Artificial Intelligence

Lecturer 3 – Search Algorithms

Brigitte Jaumard
Dept of Computer Science and Software Engineering
Concordia University
Montreal (Quebec) Canada

Outline

- Problem-Solving Agents
- Problem Types
- Problem Formulation
- Example Problems
- Basic Search Algorithms
  - Graph search
  - Best-first search
  - A* search

PROBLEM-SOLVING

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

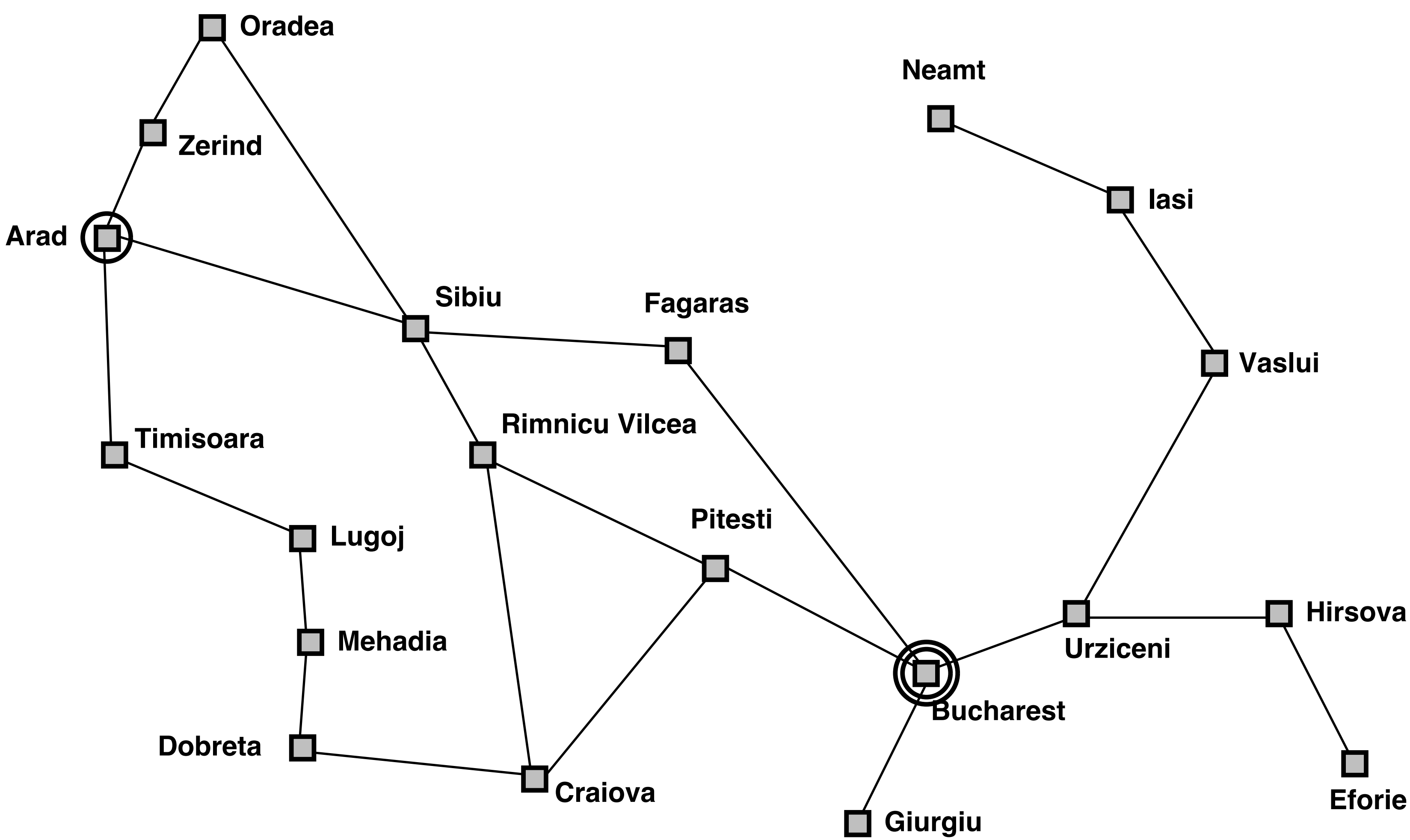
Formulate problem:

states: various cities
operators: drive between cities

Find solution:

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

Example: Romania



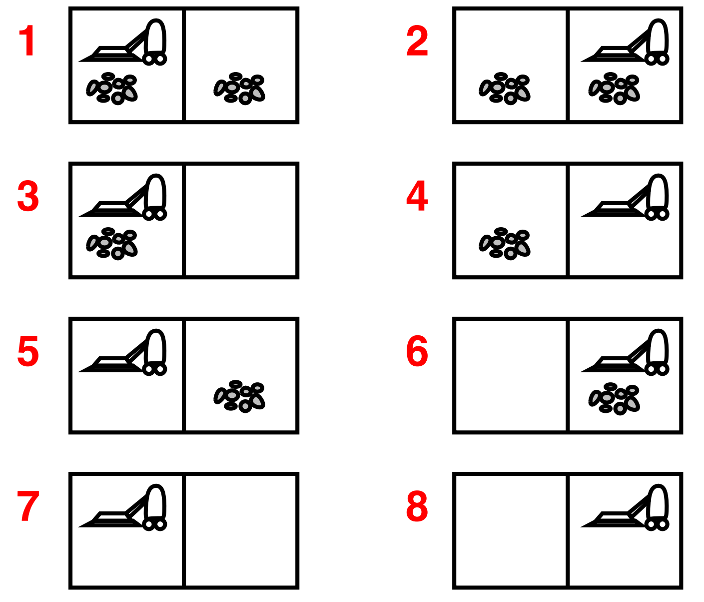
PROBLEM-TYPES

Problems Types

- **Deterministic, fully observable** ⇒ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- **Non-observable** ⇒ **conformant problem**
  - Agent may have no idea where it is; solution (if any) is a sequence
- **Nondeterministic and/or partially observable** ⇒ **contingency problem**
  - percepts provide new information about current state
  - solution is a contingent plan or a policy
  - often interleave search, execution
- **Unknown state space** ⇒ **exploration problem** ("online")

Example: Vacuum World

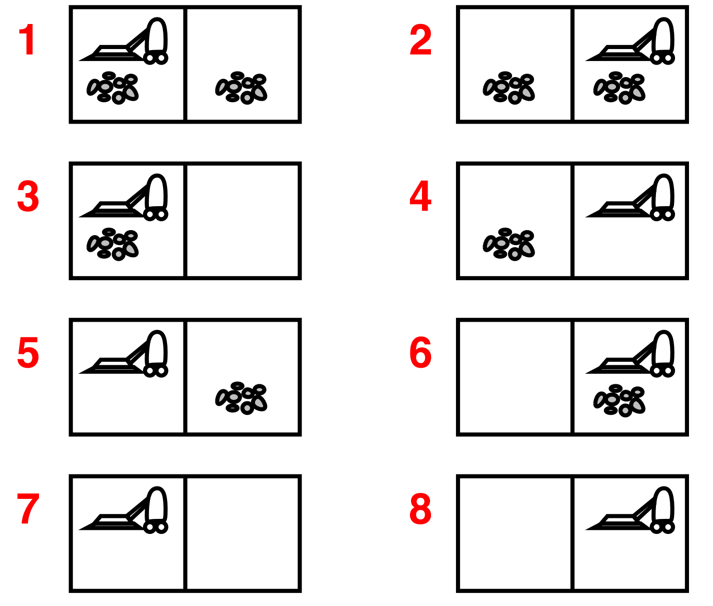
Single-state, start in #5. Solution??



Example: Vacuum World (Cont'd)

Single-state, start in #5. Solution??
$[Right, Suck]$

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

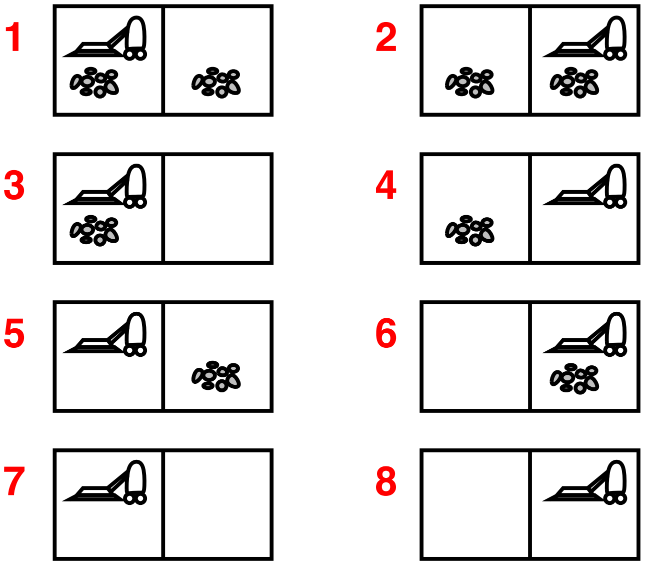


Example: Vacuum World (Cont'd)

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$

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

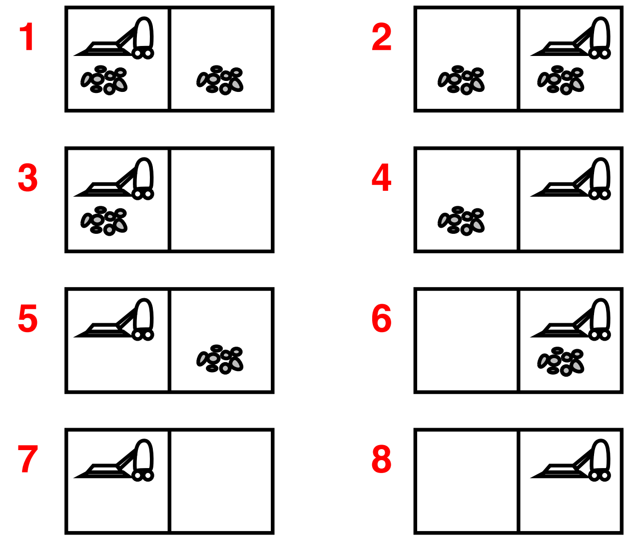


Example: vacuum world (Cont'd)

Single-state, start in $\#5$. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$

Contingency, start in $\#5$
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
$[Right, \textbf{if } dirt \textbf{ then } Suck]$



PROBLEM FORMULATION

Single-state problem formulation

A *problem* is defined by four items:

<u>*initial state*</u> e.g., "at Arad"

<u>*operators*</u> (or *successor function* $S(x)$)
e.g., Arad $\rightarrow$ Zerind Arad $\rightarrow$ Sibiu etc.

