

# ARTIFICIAL INTELLIGENCE

## SEARCH WEEKS 11 AND 12

# Artificial Intelligence, Weeks 11 and 12

Based on "Artificial Intelligence — A Modern Approach" by Stuart Russel and Peter Norvig, Chapter 3

See also: "Artificial Intelligence in the 21st Century" by Steven Lucci and Danny Kopec, Chapters 2 and 3

Slides edited by Benjamin Inden based on slides by Stuart Russel,

<http://aima.cs.berkeley.edu>

# Outline Part 1

- ◇ What is a search problem?
- ◇ Example search problems
- ◇ Basic search algorithms

## Problem Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

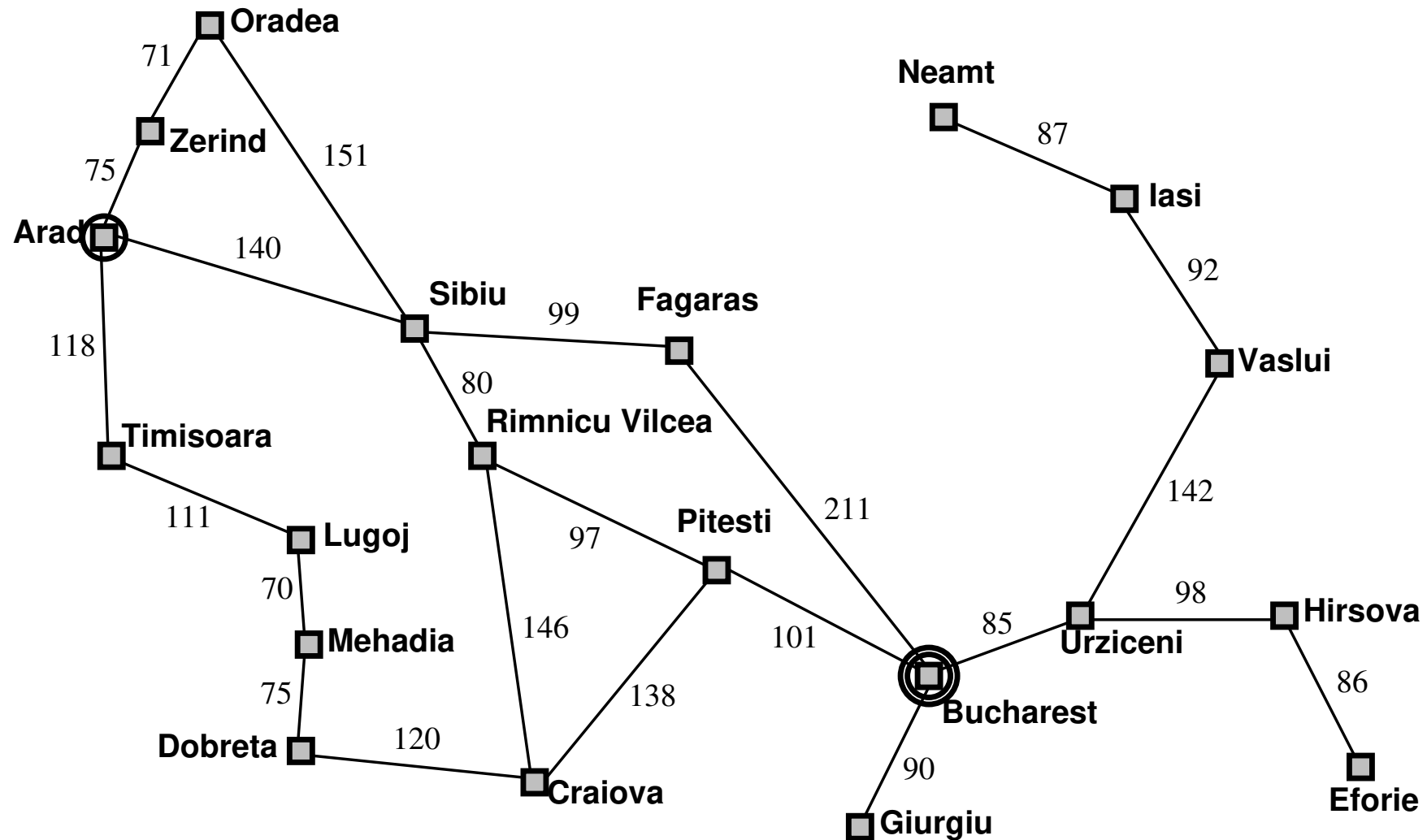
states: various cities

actions: drive between cities

Find solution:

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

## Example: Romania



## Single-state problem formulation

A **problem** is defined by four items:

