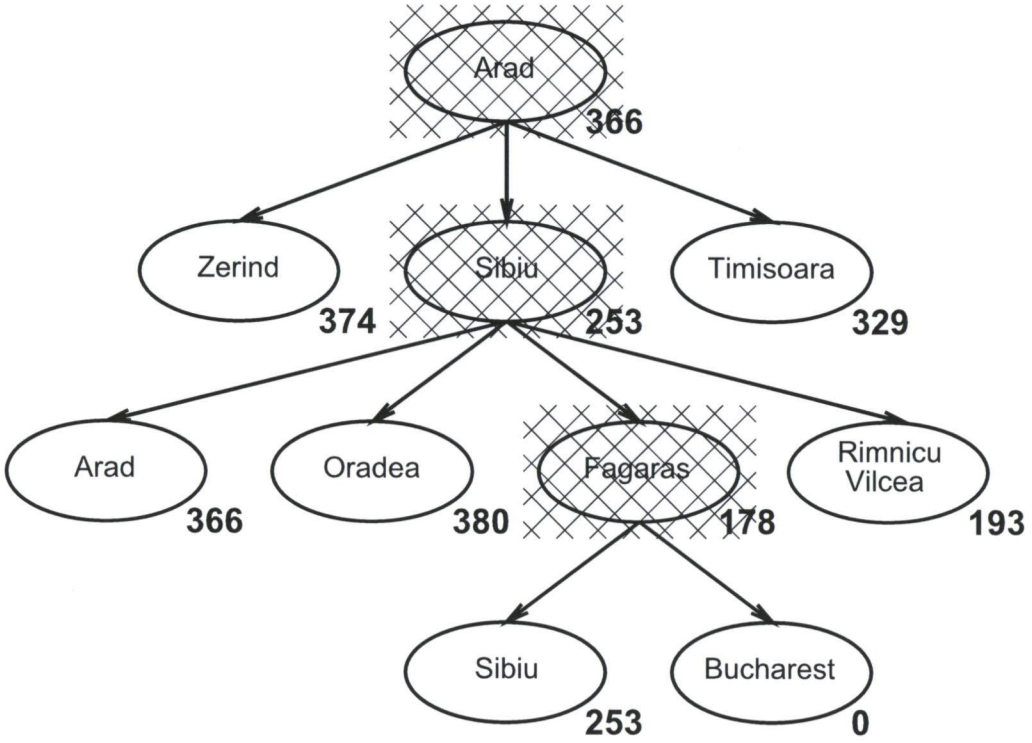
Greedy search expands the node that *appears* to be closest to goal

Greedy search example




Arad
366
Zerind
374
Sibiu
253
Timisoara
329


Arad
366
Zerind
374
Sibiu
253
Timisoara
329
Arad
366
Oradea
380
Fagaras
178
Rimnicu
Vilcea
193


Arad
366
Zerind
374
Sibiu
253
Timisoara
329
Arad
366
Oradea
380
Fagaras
178
Rimnicu
Vilcea
193
Sibiu
253
Bucharest
0

Properties of greedy search

Complete??

Time??

Space??

Optimal??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

A* search

Idea: avoid expanding paths that are already expensive

Evaluation function $f(n) = g(n) + h(n)$

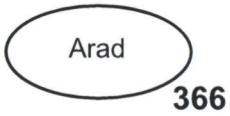
$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

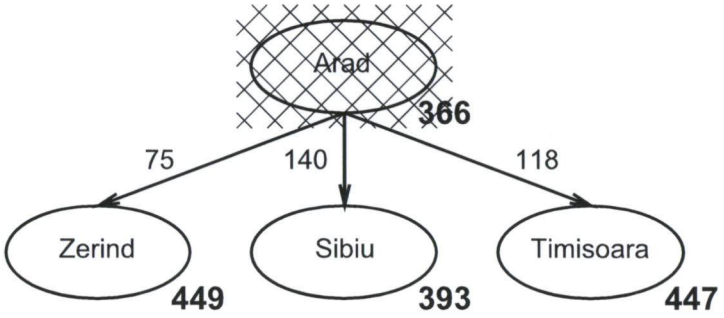
A* search uses an *admissible* heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n.