<u>*goal test*</u>, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., $NoDirt(x)$

<u>*path cost*</u> (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

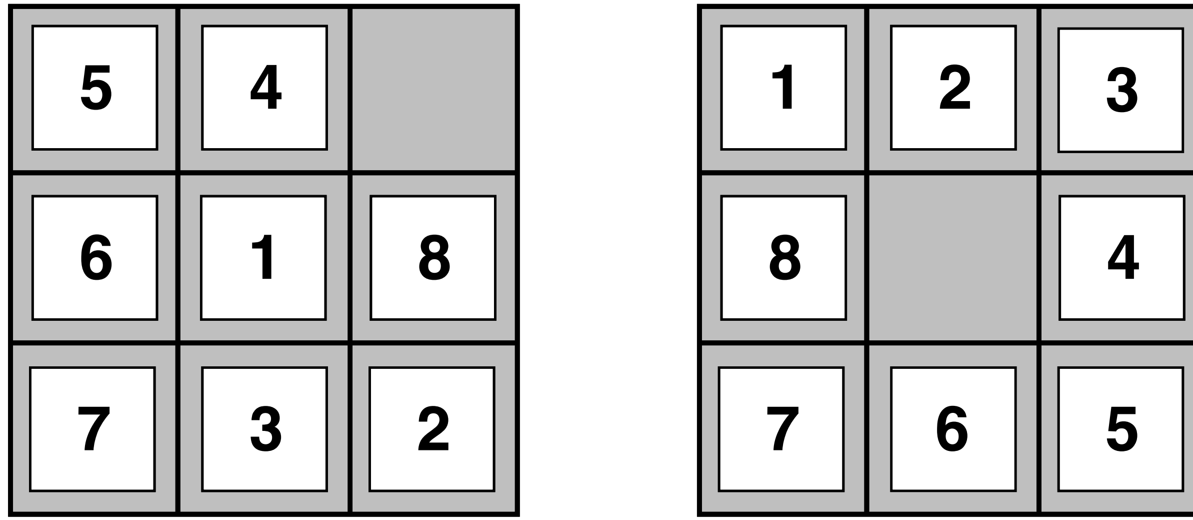
(Abstract) operator = complex combination of real actions
e.g., "Arad $\rightarrow$ Zerind" represents a complex set
of possible routes, detours, rest stops, etc.
For guaranteed realizability, any real state "in Arad"
must get to *some* real state "in Zerind"

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

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

EXAMPLE OF PROBLEMS

Example: The 8-Puzzle



Start State **Goal State**

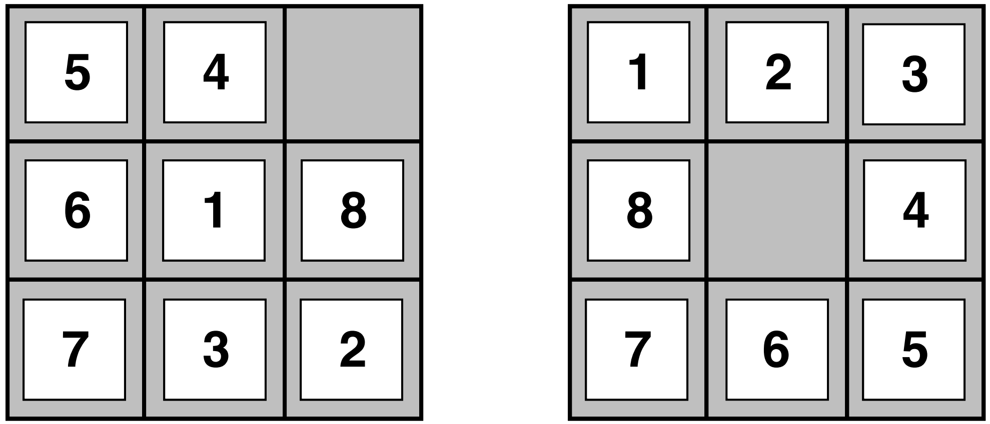
states??

operators??

goal test??

path cost??

Example: The 8-puzzle

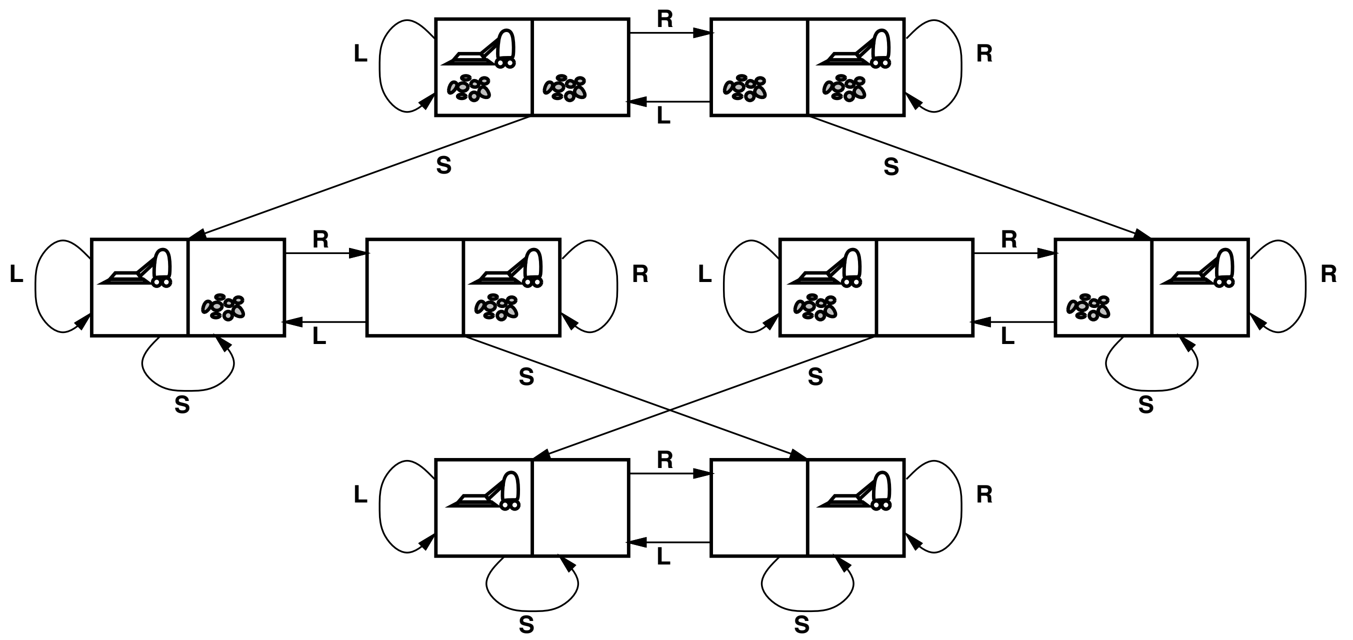


Start State **Goal State**

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

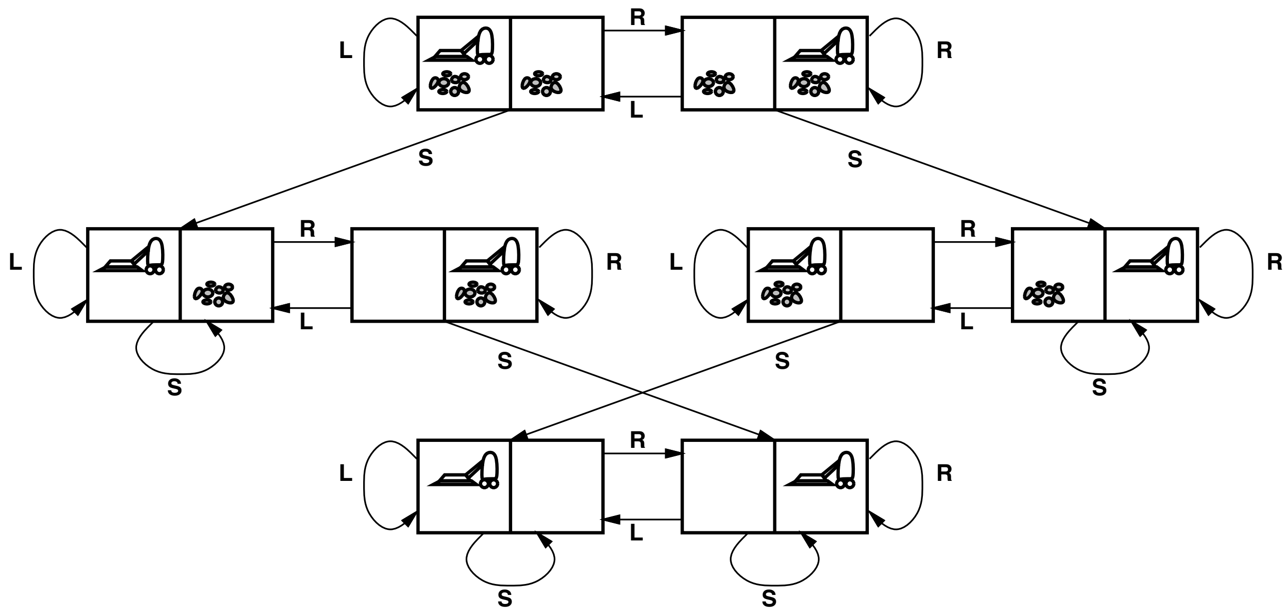
[Note: optimal solution of n-Puzzle family is NP-hard]

Example: Vacuum World State Space Graph



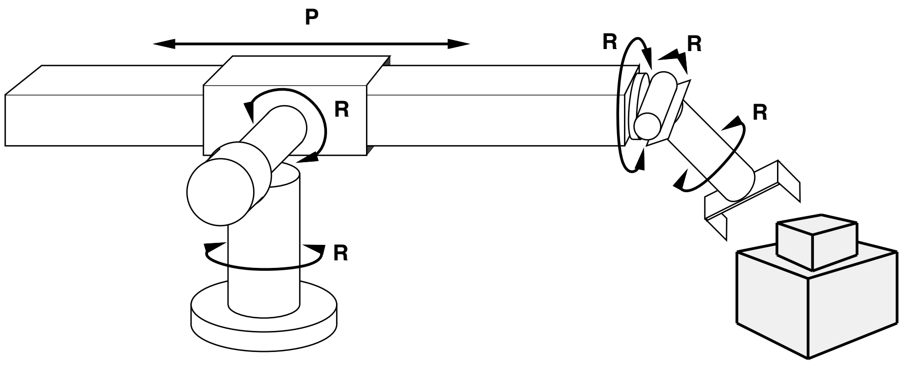
states??
operators??
goal test??
path cost??

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

BASIC SEARCH ALGORITHMS

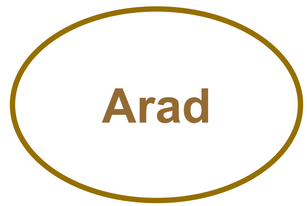
Search Algorithms

Basic idea:

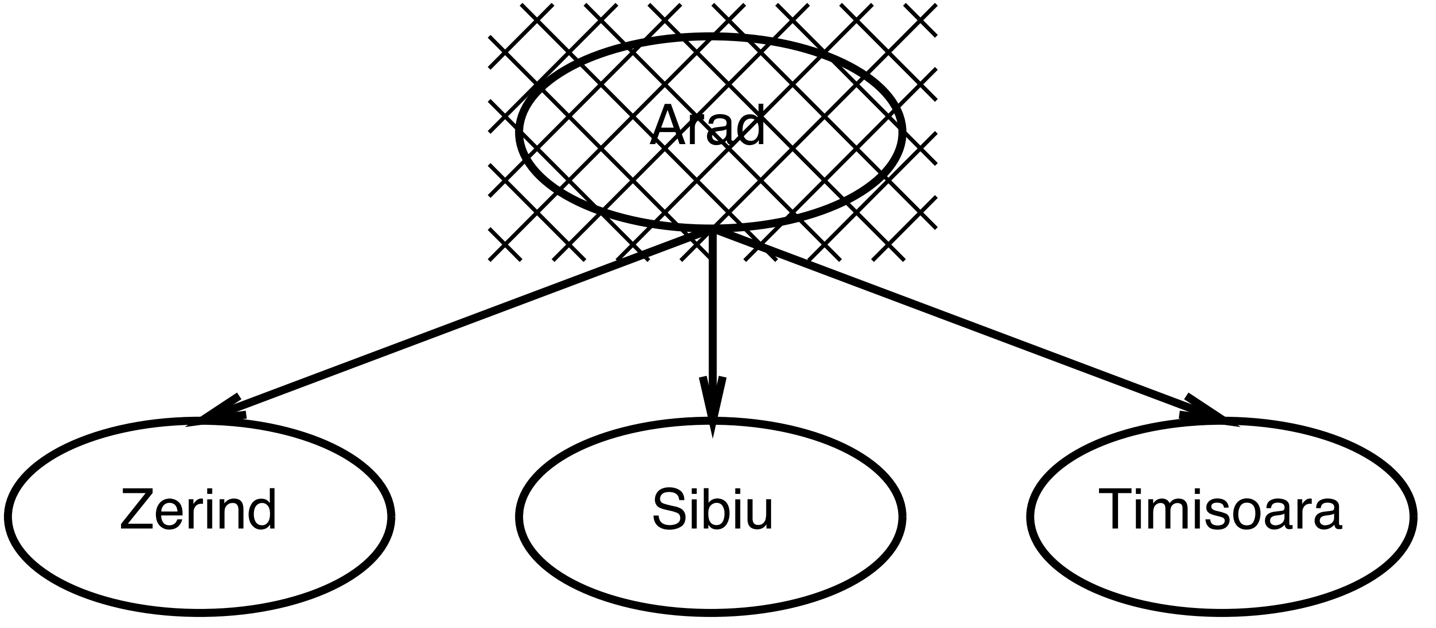
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

```
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
```