**initial state** e.g., “at Arad”

**successor function**  $S(x)$  = set of action–state pairs

e.g.,  $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \dots\}$

**goal test**, can be

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

**implicit**, e.g.,  $NoDirt(x)$

**path cost** (additive)

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

$c(x, a, y)$  is the **step cost**, assumed to be  $\geq 0$

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

## 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.

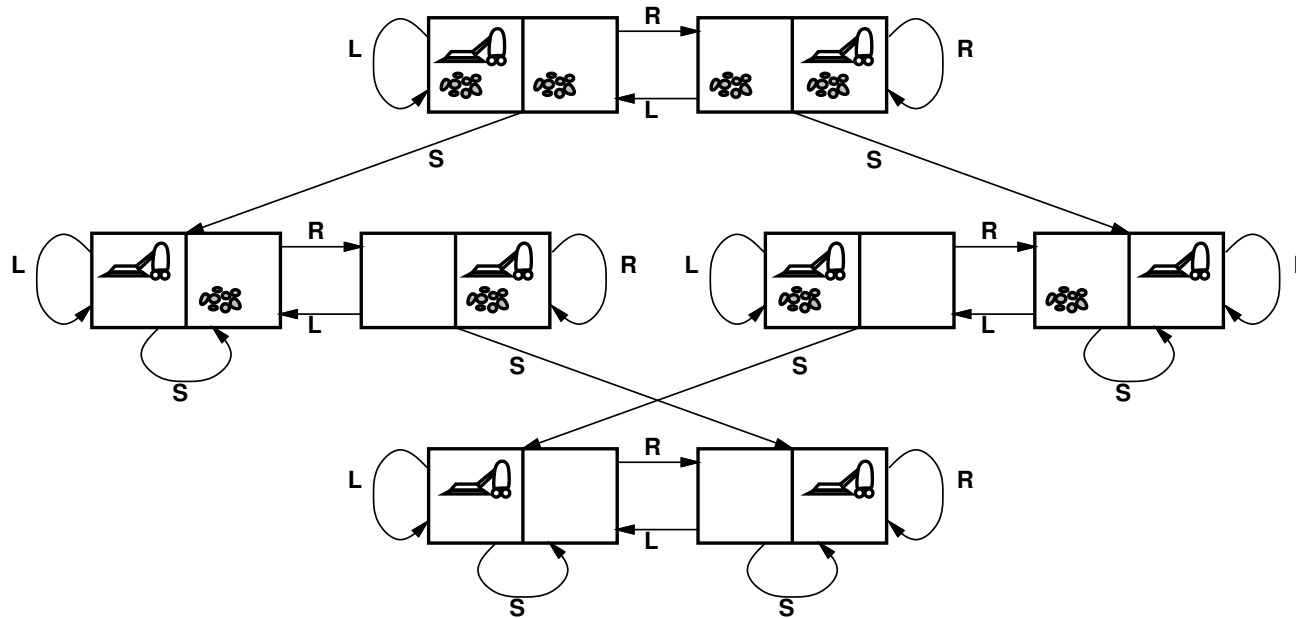
For guaranteed realizability, **any** real state “in Arad”  
must get to **some** real state “in Zerind”

(Abstract) solution =

set of real paths that are solutions in the real world

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

## Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)



## Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

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

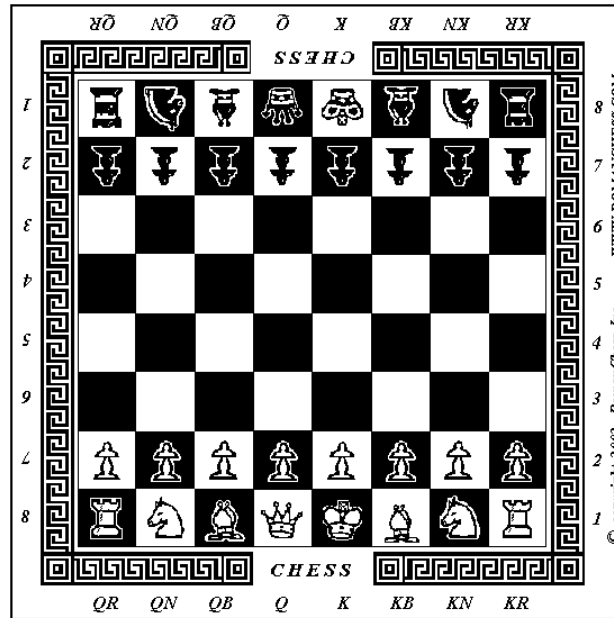
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