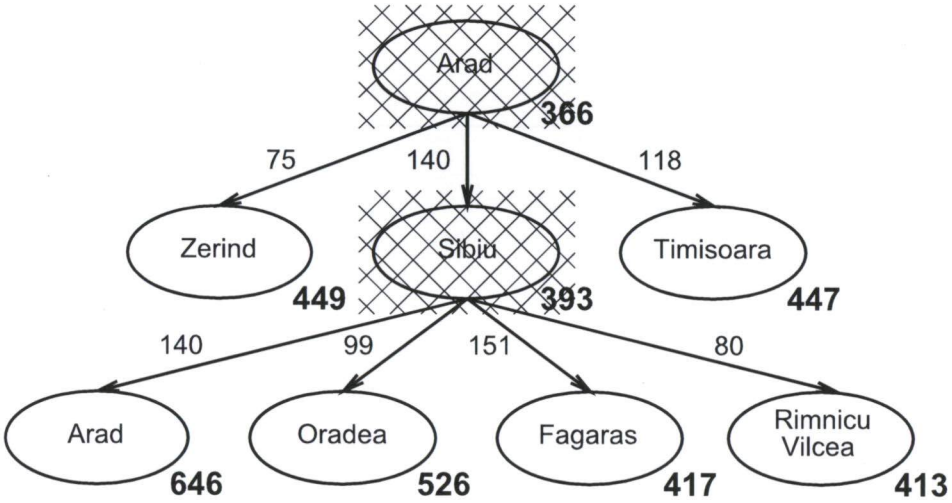
E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