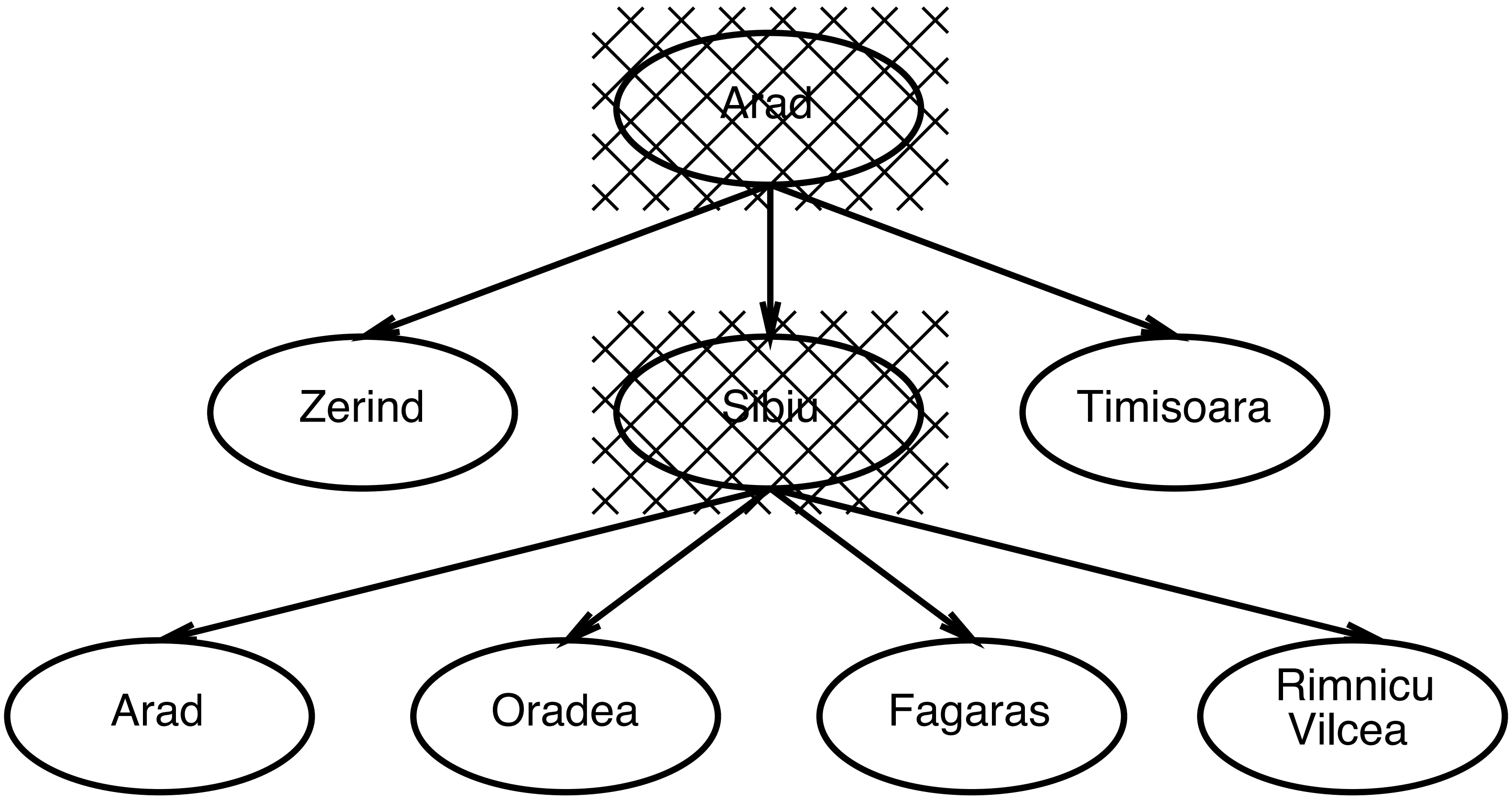
General Search Example



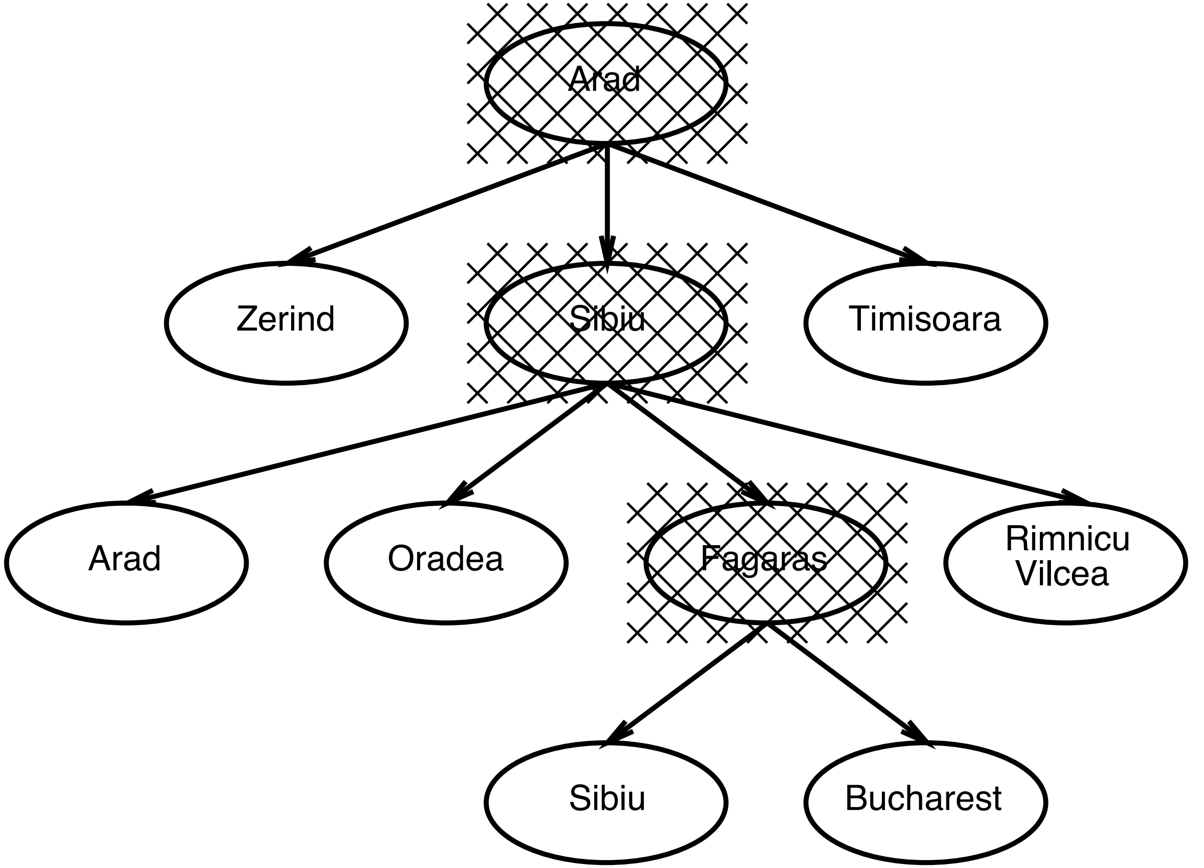
Search Tree (Cont'd)



Search Tree (Cont'd)



Search Tree (Cont'd)