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

# Example: Chess



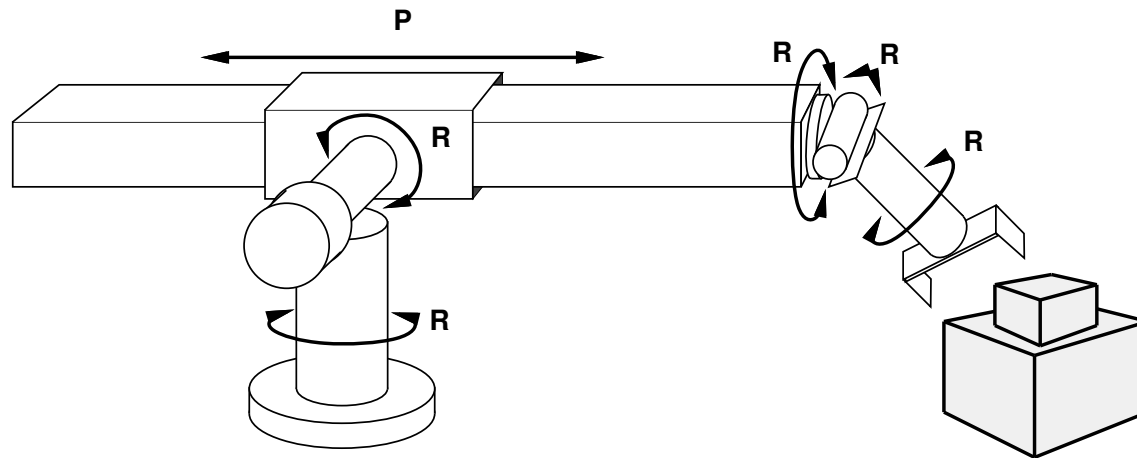
states??: locations of all pieces

actions??: move a piece

goal test??: = checkmate

path cost??: 1 per move

## 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

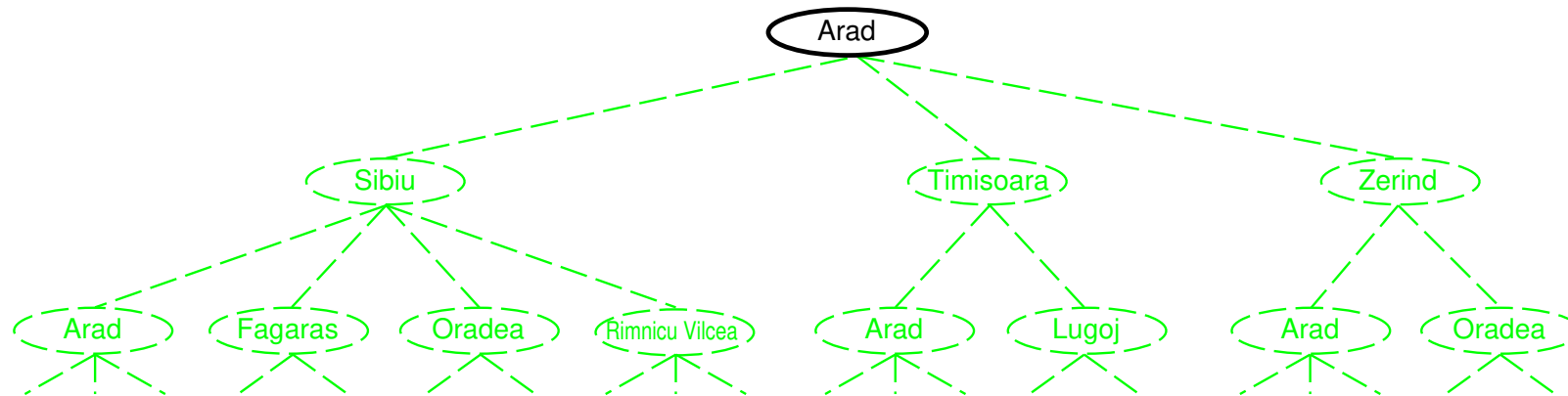
# Tree search algorithms

Basic idea:

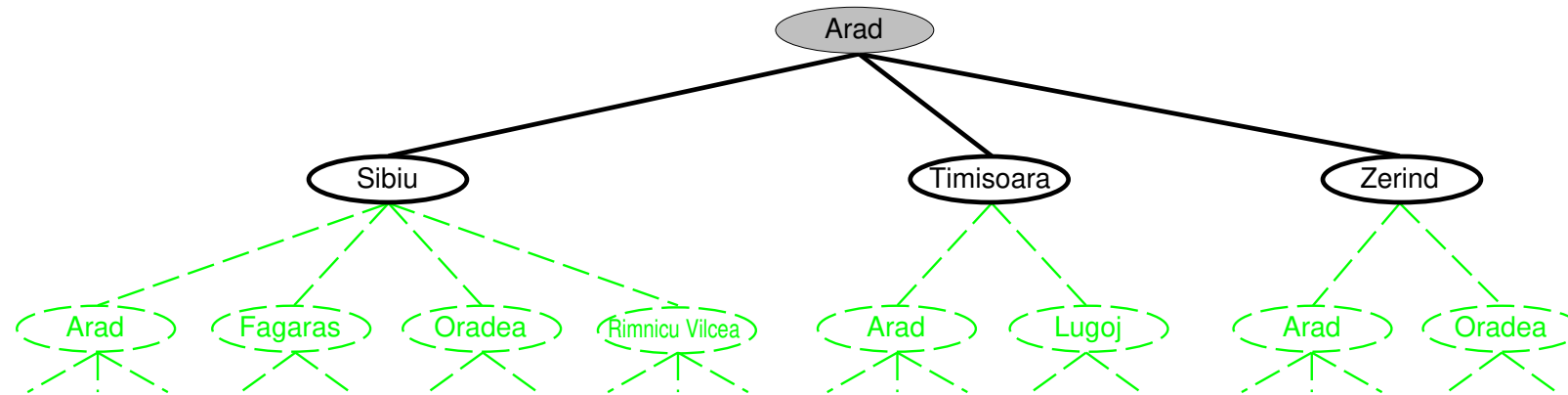
offline, simulated exploration of state space  
by generating successors of already-explored states  
(a.k.a. **expanding** states)

```
function TREE-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
```