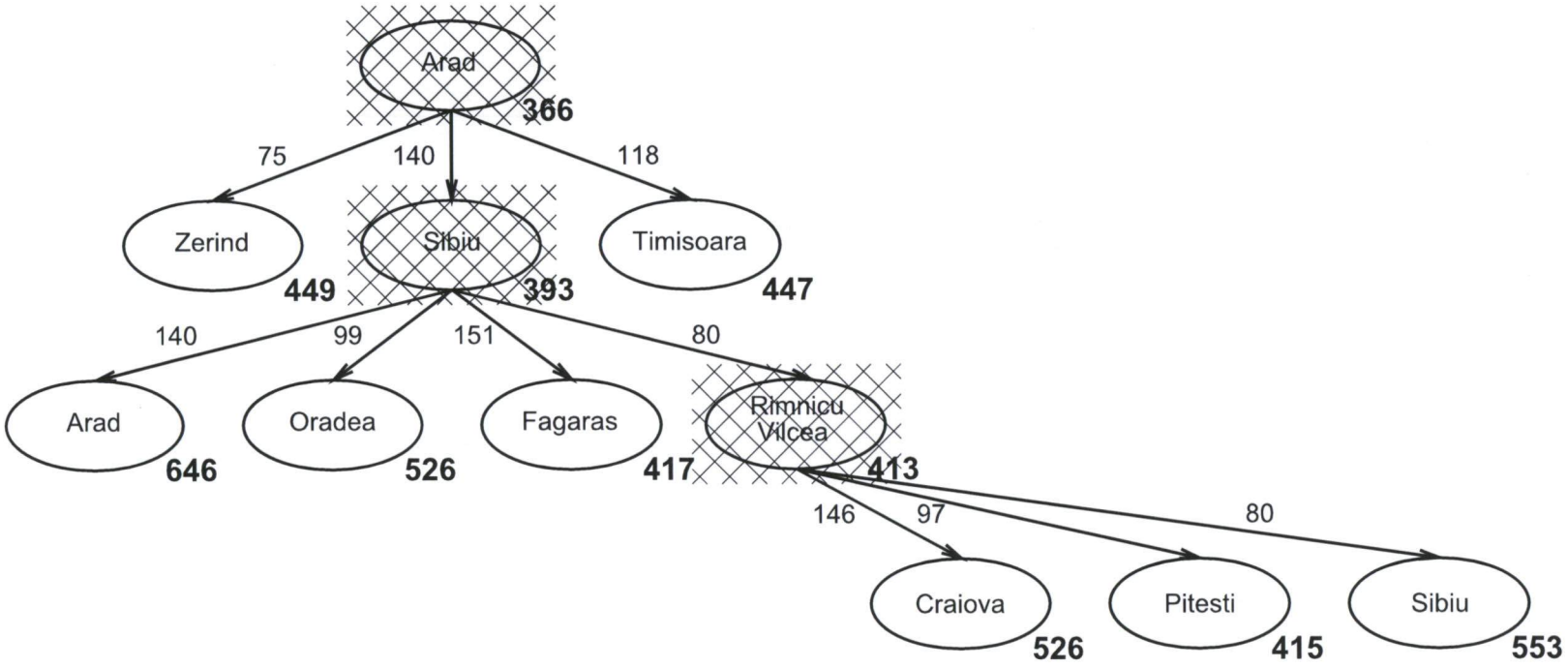
Theorem: A* search is optimal

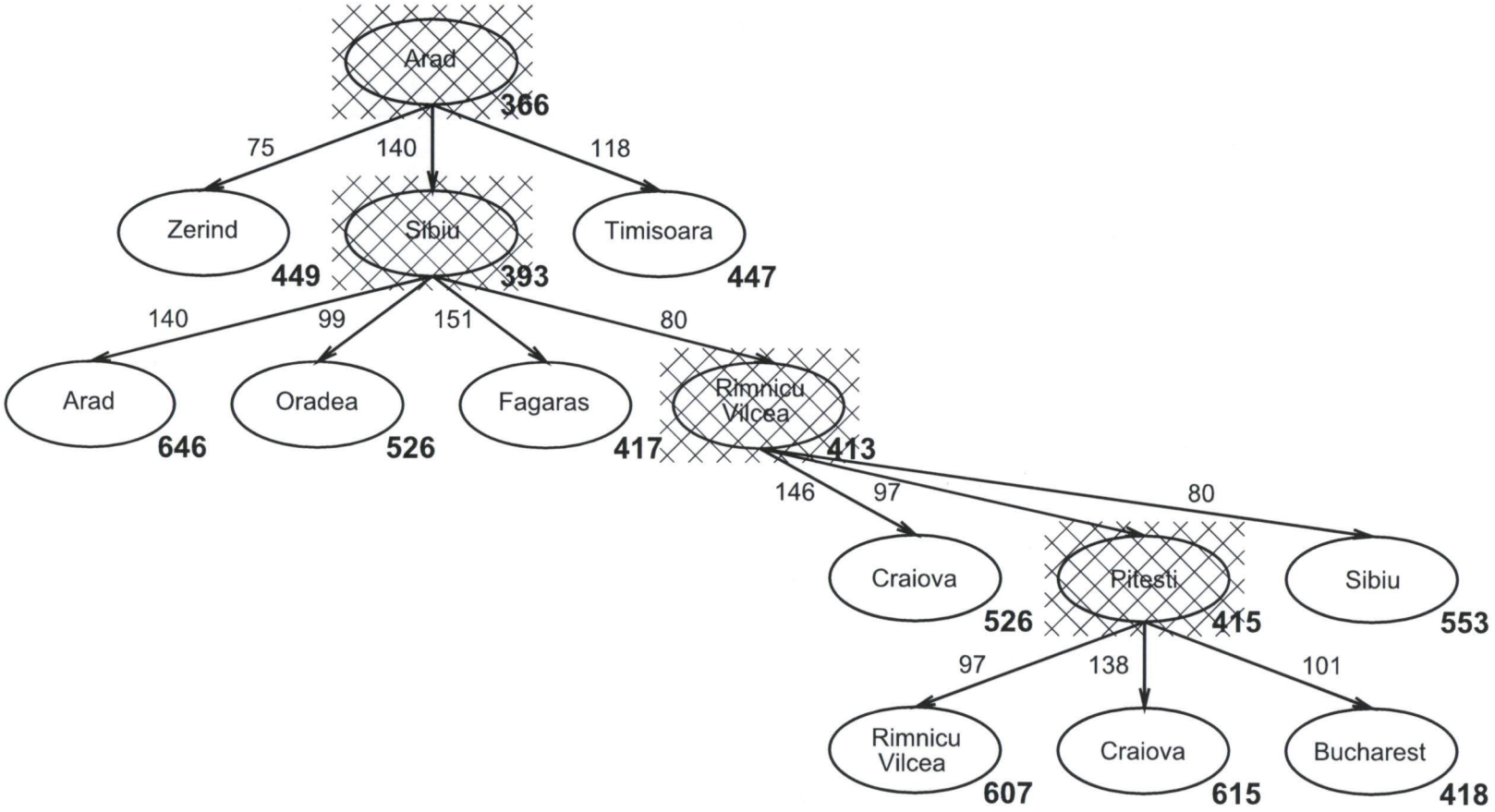
A* search example

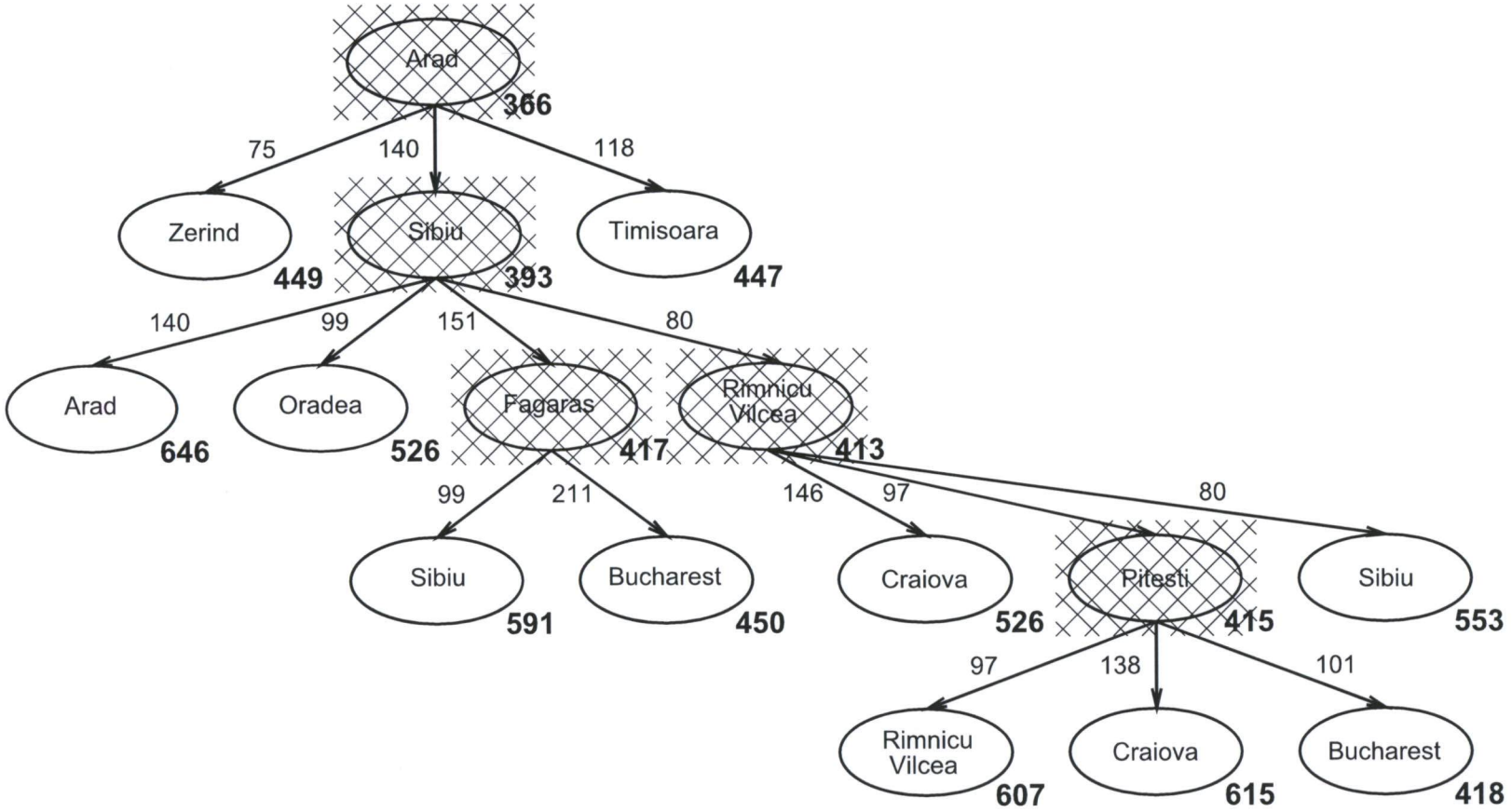



Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu
Vilcea
413

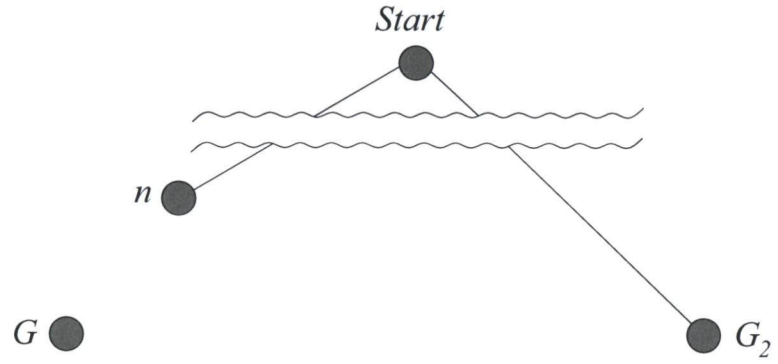

Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu Vilcea
413
146
97
80
Craiova
526
Pitesti
415
Sibiu
553


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu Vilcea
413
146
97
80
Craiova
526
Pitesti
415
Sibiu
553
97
138
101
Rimnicu Vilcea
607
Craiova
615
Bucharest
418


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu Vilcea
413
99
211
146
97
80
Sibiu
591
Bucharest
450
Craiova
526
Pitesti
415
Sibiu
553
97
138
101
Rimnicu Vilcea
607
Craiova
615
Bucharest
418

Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



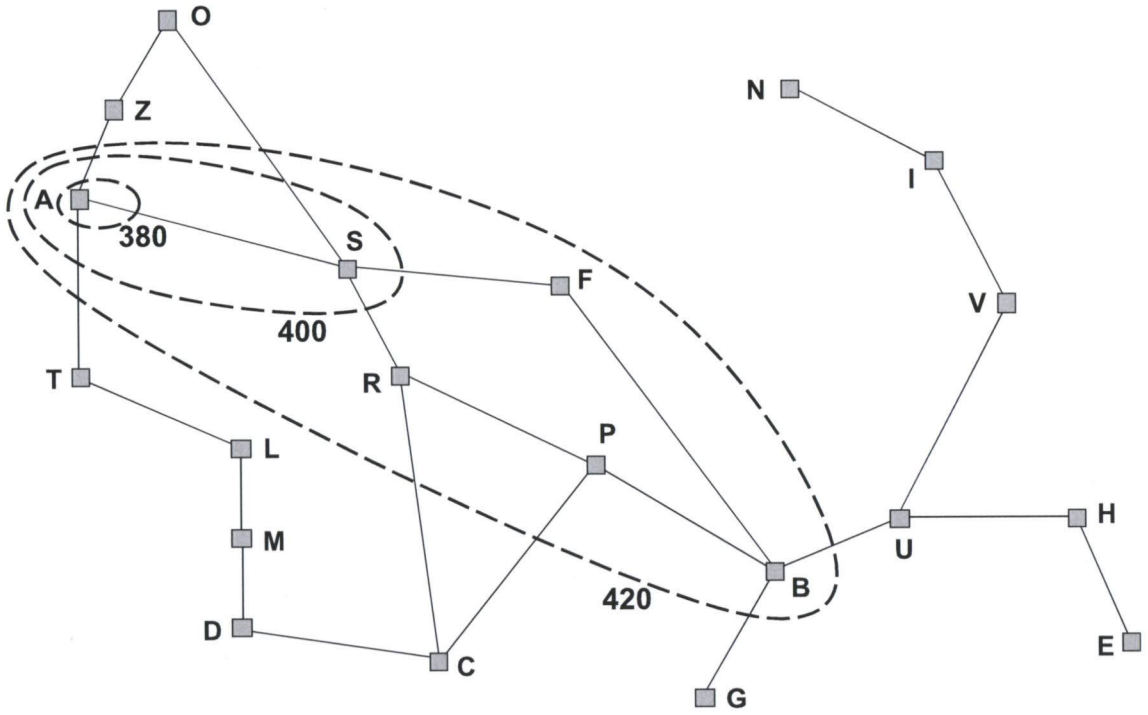
$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in h $\times$ length of soln.]