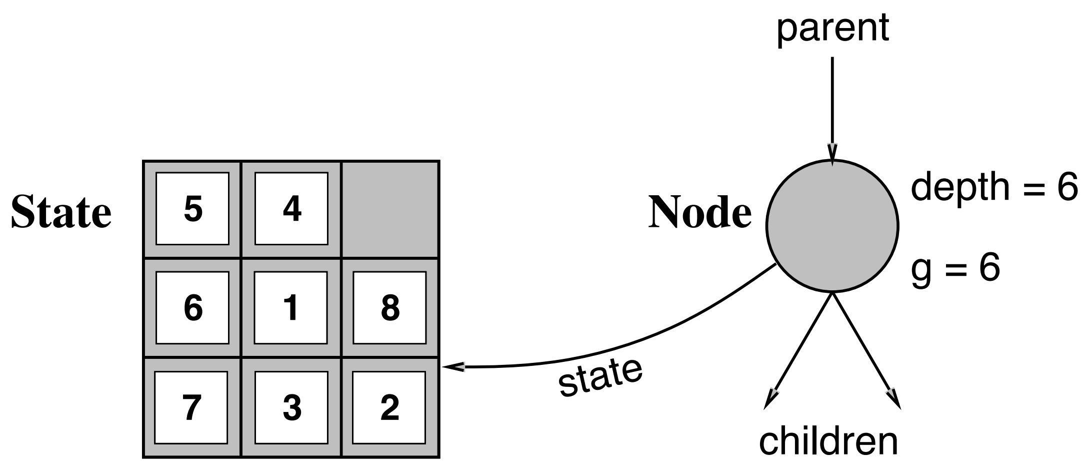
Implementation of Search Algorithms

```
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 (Cont'd): 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

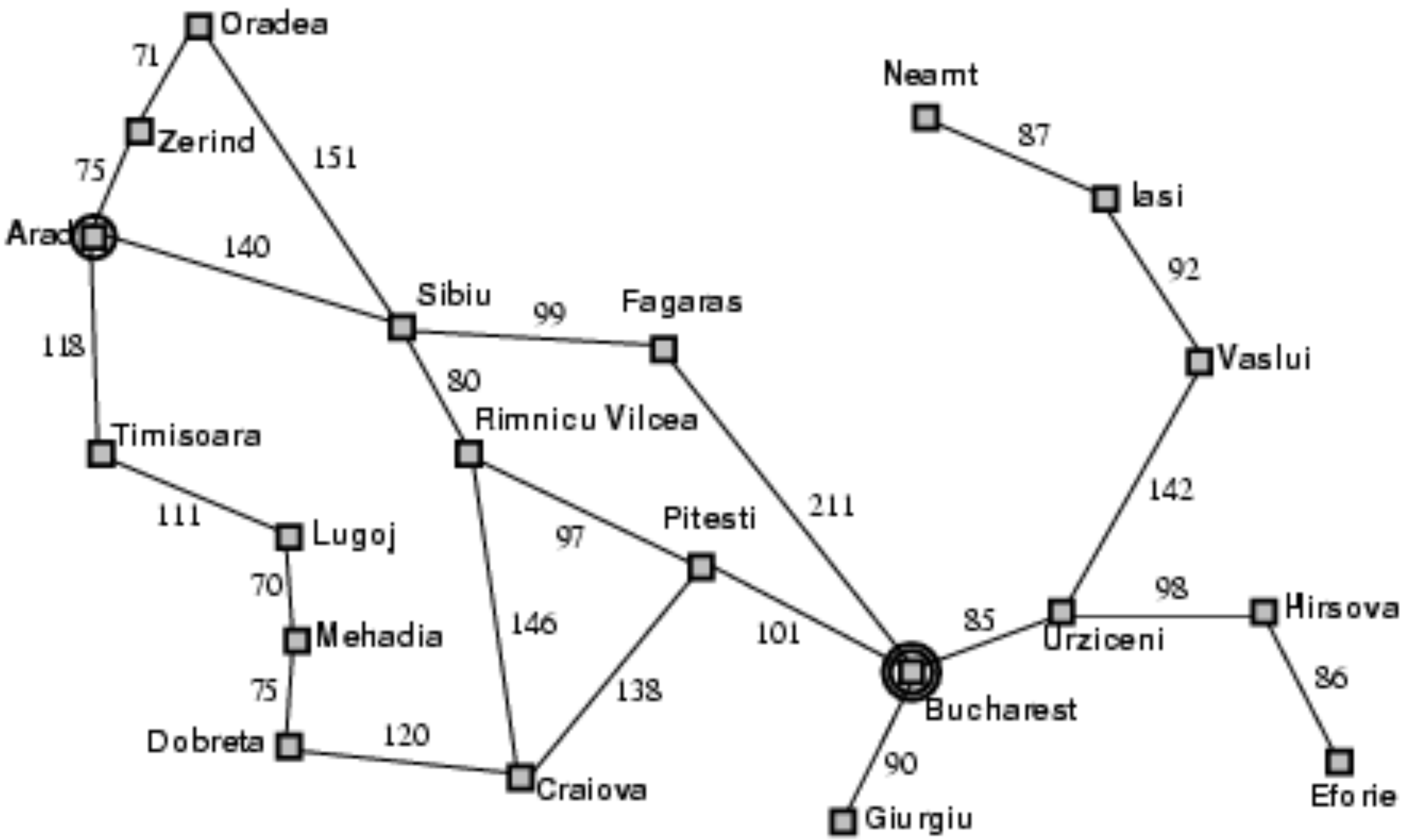
A strategy is defined by picking the *order of node expansion*

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 ∞)

GRAPH SEARCH

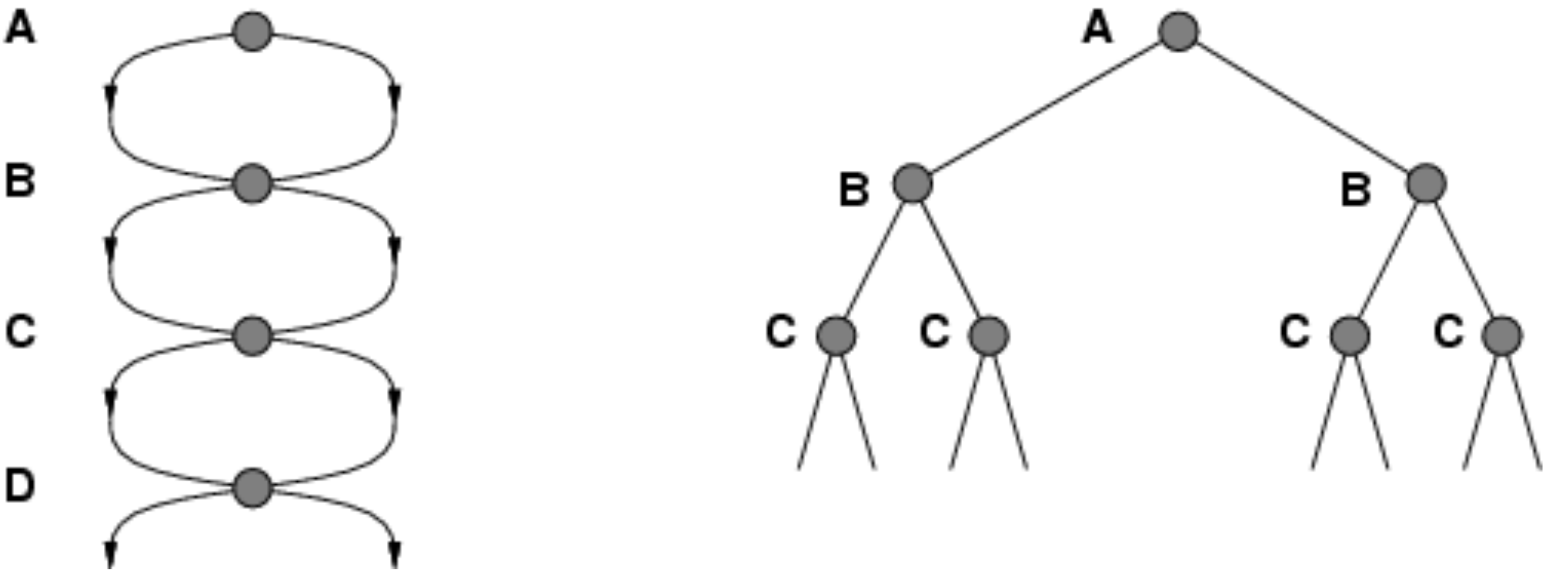
Graph search



Get from Arad to Bucharest as quickly as possible

Graph search

- Failure to detect repeated states can turn a linear problem into an exponential one!



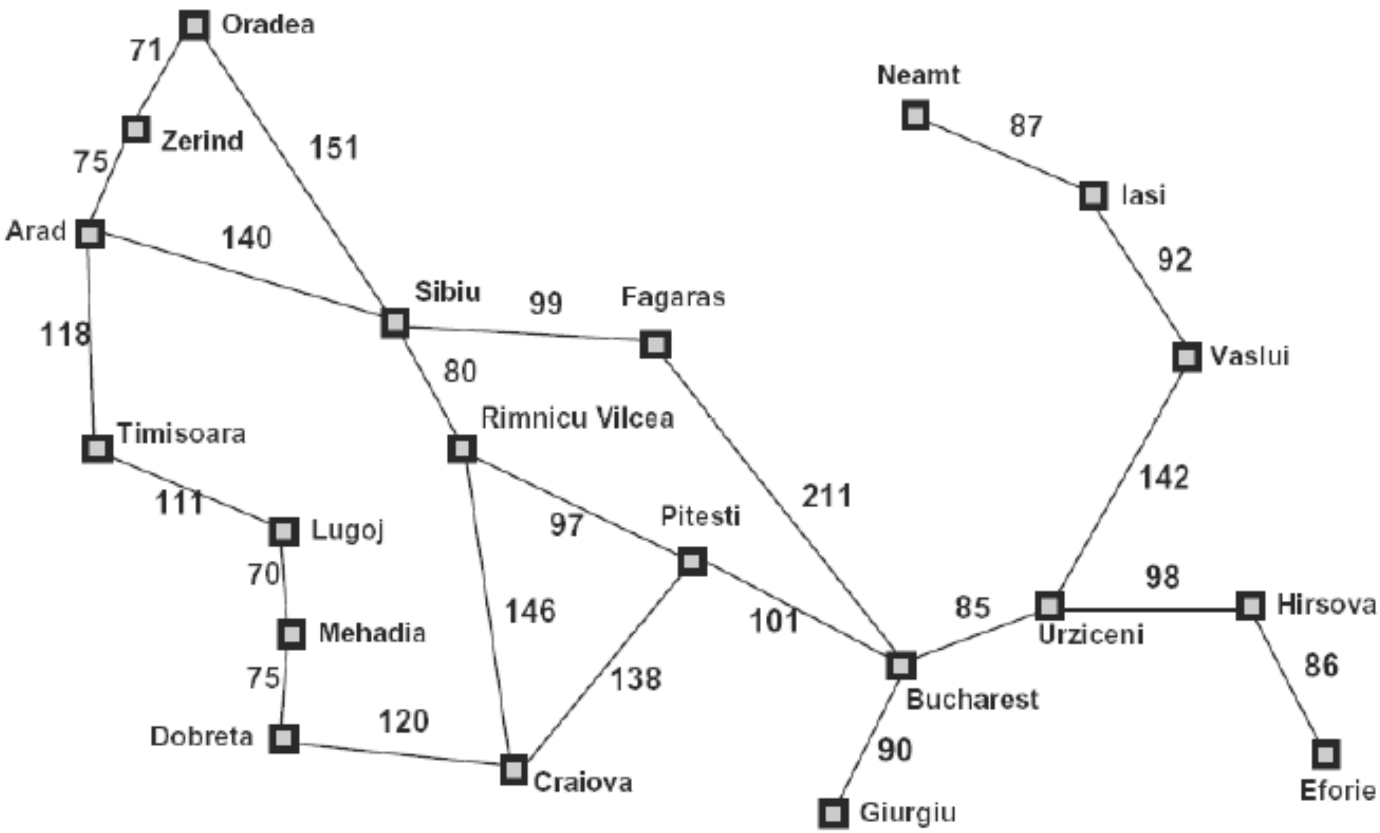
- Very simple fix: never expand a node twice

Graph search