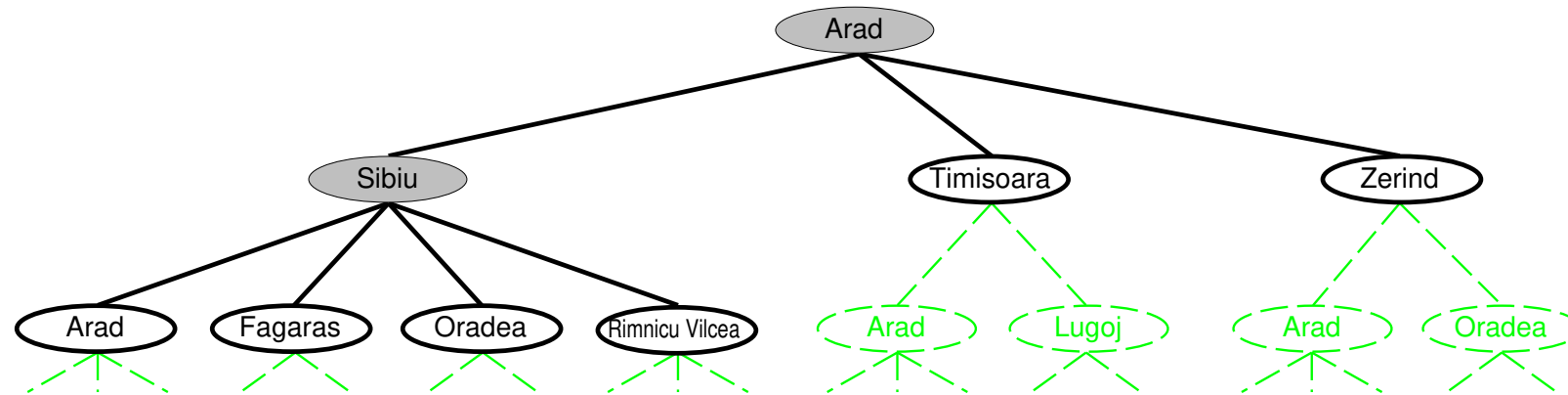
# Tree search example



# Tree search example



# Tree search example



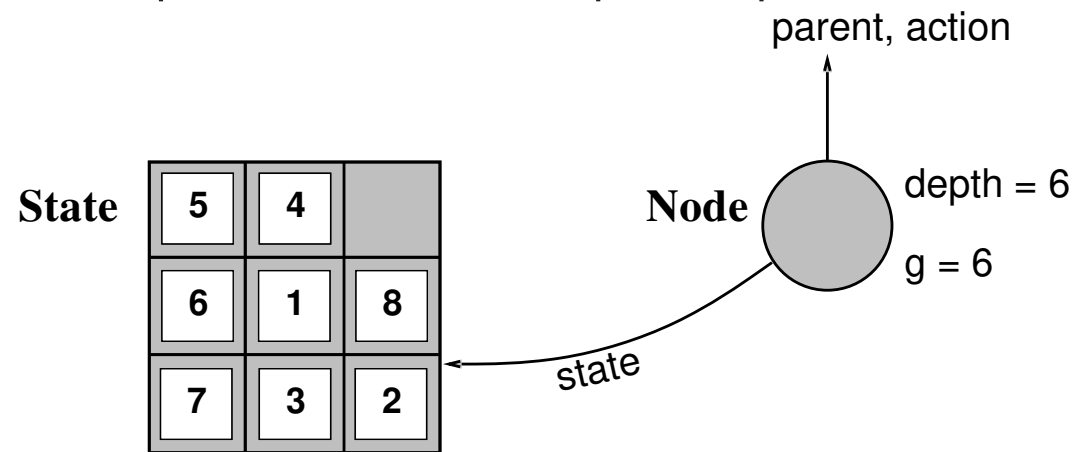
## Implementation: states vs. nodes

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 SUCCESSORFN of the problem to create the corresponding states.



## Implementation: general tree search

```

function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)

```

---

```

function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action,
result)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors

```

## 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

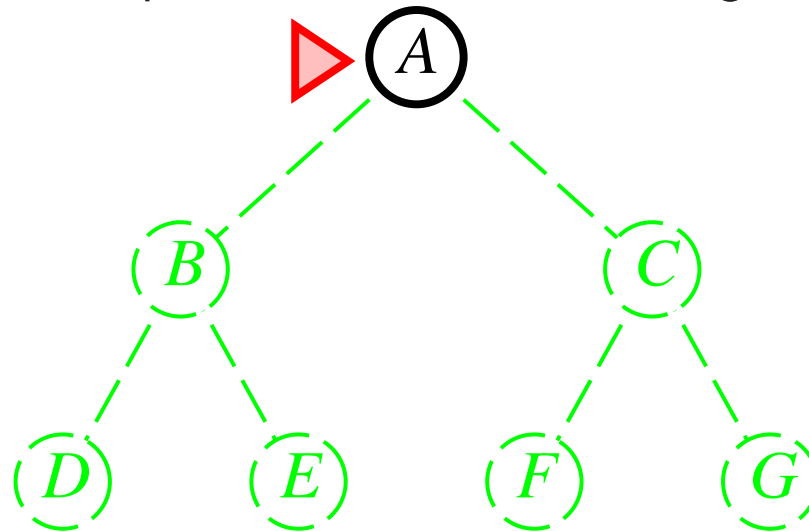
Iterative deepening search

# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end

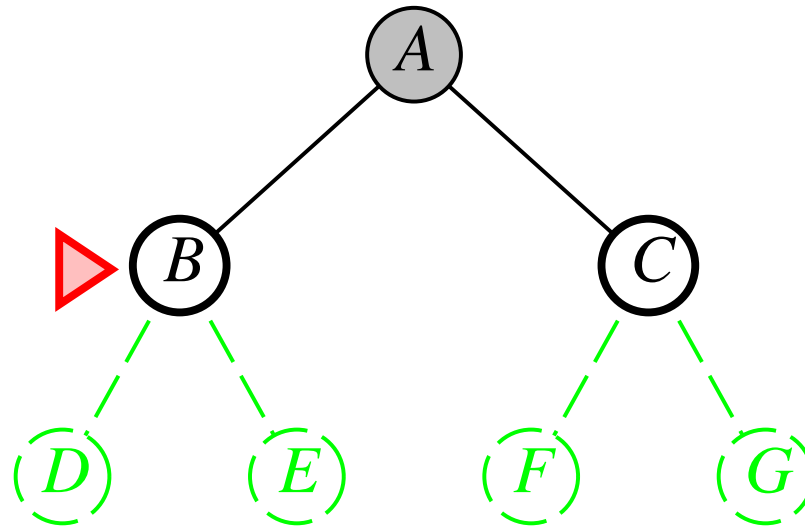


# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end

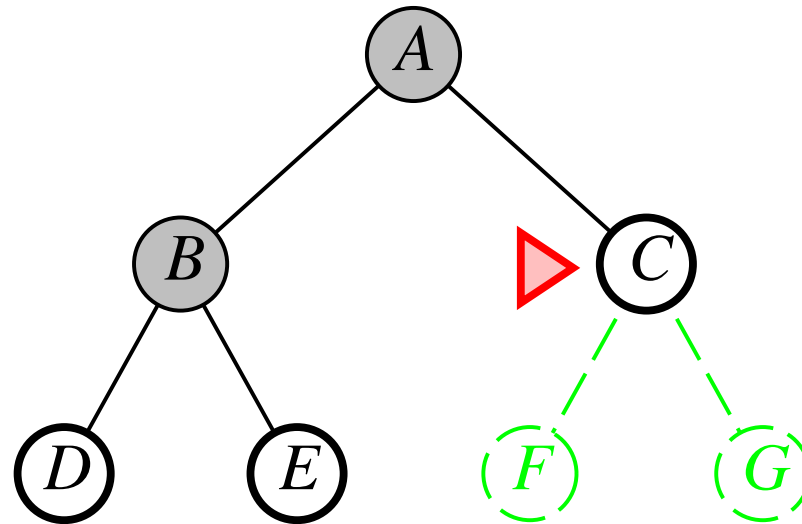


# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end

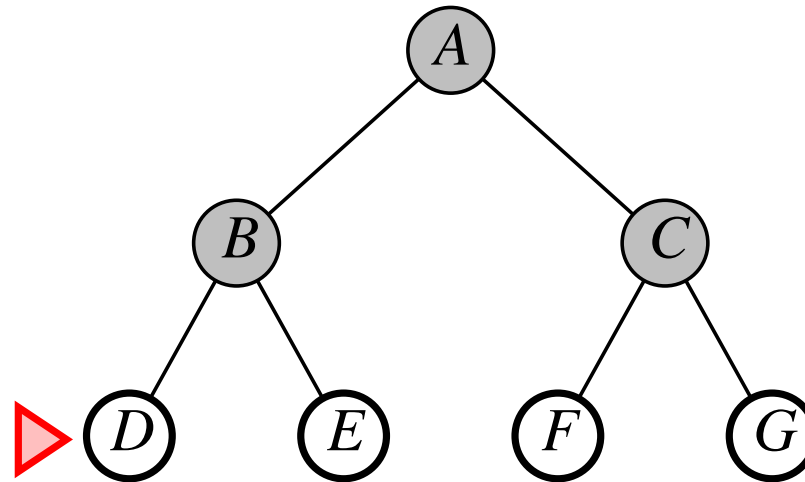


# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end





## Properties of breadth-first search

Complete?? Yes (if  $b$  is finite)

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

Space??  $O(b^{d+1})$  (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 100MB/sec  
so 24hrs = 8640GB.

## Uniform-cost search

Expand least-cost unexpanded node

### Implementation:

*fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

Time?? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

Space?? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$

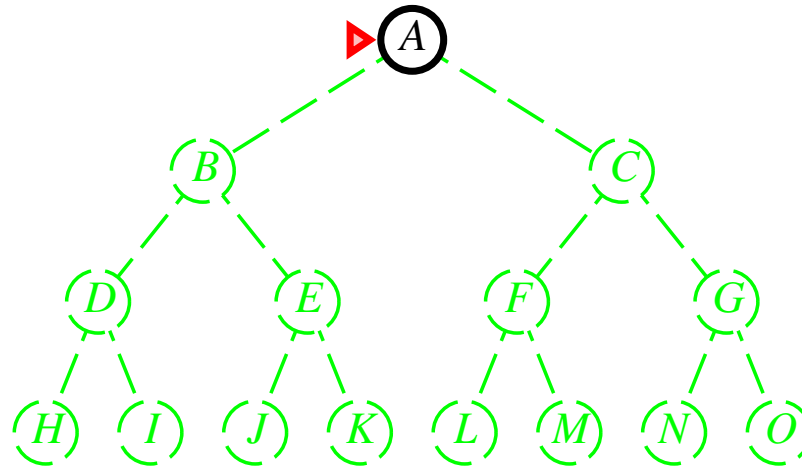
Optimal?? Yes—nodes expanded in increasing order of  $g(n)$

# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

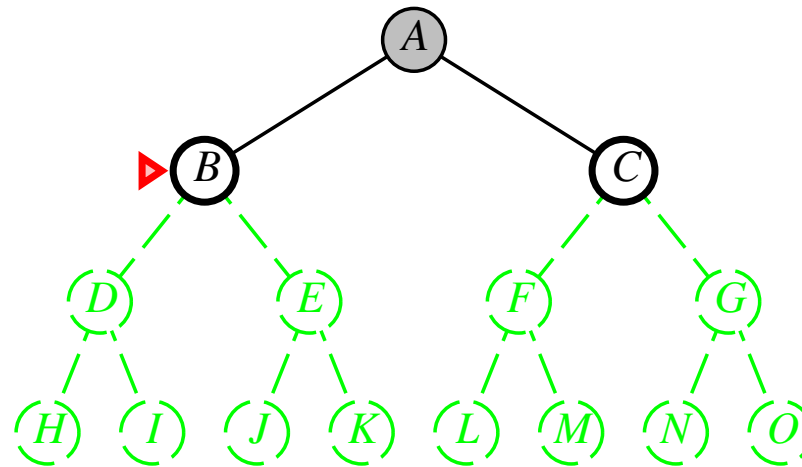


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

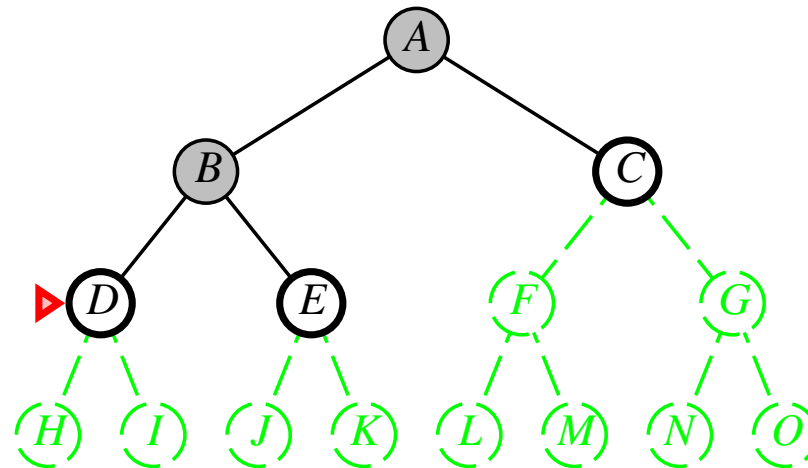


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front

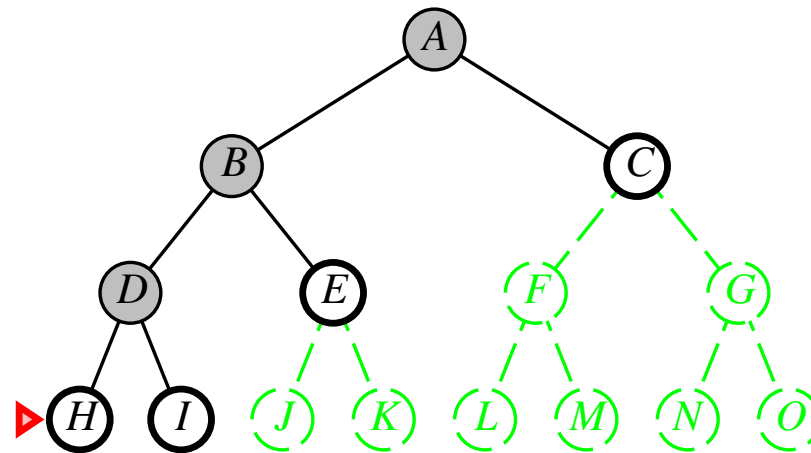


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

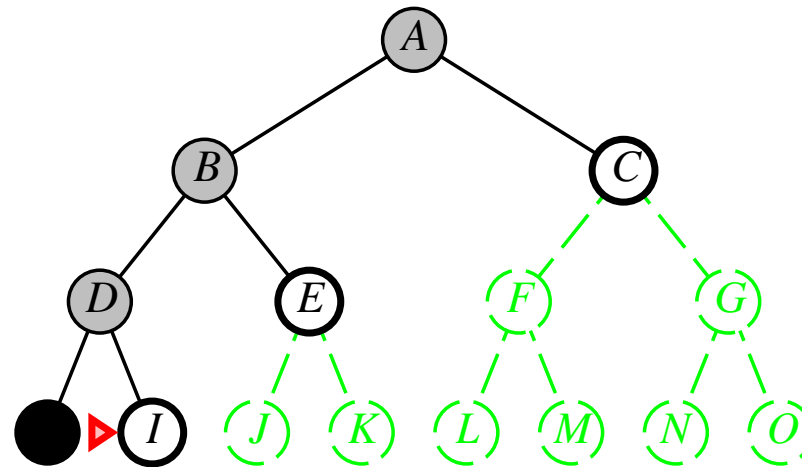


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front

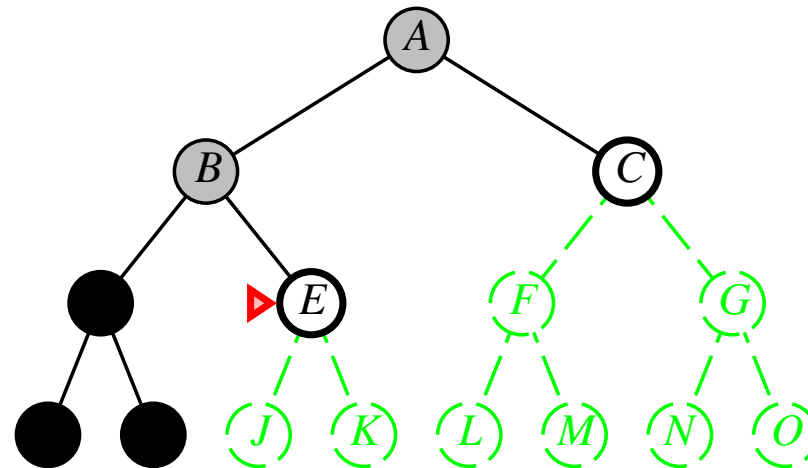


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front



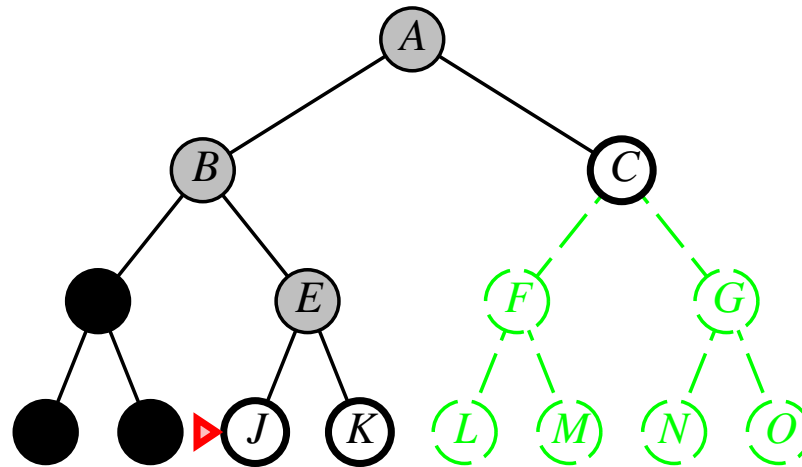


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

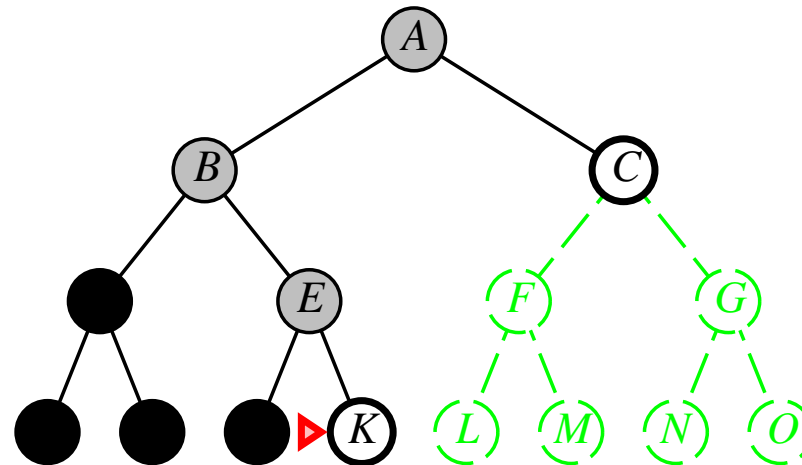


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front

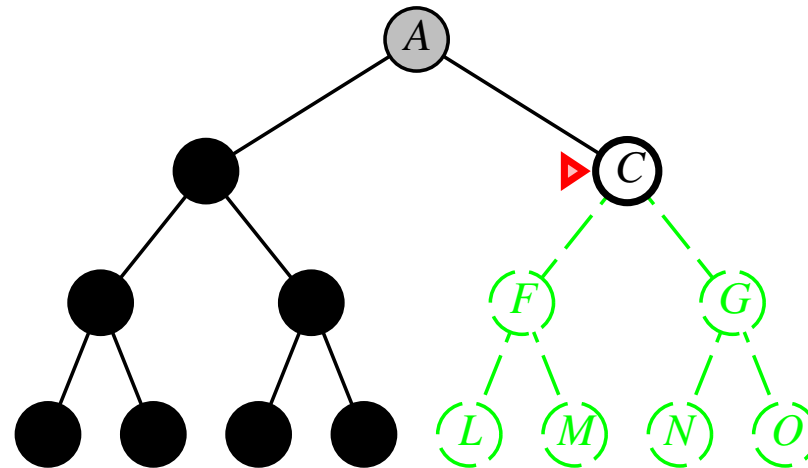


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front

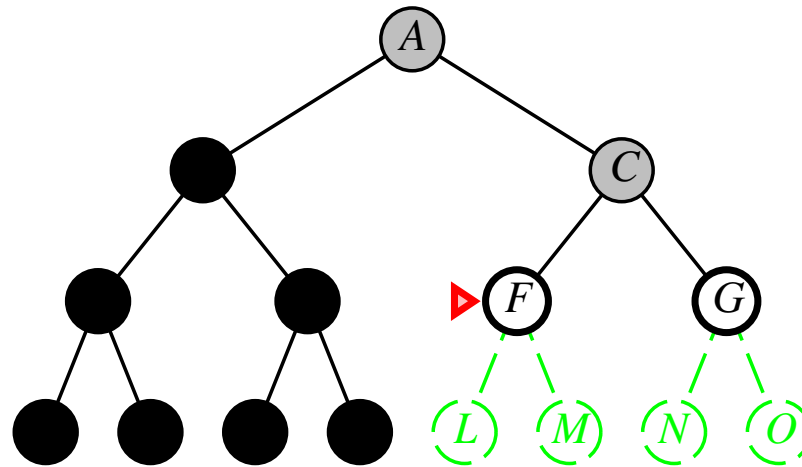


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front

