

# PROBLEM SOLVING AND SEARCH

## CHAPTER 3, SECTIONS 1–4

# 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  $\leftarrow$  UPDATE-STATE(state, p)
    if s is empty then
        g  $\leftarrow$  FORMULATE-GOAL(state)
        problem  $\leftarrow$  FORMULATE-PROBLEM(state, g)
        s  $\leftarrow$  SEARCH(problem)
        action  $\leftarrow$  RECOMMENDATION(s, state)
        s  $\leftarrow$  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:

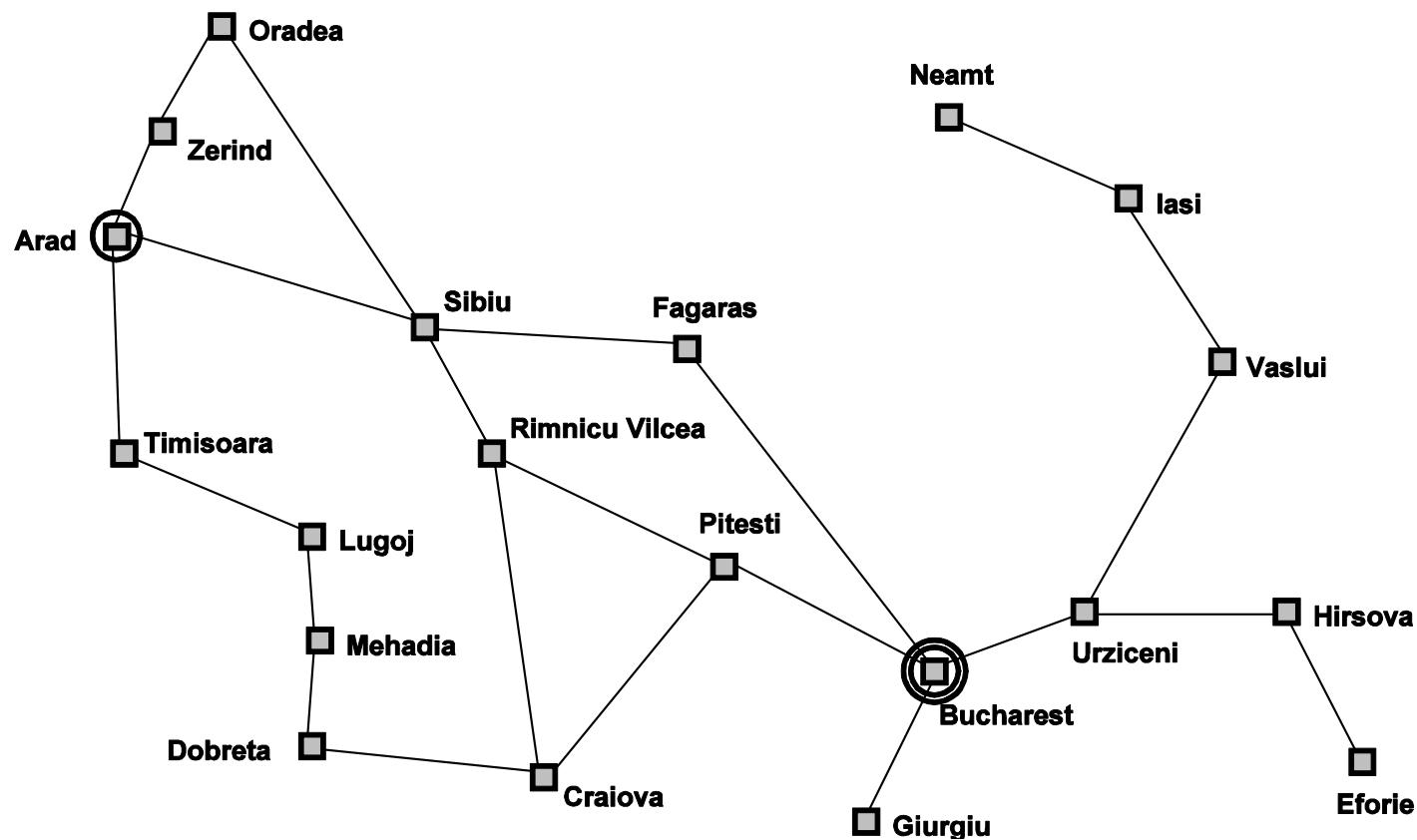
*states:* various cities

*operators:* drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

## Example: Romania



## Single-state problem formulation

A single-state problem is a problem where we can clearly observe what state we are in (as opposed to a partially observable environment where we cannot uniquely observe what state we are in)

A *problem* is defined by four items:

*initial state* e.g., “at Arad”

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

*goal test*, can be

explicit, e.g.,  $x = \text{“at Bucharest”}$

implicit, e.g., *Checkmate* in chess

*path cost* (additive)

e.g., sum of distances, number of actions executed, etc.