```
function Graph-Search(problem, fringe) returns a solution, or failure
fringe ← Insert(Make-Node(Initial-State(problem)), fringe);
closed ← an empty set
while (fringe not empty)
   node ← RemoveFirst(fringe);
   if (Goal-Test(problem, State(node))) then return Solution(node);
   if (State(node) is not in closed then
      add State(node) to closed
      fringe ← InsertAll(Expand(node, problem), fringe);
   end if
end
return failure;
```

- Never expand a node twice!

Straight Line Distances

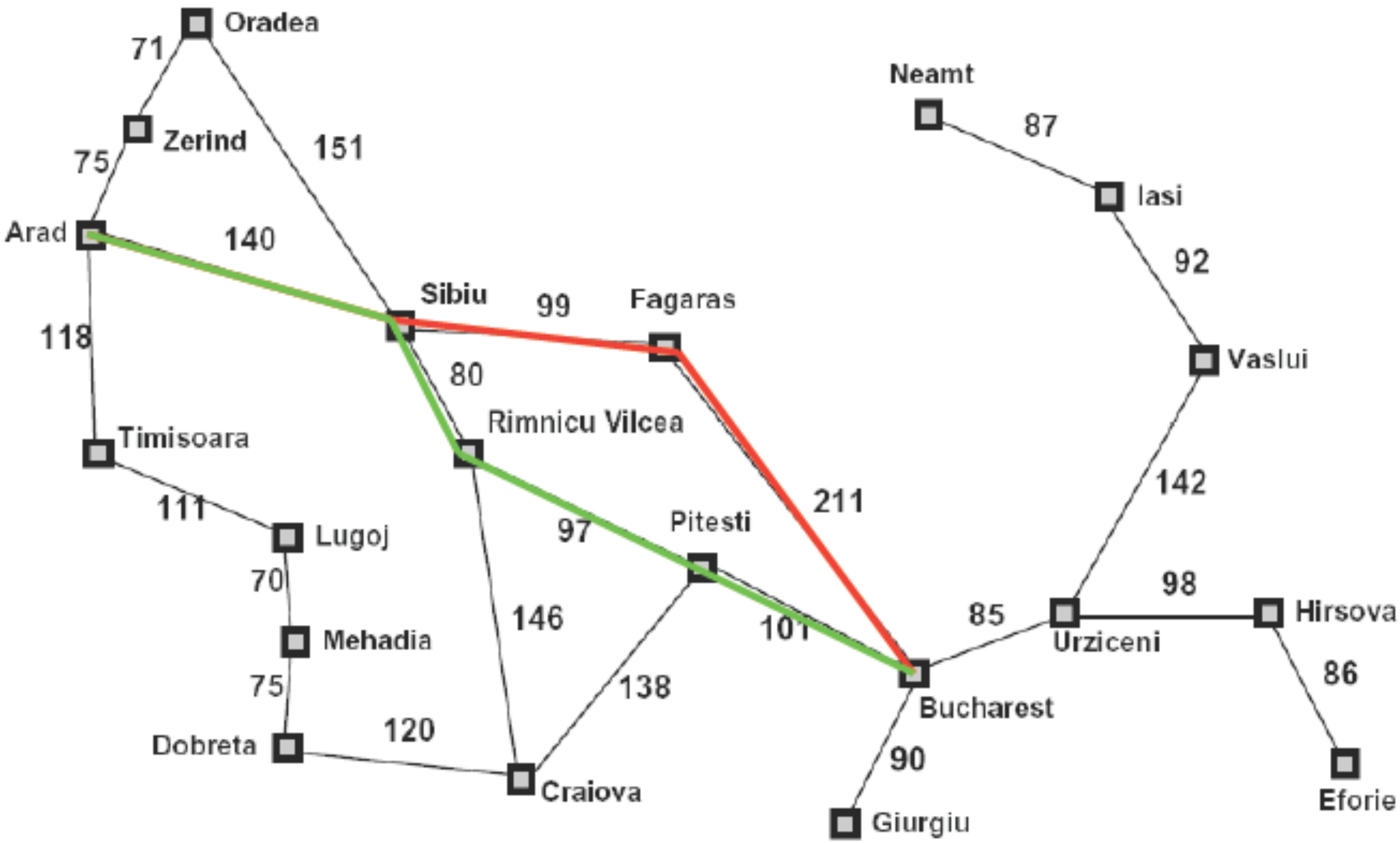


Straight−line distance to Bucharest

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

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"
  - → Expand most desirable unexpanded node
- Order the nodes in fringe in decreasing order of desirability
- Special cases:
  - greedy best-first search
  - A^* search



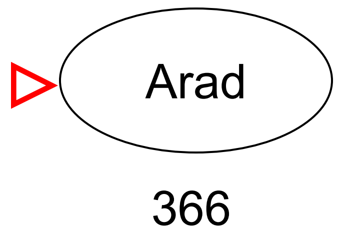
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search

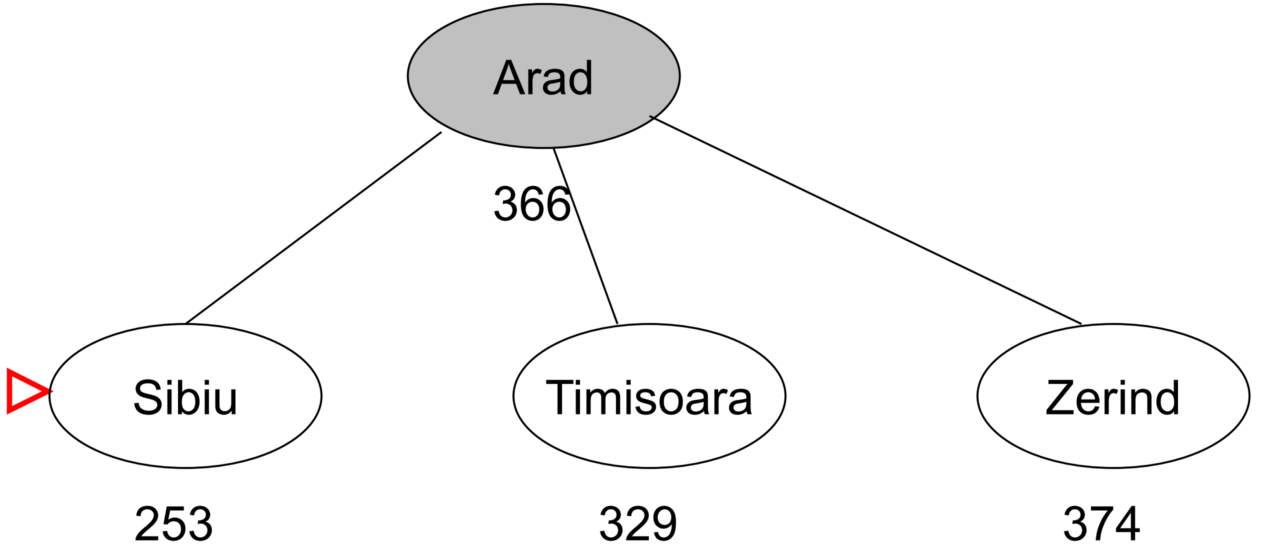
- Evaluation function $f(n) = h(n)$ (heuristic)

  = estimate of cost from n to *goal*
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

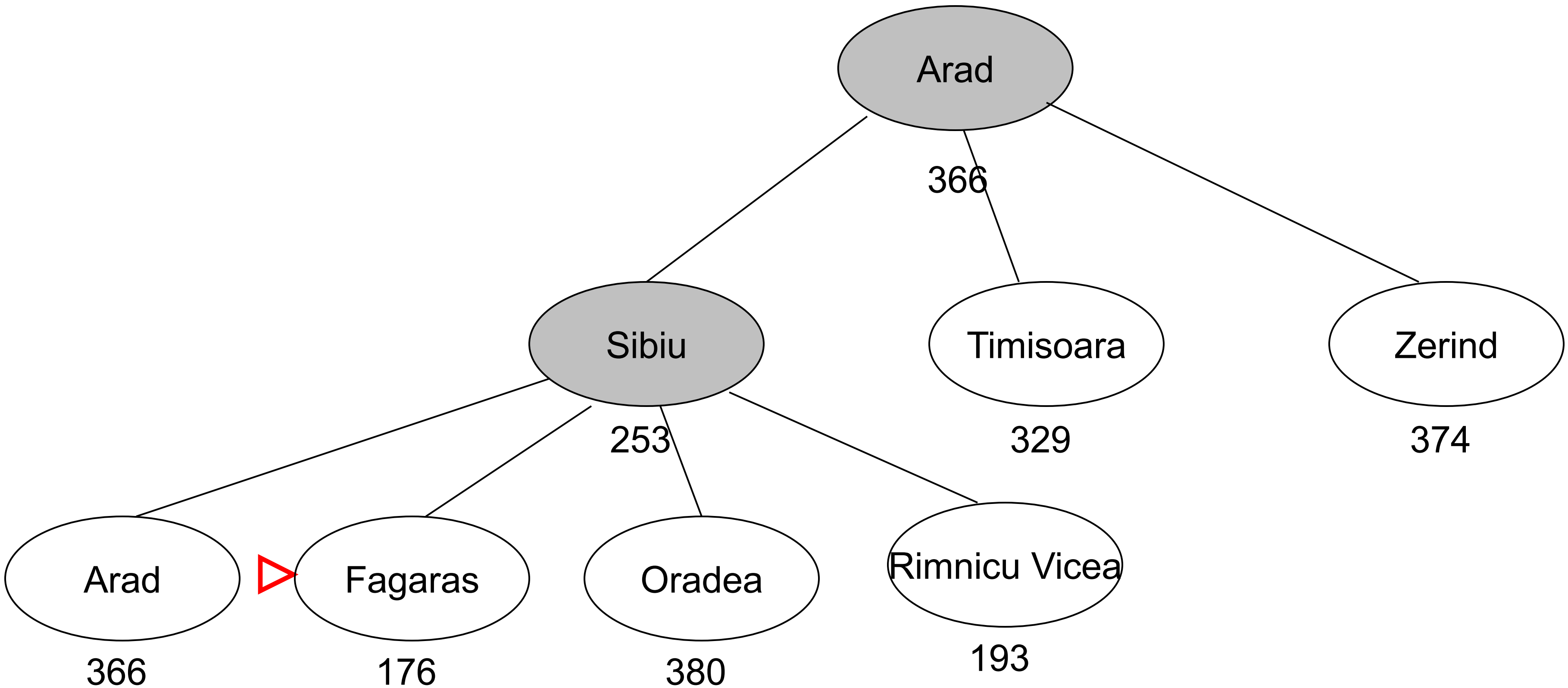
Greedy best-first search example



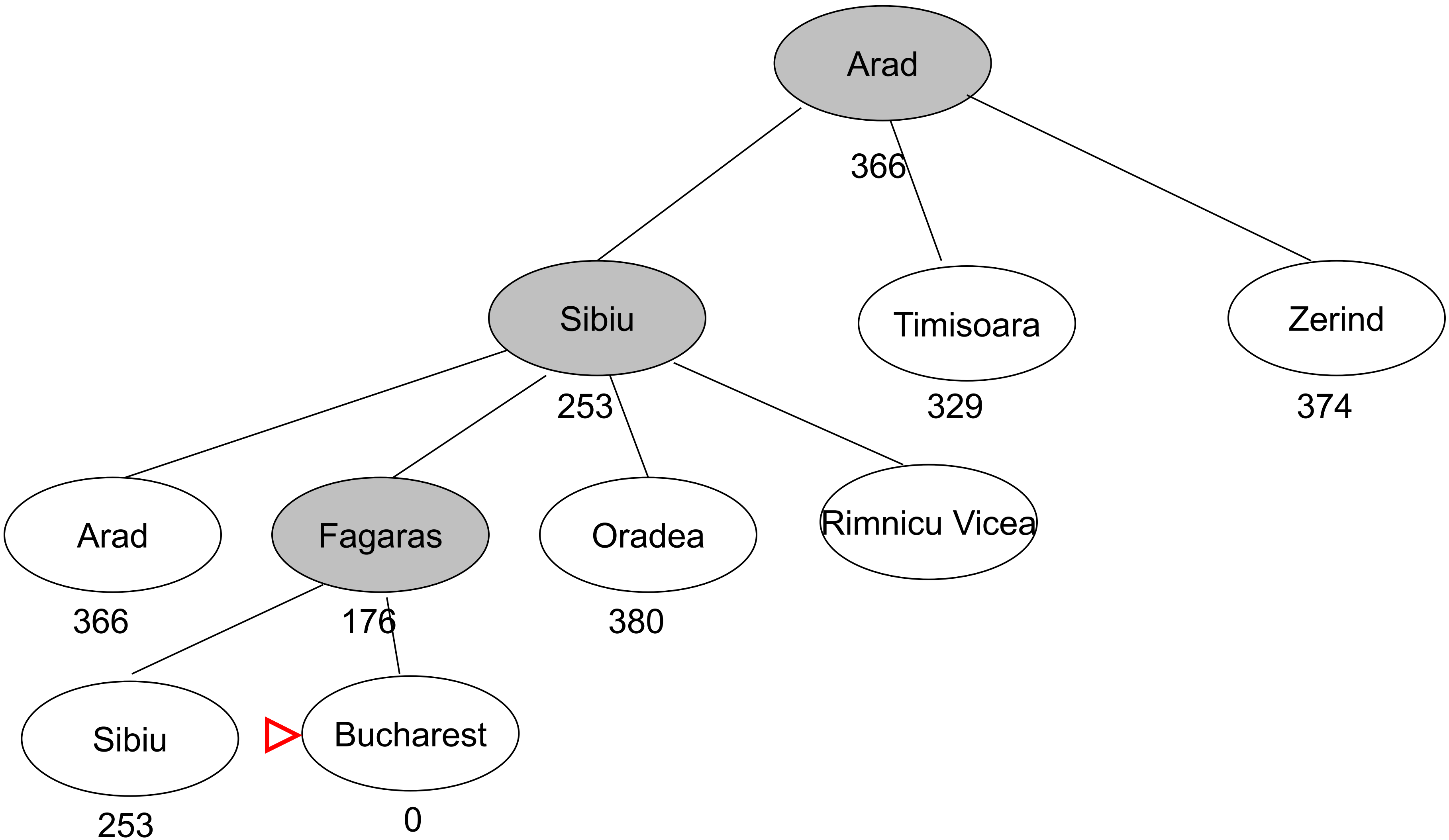
Greedy best-first search example



Greedy best-first search example

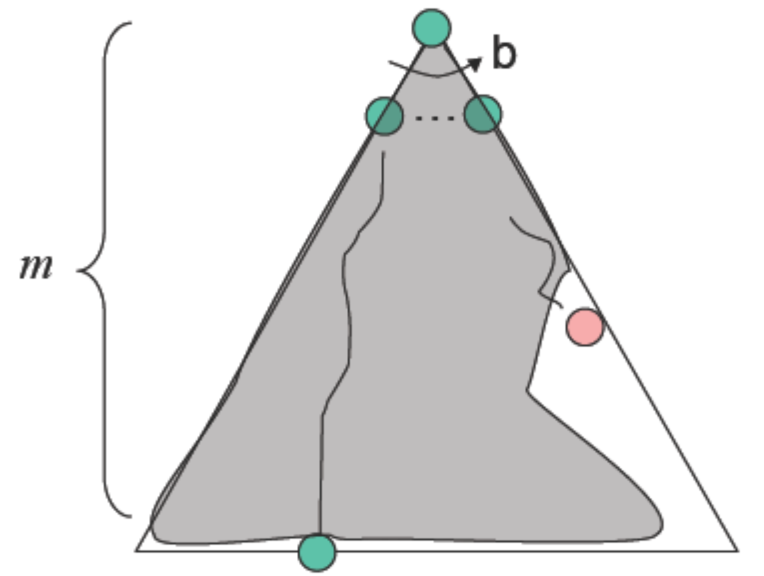


Greedy best-first search example



Greedy Best-First Search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt → …
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No



- What do we need to do to make it complete?

⇒ A* search

- Can we make it optimal? → No

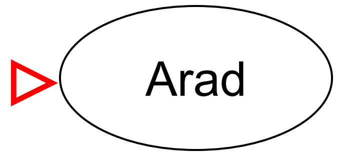
A* search

- Idea: Expand unexpanded node with lowest evaluation value

- Evaluation function ***f(n) = g(n) + h(n)***
- ***g(n)*** = cost so far to reach ***n***
- ***h(n)*** = estimated cost from ***n*** to goal
- ***f(n)*** = estimated total cost of path through ***n*** to goal

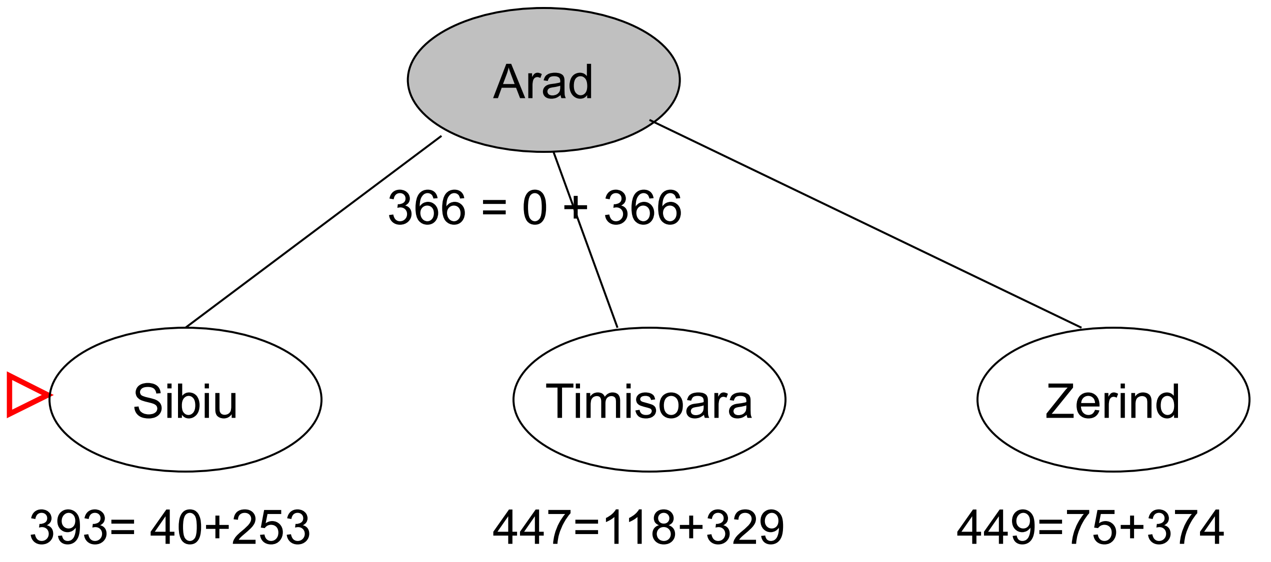
- Nodes are ordered according to ***f(n).***

A* search example

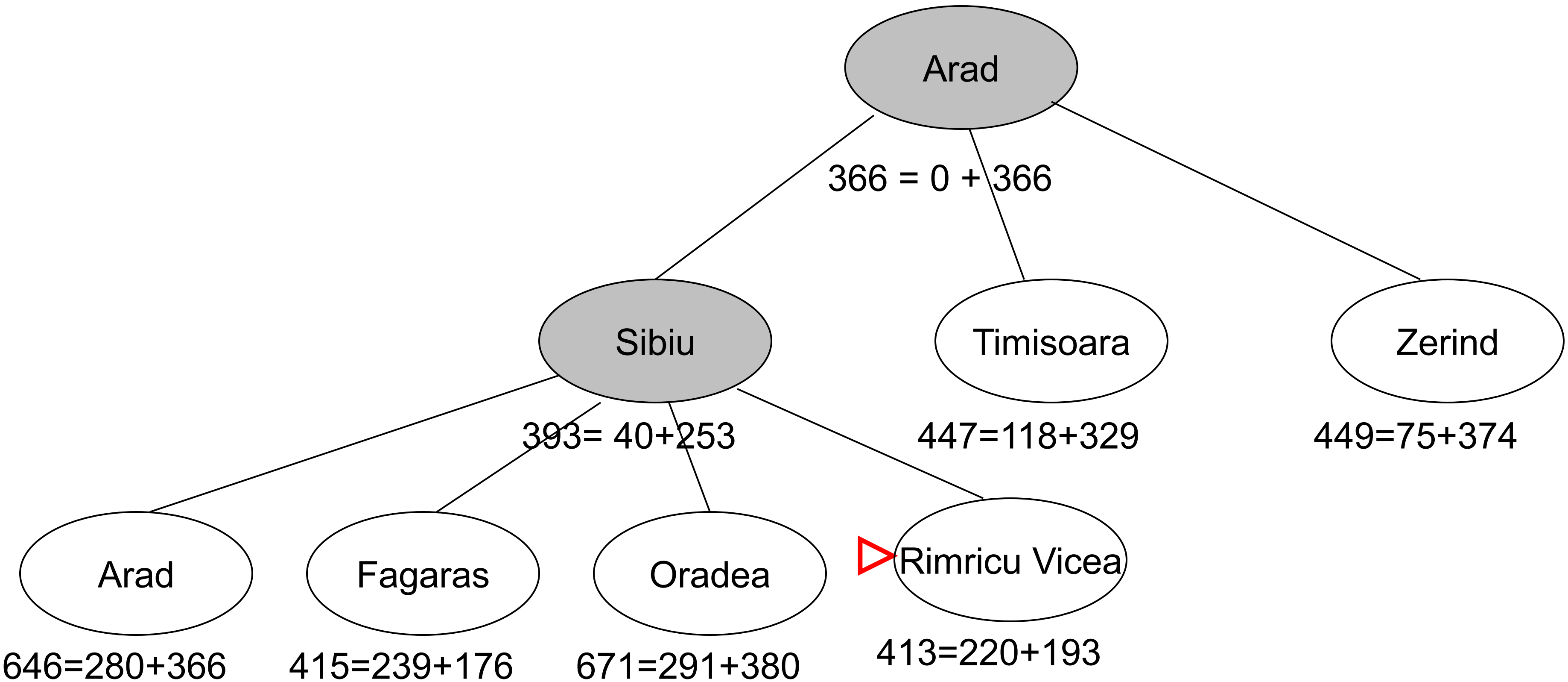


366 = 0 + 366

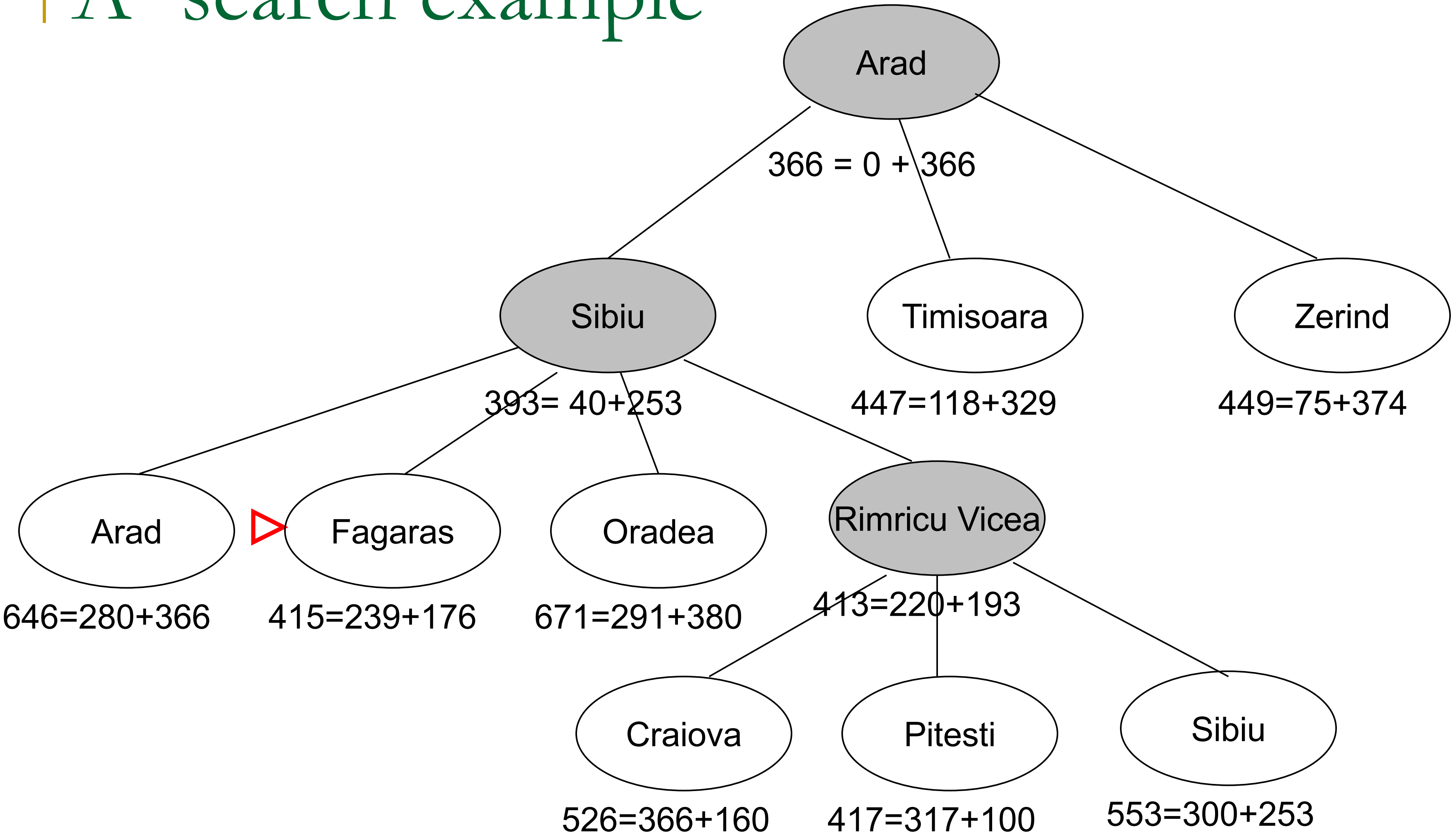
A* search example



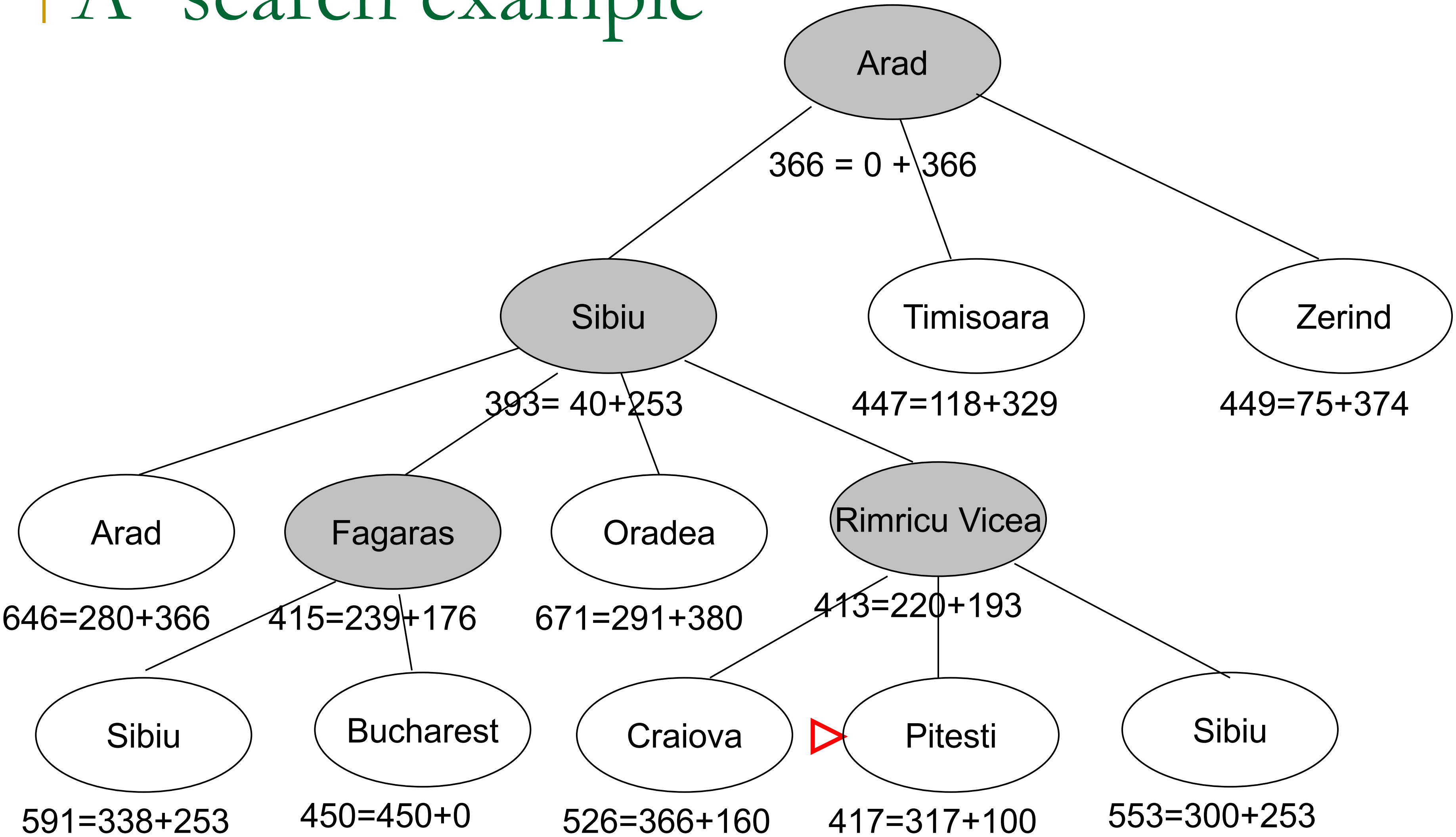
A* search example



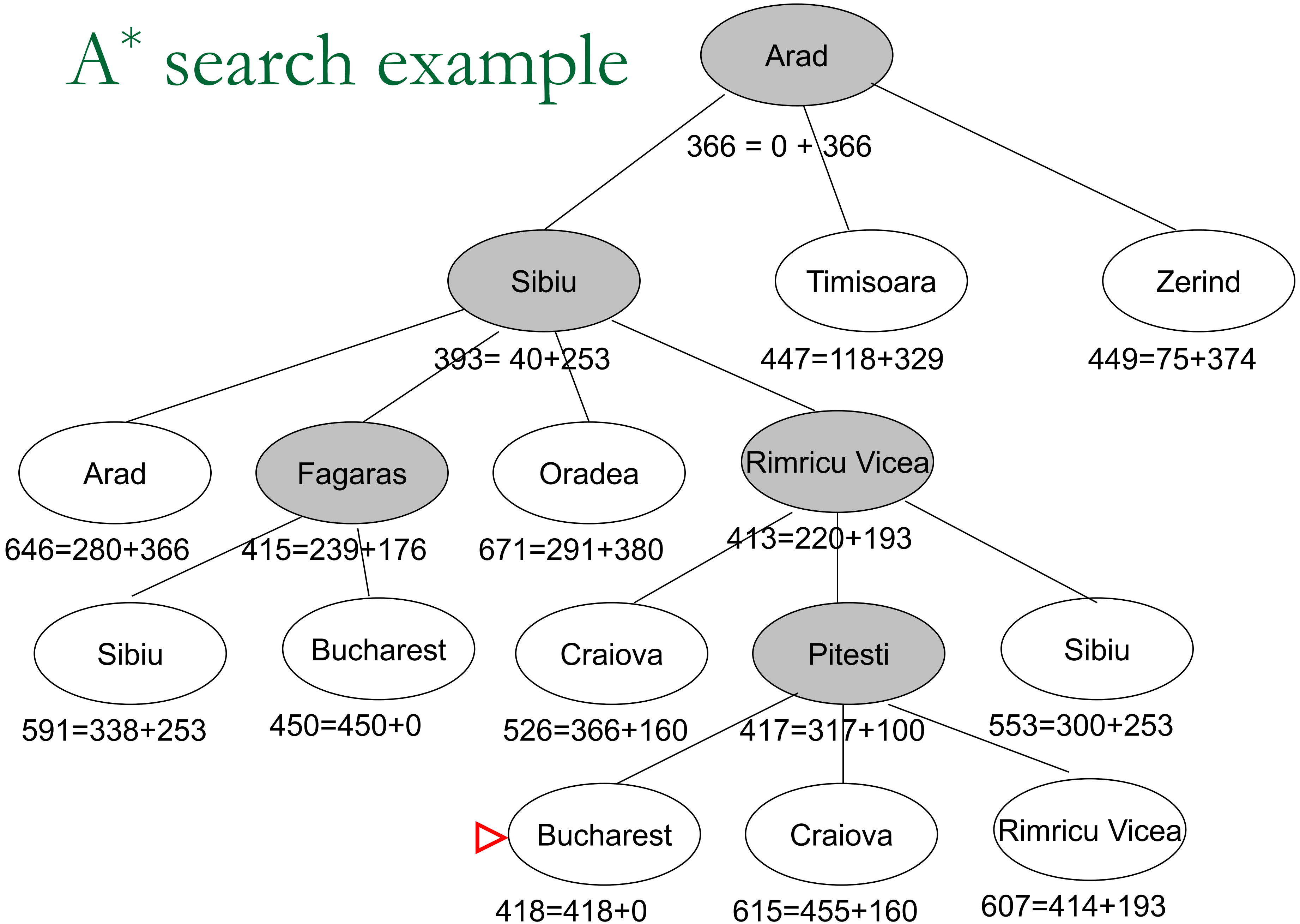
A* search example



A* search example



A* search example



Can we Prove Anything?

- If the state space is finite and we avoid repeated states, the search is **complete,** but in general is **not optimal**