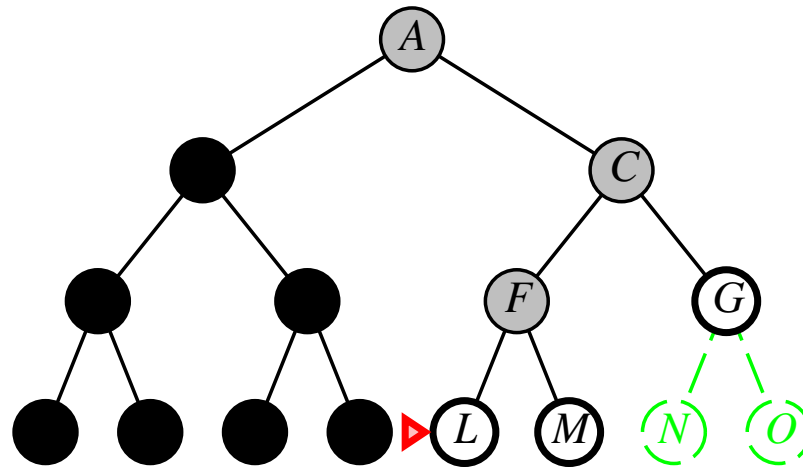


## Depth-first search

## Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

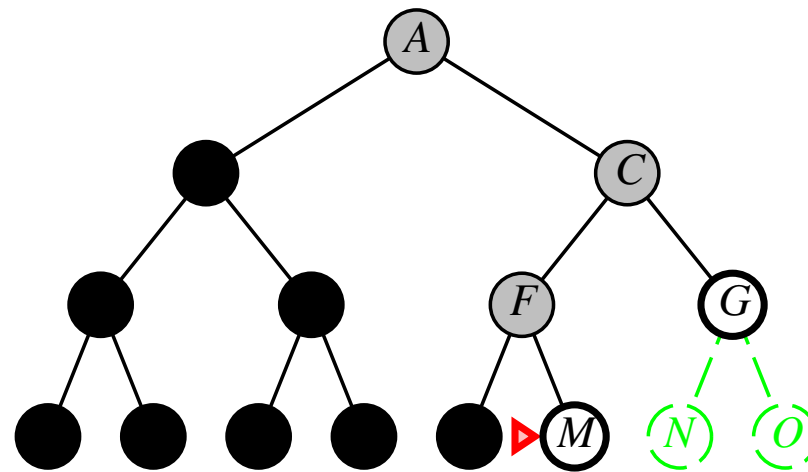


# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front



## 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  $l$ ,  
i.e., nodes at depth  $l$  have no successors

### Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred?  $\leftarrow$  false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred?  $\leftarrow$  true
    else if result  $\neq$  failure then return result
  if cutoff-occurred? then return cutoff else return failure
```



## Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  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
```

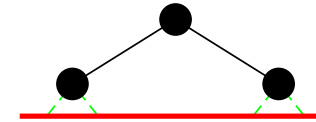
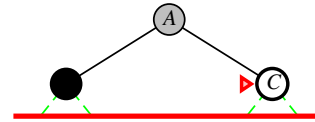
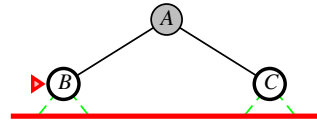
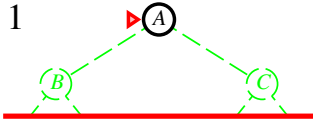
# Iterative deepening search $l = 0$