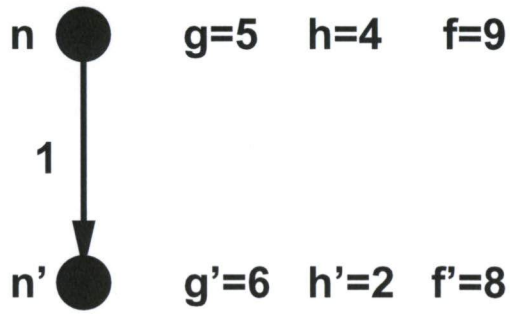
Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Proof of lemma: Pathmax

For some admissible heuristics, f may *decrease* along a path

E.g., suppose n' is a successor of n



But this throws away information!
$f(n) = 9 \Rightarrow$ true cost of a path through n is ≥ 9
Hence true cost of a path through n' is ≥ 9 also

Pathmax modification to A*:
Instead of $f(n') = g(n') + h(n')$, use $f(n') = max(g(n') + h(n'), f(n))$

With pathmax, f is always nondecreasing along any path

Admissible heuristics

E.g., for the 8-pu

Admissible heuristics

E.g., for the 8-pu

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 *dominates* h_1 and is better for search

Typical search costs:

$d = 14$ IDS $=$ 3,473,941 nodes
A$^*(h_1) =$ 539 nodes
A$^*(h_2) =$ 113 nodes
$d = 14$ IDS $=$ too many nodes
A$^*(h_1) =$ 39,135 nodes
A$^*(h_2) =$ 1,641 nodes

Relaxed problems

Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem

If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution

For TSP: let path be *any* structure that connects all cities

$\Longrightarrow$ minimum spanning tree heuristic

Iterative improvement algorithms

In many optimization problems, *path* is irrelevant;
the goal state itself is the solution

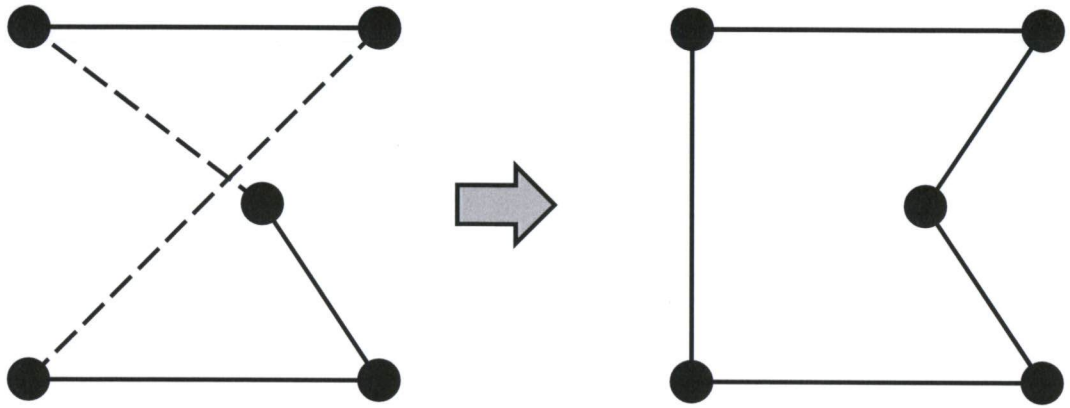
Then state space = set of "complete" configurations;
find *optimal* configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., n-queens

In such cases, can use *iterative improvement* algorithms;
keep a single "current" state, try to improve it