A *solution* is a sequence of actions leading from the initial state to a goal state (Note: we sometimes refer to actions as “operators” )

## Selecting a state space

Real world is absurdly complex

⇒ state space must be *abstracted* for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

e.g., “Arad → Zerind” represents a complex set  
of possible routes, detours, rest stops, etc.

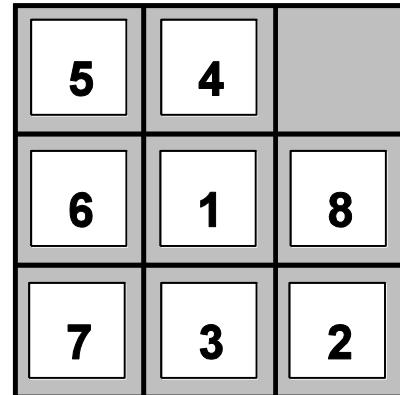
For guaranteed realizability, any real state “in Arad”  
must get to *some* real state “in Zerind”

(Abstract) solution =

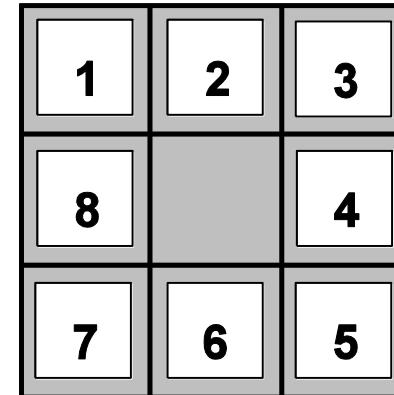
set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!

## Example: The 8-puzzle



Start State



Goal State

states??

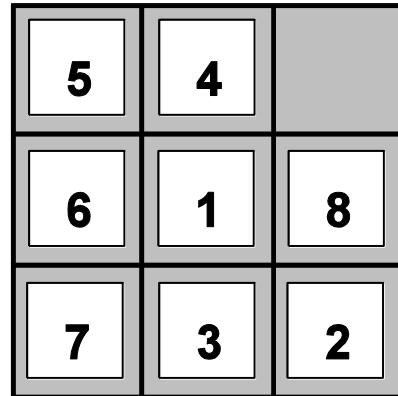
actions??

goal test??

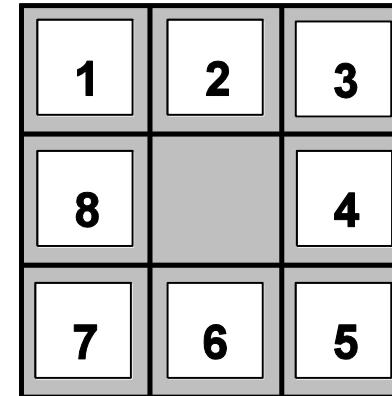
path cost??

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

## Example: The 8-puzzle



Start State



Goal State

states??: integer locations of tiles (ignore intermediate positions)

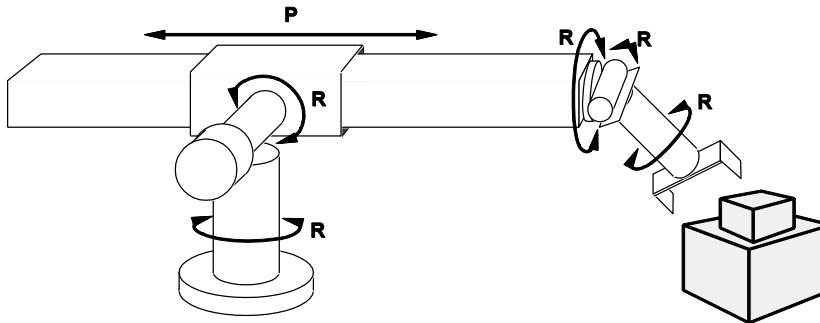
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

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

## Example: 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

# Search algorithms

Basic idea:

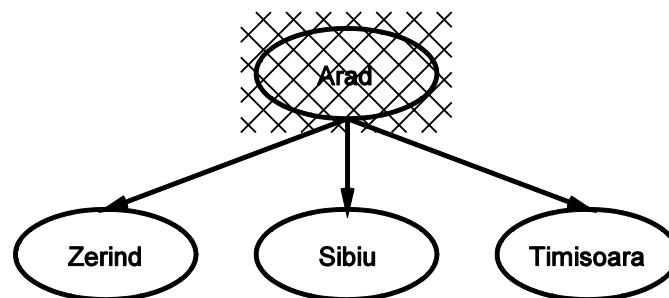
offline, simulated exploration of state space  
by generating successors of already-explored states  
(a.k.a. *expanding states*)

```
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
```

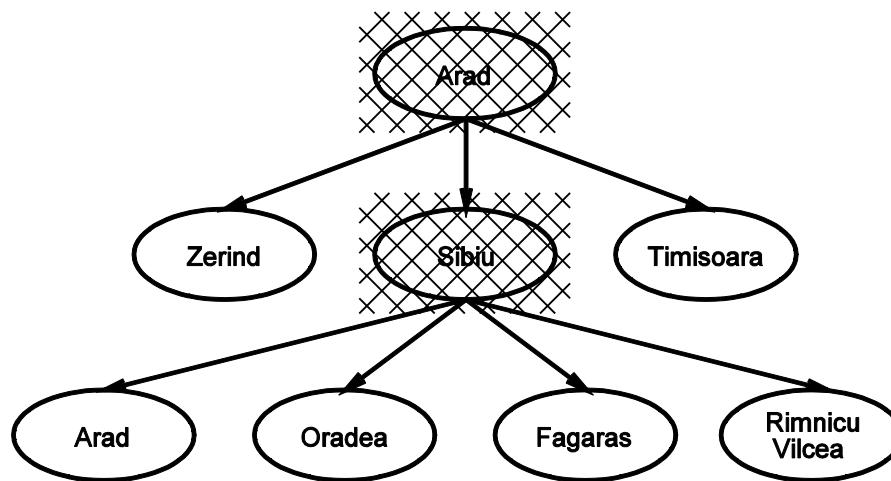
# General search example



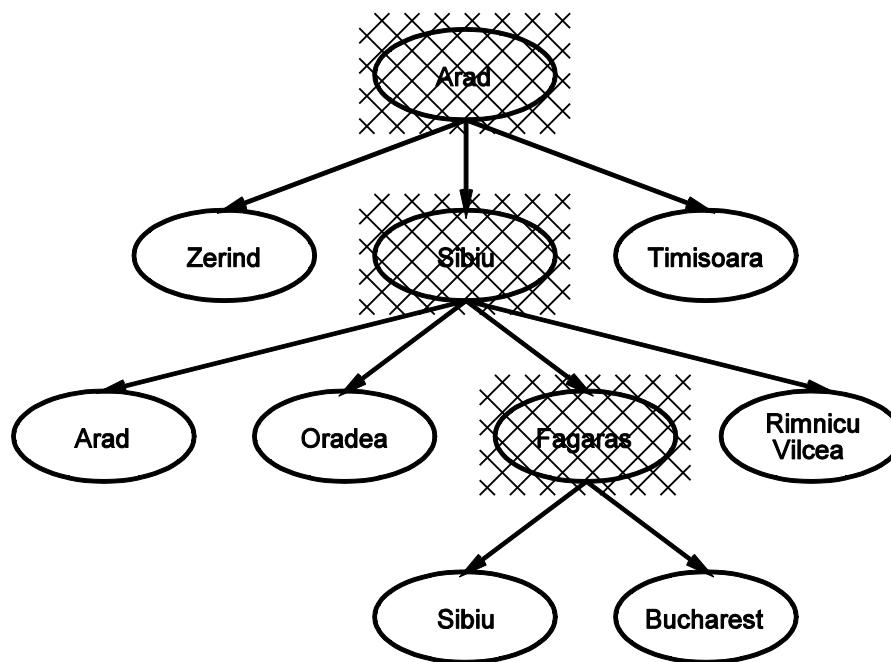
## General search example



# General search example



# General search example



# Implementation of search algorithms

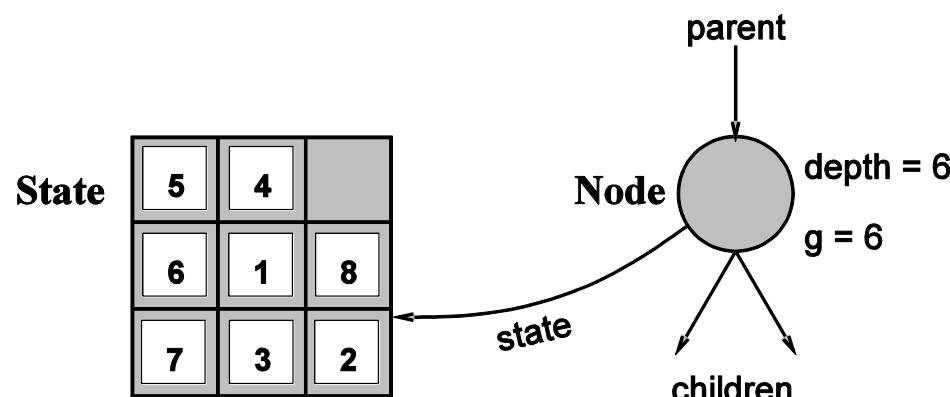
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure
  nodes  $\leftarrow$  MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem])))