- If the state space is finite and we do not avoid repeated states, the search is in general **not complete**
- If the state space is infinite, the search is in general **not complete**

Admissible heuristic

- Let h*(N) be the **true** cost of the optimal path from N to a goal node
- Heuristic h(N) is **admissible (lower bound)** if:

$$0 \leq h(N) \leq h^*(N)$$

- An **admissible** heuristic is always optimistic

Admissible heuristics

The 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

- <u>$h_1(S)$ = ?</u> 7
- <u>$h_2(S)$ = ?</u> 2+3+3+2+4+2+0+2 = 18

Heuristic quality

- Effective branching factor b*
  - Is the branching factor that a uniform tree of depth *d* would have in order to contain *N+1* nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d$$

  - Measure is fairly constant for sufficiently hard problems.
    - Can thus provide a good guide to the heuristic's overall usefulness.
    - A good value of b* is 1.

Heuristic quality and dominance

- 1200 random problems with solution lengths from 2 to 24
- If $h_1(n) \geq h_2(n)$ for all n (both admissible)
  then h_2 dominates h_1 and is better for search

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Inventing admissible heuristics

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:
  - Relaxed 8-puzzle for h_1 : a tile can move anywhere

    As a result, $h_1(n)$ gives the shortest solution
  - Relaxed 8-puzzle for h_2 : a tile can move to any adjacent square.

    As a result, $h_2(n)$ gives the shortest solution.

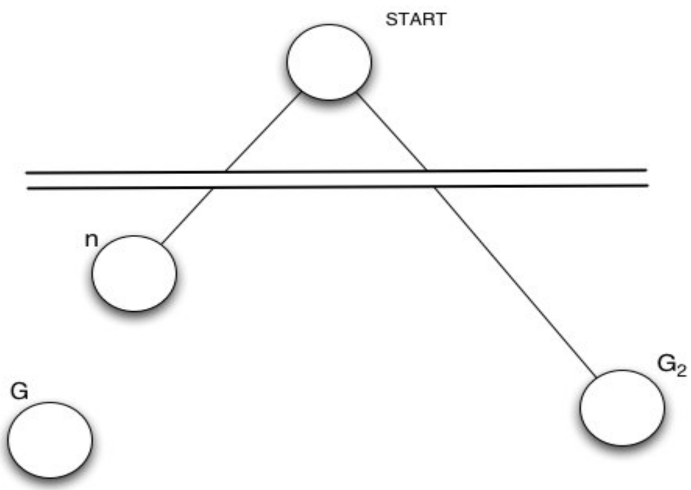
The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.

Optimality of graph-search and of A*

- The tree-search version of A* is optimal if *h(n)* is **admissible**

- The graph-search is optimal if *h(n)* is **consistent**

Optimality of A*(standard proof)



- Suppose suboptimal goal G_2 in the queue.
- Let n be an unexpanded node on a shortest to optimal goal G.

$f(G_2)$	$= g(G_2)$	since $h(G_2)=0$
	$> g(G)$	since G_2 is suboptimal
	$>= f(n)$	since h is admissible

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

Optimality for graphs?

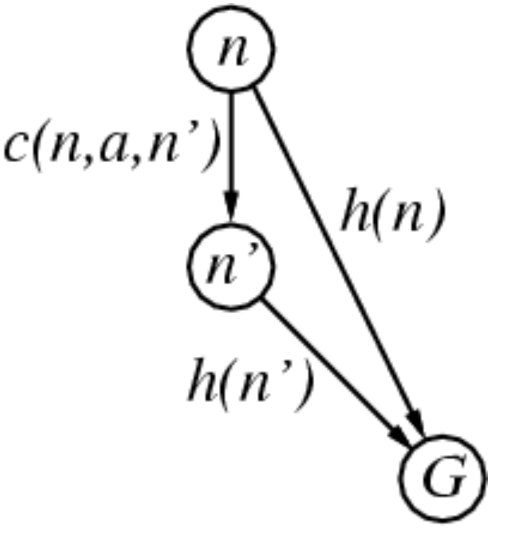
- Admissibility is not sufficient for graph search
  - In graph search, the optimal path to a repeated state could be discarded if it is not the first one generated
  - Can fix problem by requiring consistency property for *h(n)*

- A heuristic is consistent if for every successor *n'* of a node *n* generated by any action *a*,