Constant space, suitable for online as well as offline search

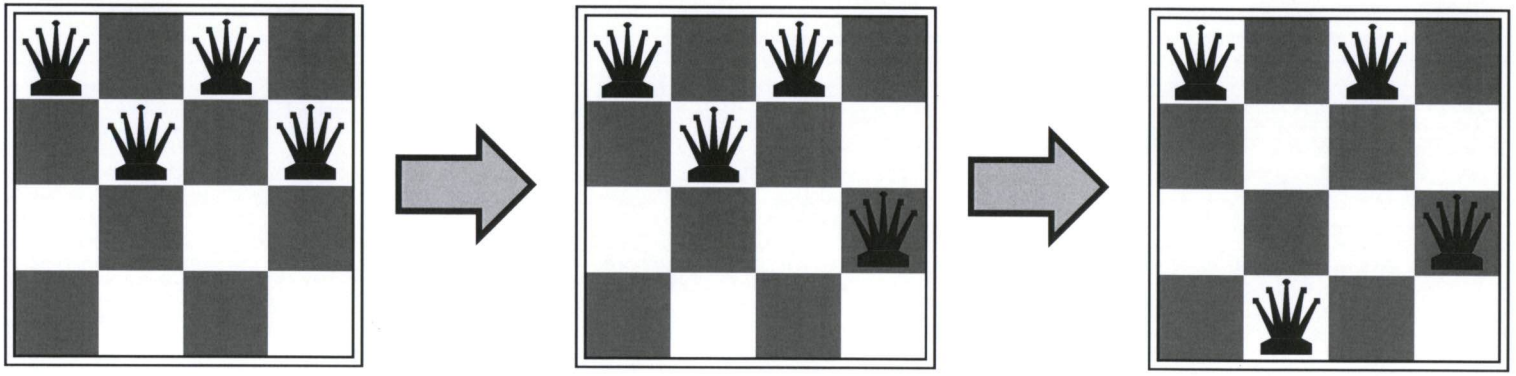
Example: Travelling Salesperson Problem

Find the shortest tour that visits each city exactly once



Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal



Hill-climbing (or gradient ascent/descent)

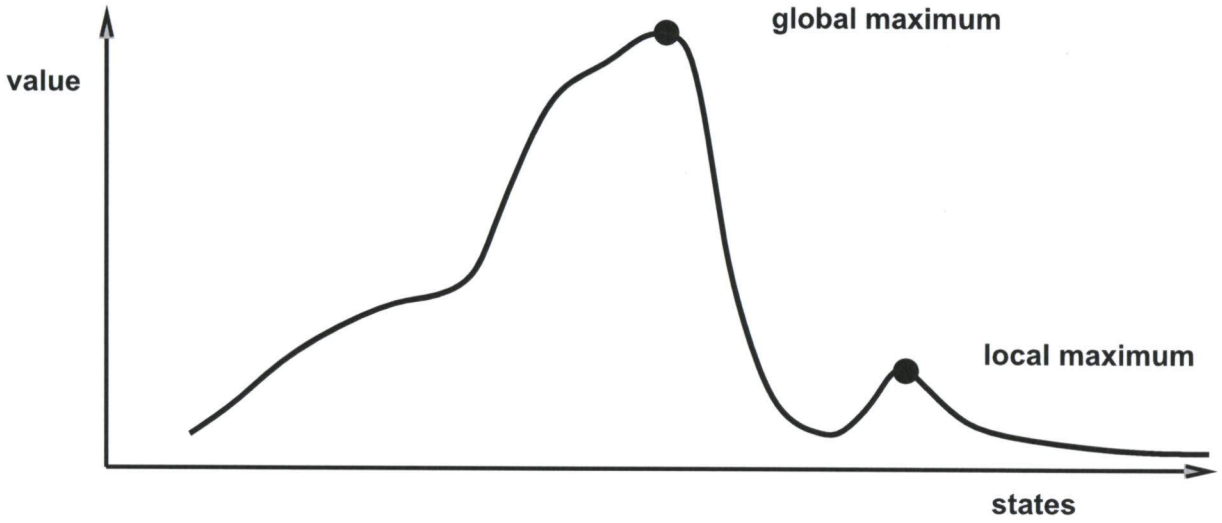
"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                     next, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next ← a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current ← next
    end
```

Hill-climbing contd.

Problem: depending on initial state, can get stuck on local maxima



Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling the probability of downward steps

  current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if T=0 **then return** *current*
    next $\leftarrow$ a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
    else *current* $\leftarrow$ *next* only with probability e

Properties of simulated annealing

At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.