  loop do
    if nodes is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes  $\leftarrow$  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 various fields and using OPERATORS (or ACTIONS) of 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  $\infty$ )

## 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

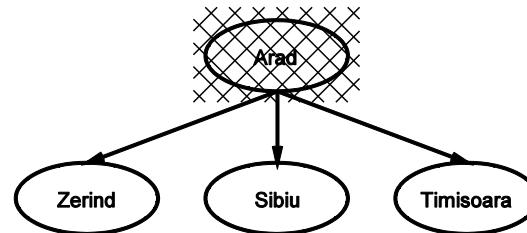


# Breadth-first search

Expand shallowest unexpanded node

Implementation:

QUEUEINGFN = put successors at end of queue

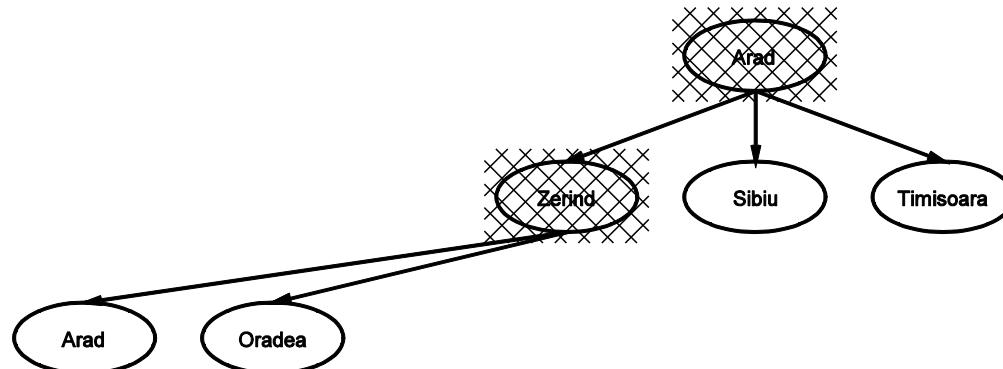


# Breadth-first search

Expand shallowest unexpanded node

Implementation:

QUEUEINGFN = put successors at end of queue

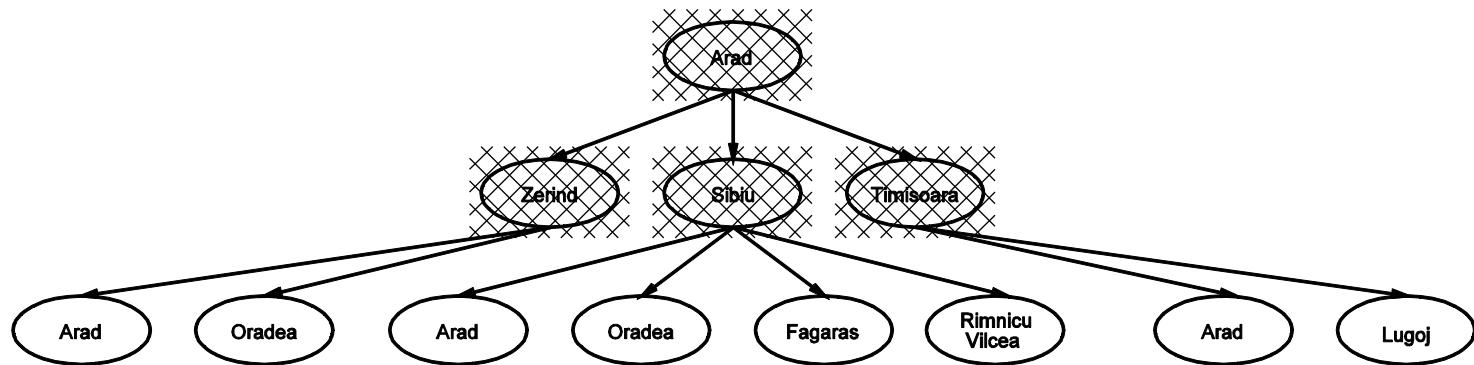


# Breadth-first search

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)

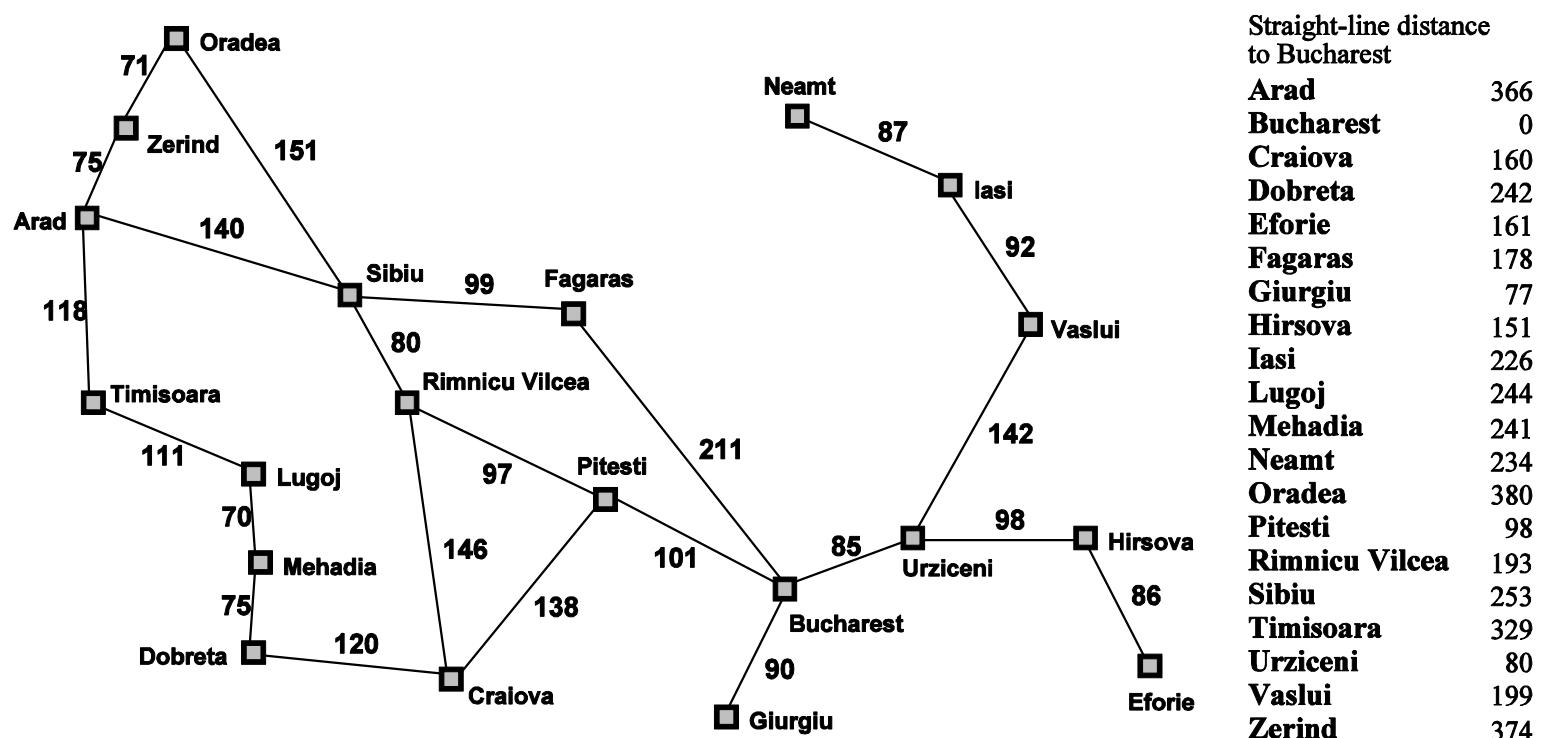
Time??  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$ , i.e., exponential in  $d$