$h(n) \leq c(n,a,n') + h(n')$

$c(n,a,n')$ = step cost $n \rightarrow n'$ with action *a*

(aka "monotonic")



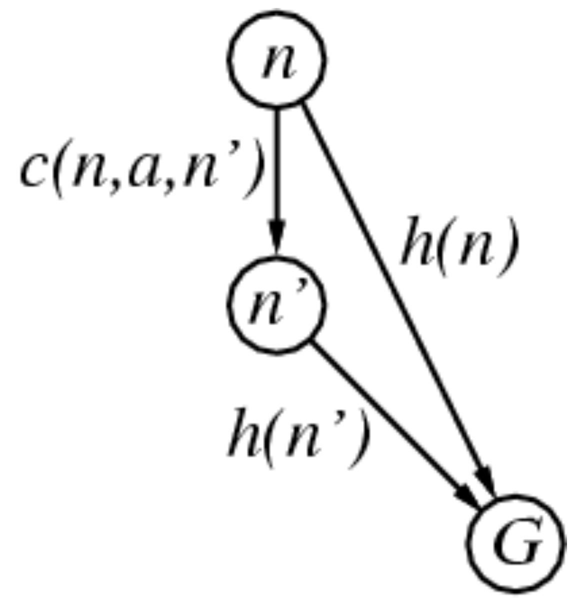
- consistent heuristics are also admissible
- admissible heuristics are not always consistent

A* is optimal with consistent heuristics

- If *h* is consistent, we have

f(n') = g(n') + h(n')
= g(n) + c(n,a,n') + h(n')
≥ g(n) + h(n)
= f(n)

i.e., *f(n)* is non-decreasing along any path.

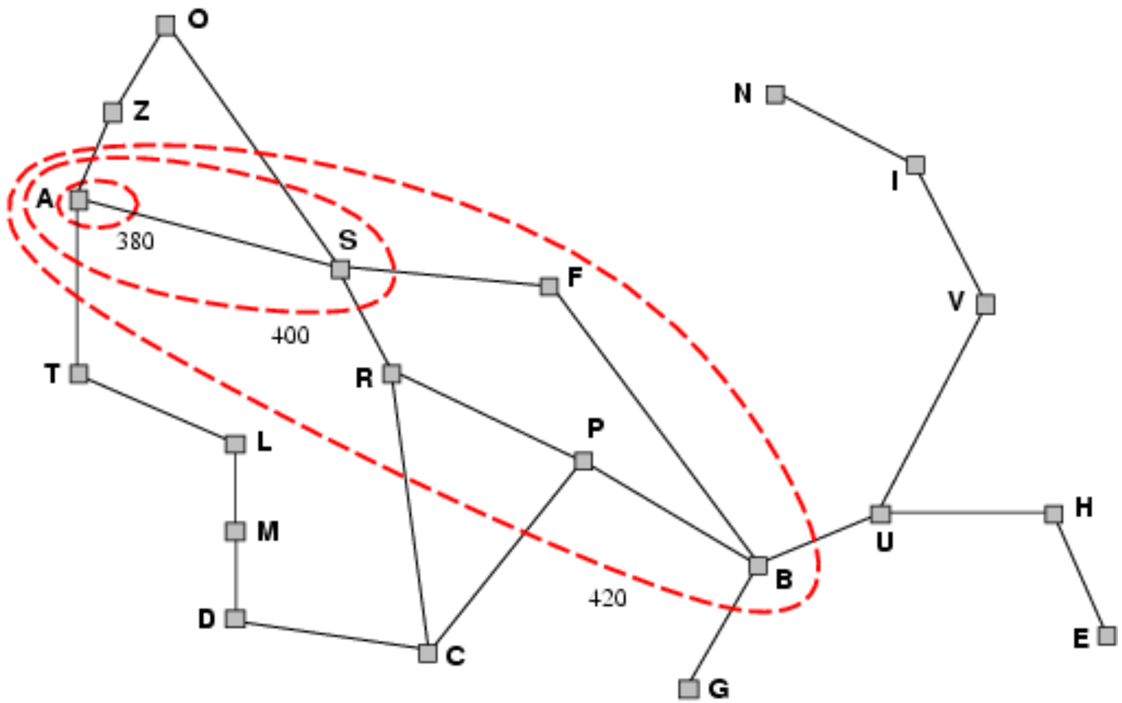


Thus, first goal-state selected for expansion must be optimal

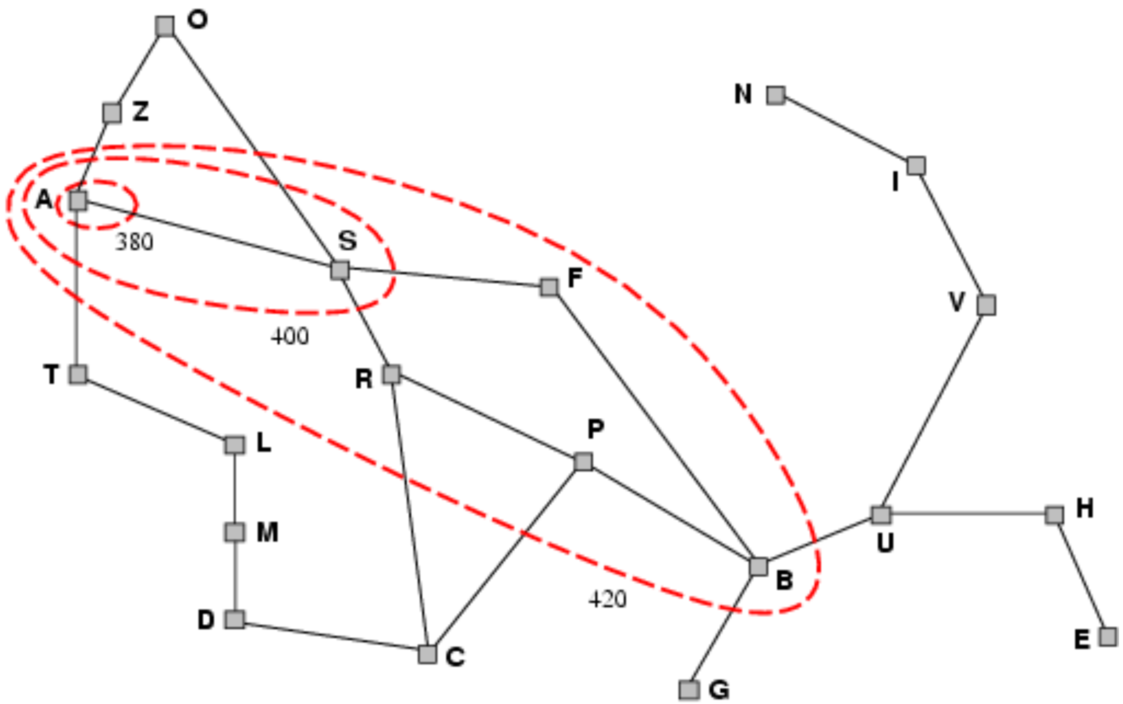
- Theorem:
  - If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal
  -

Contours of A* Search

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



Contours of A* Search



- With uniform-cost (h(n) = 0, contours will be circular
- With good heuristics, contours will be focused around optimal path
- A* will expand all nodes with cost f(n) < C*

A* search, evaluation

- Completeness: YES
  - Since bands of increasing *f* are added
  - Unless there are infinitely many nodes with $f<f(G)$

A* search, evaluation

- Completeness: YES
- Time complexity:
  - Number of nodes expanded is still exponential in the length of the solution.

A* search, evaluation

- Completeness: YES
- Time complexity: (exponential with path length)
- Space complexity:
  - It keeps all generated nodes in memory
  - Hence space is the major problem not time

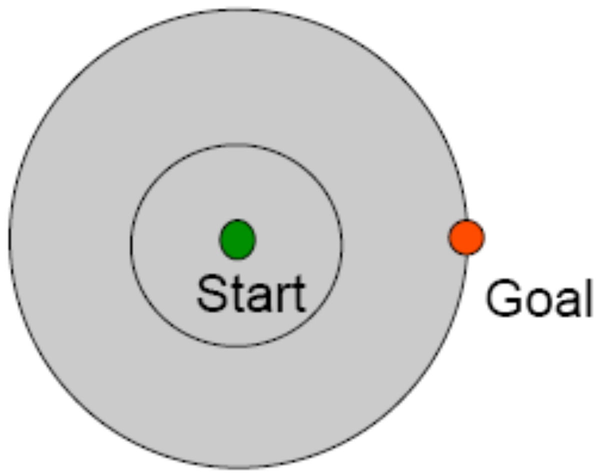
A* search, evaluation

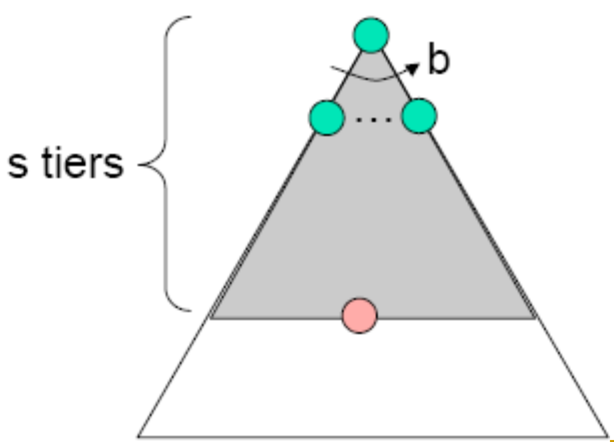
- Completeness: YES
- Time complexity: (exponential with path length)
- Space complexity:(all nodes are stored)
- Optimality: YES
  - Cannot expand f_{i+1} until f_i is finished.
  - A* expands all nodes with $f(n)< C^*$
  - A* expands some nodes with $f(n)=C^*$
  - A* expands no nodes with $f(n)>C^*$

Also *optimally efficient* (not including ties)

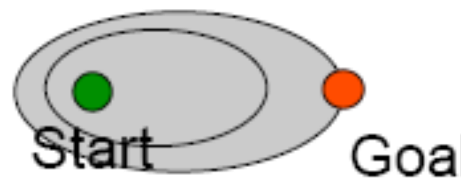
Compare Uniform Cost and A*

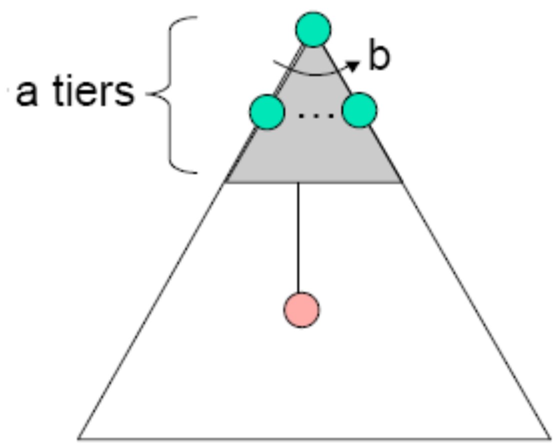
- Uniform-cost expanded in all directions





- A* expands mainly toward the goal, but does hedge its bets to ensure optimality





Reading and Suggested Exercises

- Chapter 3
- Exercises 3.2, 3.7, 3.9, 3.26

Exercise 7

- Consider the unbounded regular 2D grid state space shown below. The start state is the origin (marked) and the goal state is at *(x,y)*.

1. What is the branch factor b in this state space?
2. How many distinct states are there at depth k (for (k > 0)?
   - (i) 4^k (ii) 4k (iii) $4k^2$