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



# Uniform-cost search

Expand least-cost unexpanded node

Implementation:

QUEUEINGFN = insert in order of increasing path cost

Let  $g(n)$  denote the path cost of node  $n$ , i.e., the sum of the costs of the actions that led from the root node to  $n$

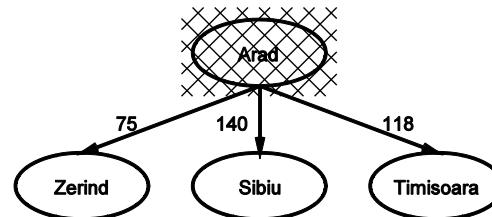


# Uniform-cost search

Expand least-cost unexpanded node

Implementation:

QUEUEINGFN = insert in order of increasing path cost

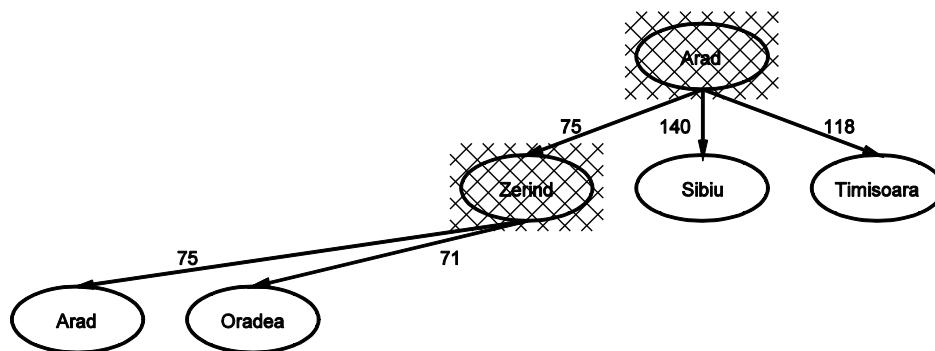


# Uniform-cost search

Expand least-cost unexpanded node

Implementation:

QUEUEINGFN = insert in order of increasing path cost

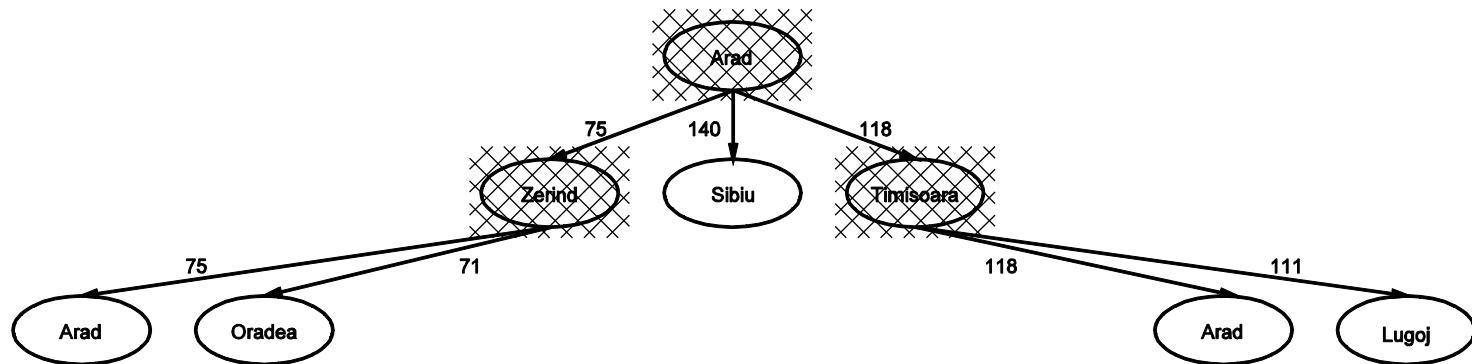


# Uniform-cost search

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

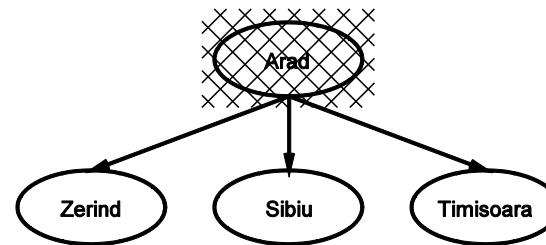


# Depth-first search

Expand deepest unexpanded node

Implementation:

QUEUEINGFN = insert successors at front of queue

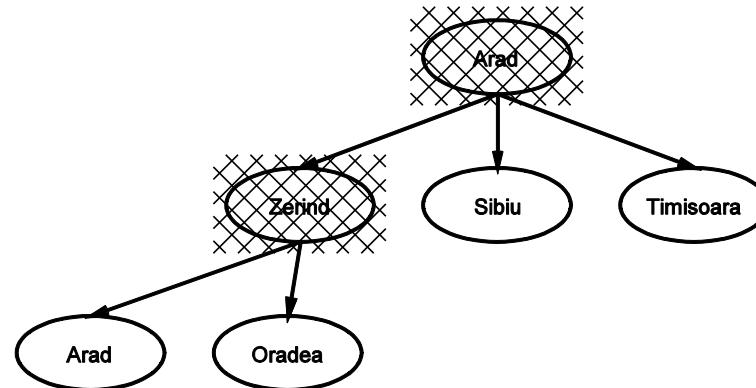


# Depth-first search

Expand deepest unexpanded node

Implementation:

QUEUEINGFN = insert successors at front of queue

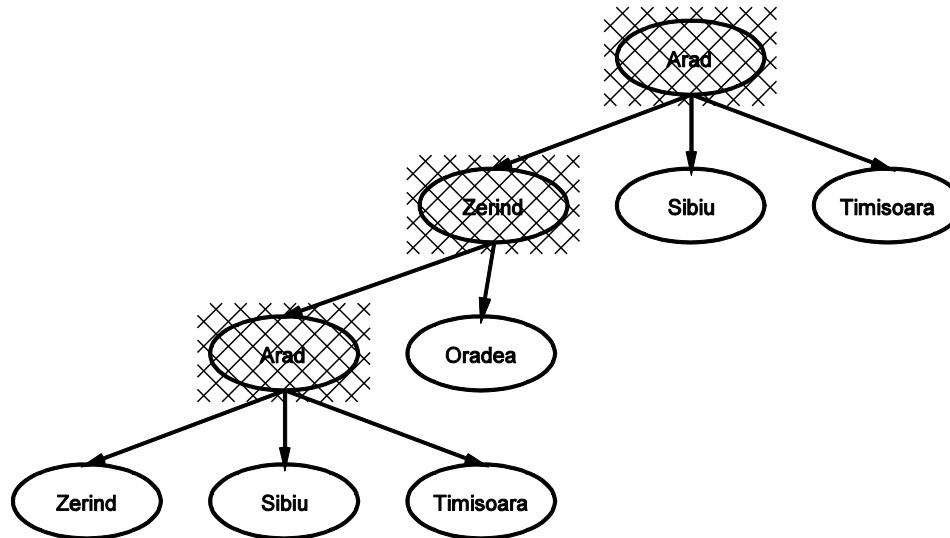


# Depth-first search

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)

## 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  
⇒ 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  $\leftarrow$  0 to  $\infty$  do
        result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
        if result  $\neq$  cutoff then return result
    end
```

Note that Depth-Limited-Search returns the result “cutoff” if it reached the depth limit of the search without finding a solution, but there were still unexplored children at greater depths. In that case, it is still worth increasing the depth limit and applying another iteration of Depth-Limited-Search.

This is in contrast to the situation where Depth-Limited-Search explored all possible nodes before the depth limit, and there were no more unexplored children to search. Hence the search should terminate.

## Properties of iterative deepening search

Complete?? Yes

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

Space??  $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

## Bidirectional Search

Search simultaneously forwards from the start point, and backwards from the goal, and stop when the two searches meet in the middle.

Problems: generate predecessors; many goal states; efficient check for node already visited by other half of the search; and, what kind of search.

## Properties of Bidirectional Search

Complete?? Yes

Time??  $O(b^{\frac{d}{2}})$

Space??  $O(b^{\frac{d}{2}})$

Optimal?? Yes (if done with correct strategy - e.g. breadth first).

# 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

Examples of skills expected:

- ◊ Formulate single-state search problem
- ◊ Apply a search strategy to solve problem
- ◊ Analyse complexity of a search strategy