3. Breadth-first search without repeated-state checking expand at most
   - (i) $((4^{x+y+1} – 1) /3) -1$ (ii) $4(x+y) - 1$ (iii) $2(x+y)(x+y+1) – 1$

     nodes before terminating
4. Breadth-first search with repeated-state checking expand up to
   - (i) $((4^{x+y+1} – 1) /3) -1$ (ii) $4(x+y) - 1$ (iii) $2(x+y)(x+y+1) – 1$

     nodes before terminating
5. Is $h = |u - x| + |v - y|$ an admissible heuristic for a state at *(u, v)*?
6. A^* search with repeated-state checking using *h* expands *O(x+y)* nodes before terminating: True or false?
7. *h* remains admissible if some links are removed: True or False?
8. *h* remains admissible if some links are added between nonadjacent states: True or False?

Exercise 8

Consider the problem of moving k knights from k starting squares s_1, s_2, …, s_k to k goal squares g_1,…,g_k, on an unbounded chessboard, subject to the rule that no two knights can land on the same square at the same time. Each action consists of moving up to k knights simultaneously. We would like to complete the maneuver in the smallest number of actions.

1. What is the maximum branching factor b in this state space?
   - **(i) 8k, (ii) 9k, (iii) 8^k, (iv) 9^k**
2. Suppose h_i is an admissible heuristic for the problem of moving knight i to goal g_i by itself. Which of the following heuristics are admissible for the k-knight problem?
   - **(i) min{h_1,…,h_k}, (ii) max{h_1,…,h_k}, (iii) $\sum_{i=1}^{k} h_i$**

Exercise 9

- Suppose there are two friends living in different cities on a map. On every turn, we can move each friend simultaneously to a neighboring city on the map. The amount of time needed to move from city i to neighbor j is equal to the road distance d(i, j) between the cities, but on each turn the friend that arrives first must wait until the other one arrives (and calls the first on his/her cell phone) before the next turn can begin. We want the two friends to meet as quickly as possible. Let us formulate this as a search problem.
  1. What is the state space? (You will find it helpful to define some formal notation here.)
  2. What is the successor function?
  3. What is the goal?
  4. What is the step cost function?
  5. Let SLD(i, j) be the straight-line distance between any two cities i and j. Which, if any, of the following heuristic functions are admissible? (If none, write NONE.) (i) SLD(i, j) (ii) 2 · SLD(i, j) (iii) SLD(i, j)/2
  6. True/False: There are completely connected maps for which no solution exists