Limit = 0



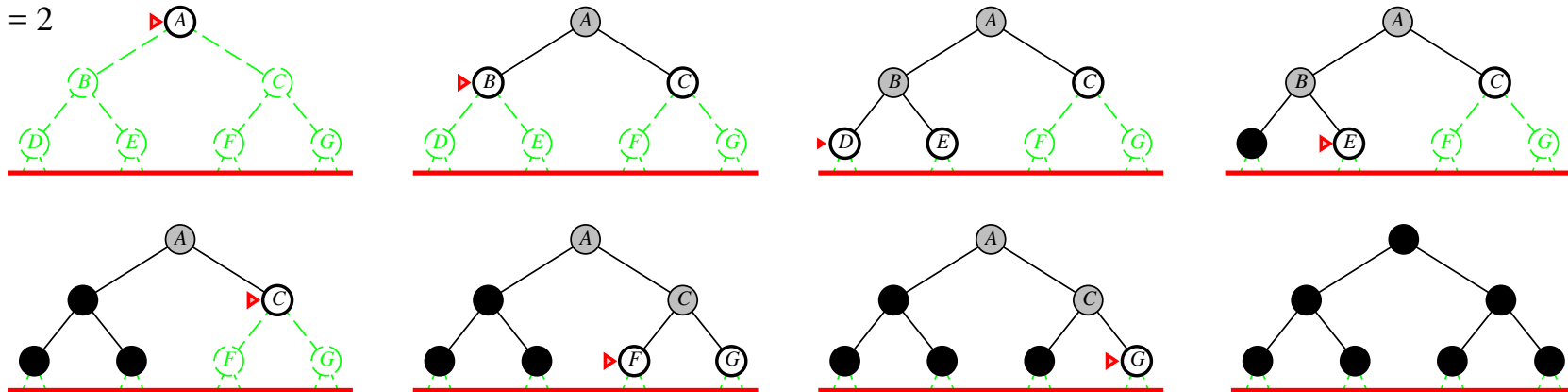
# Iterative deepening search $l = 1$

Limit = 1



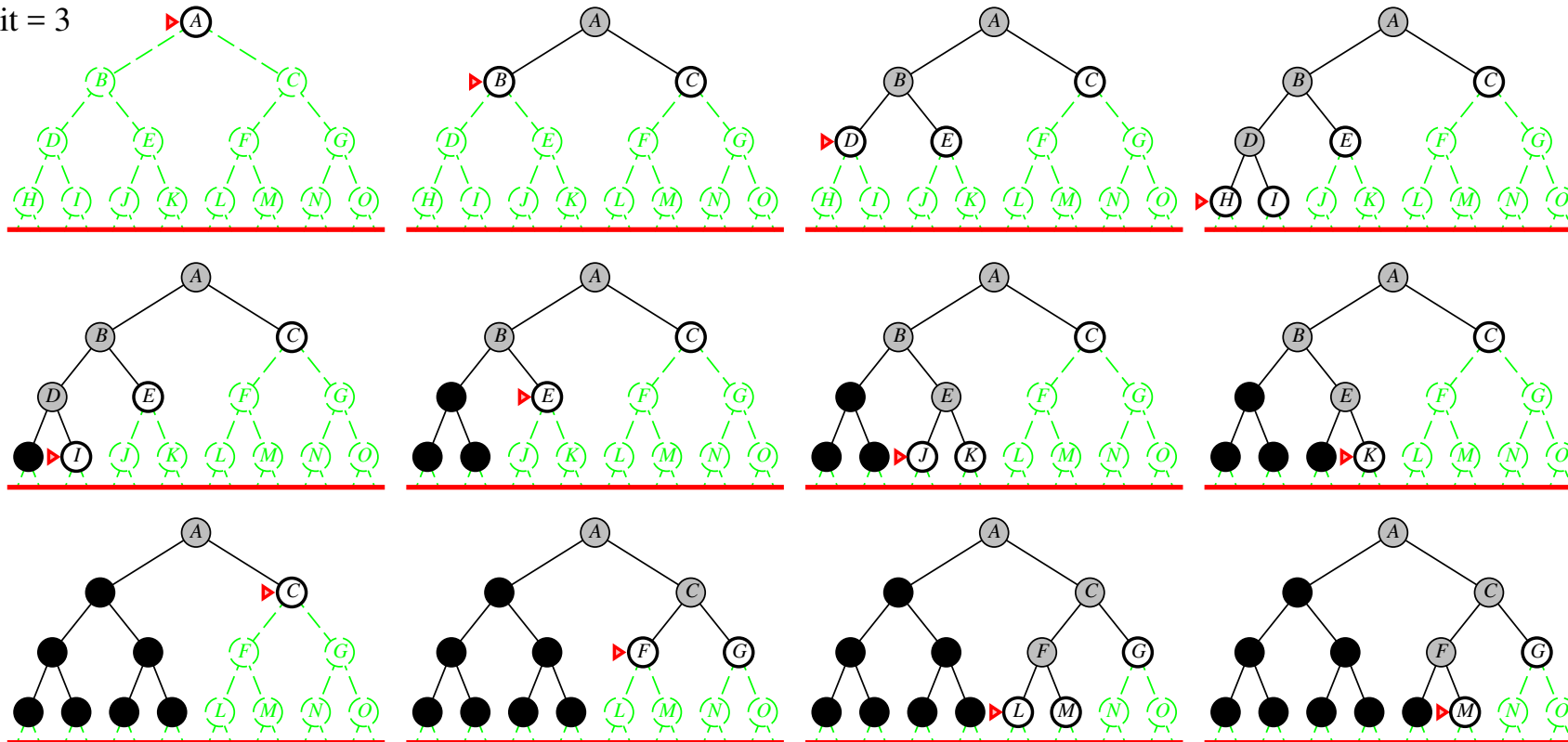
# Iterative deepening search $l = 2$

Limit = 2



# Iterative deepening search $l = 3$

Limit = 3



## 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

Numerical comparison for  $b = 10$  and  $d = 5$ , solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth  $d$  are not expanded

BFS can be modified to apply goal test when a node is **generated**

## Summary of algorithms

Criterion	Breadth- First	Uniform- Cost	Depth- First	Depth- Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	$b^{d+1}$	$b^{\lceil C^*/\epsilon \rceil}$	$b^m$	$b^l$	$b^d$
Space	$b^{d+1}$	$b^{\lceil C^*/\epsilon \rceil}$	$bm$	$bl$	$bd$
Optimal?	Yes*	Yes	No	No	Yes*

## Informed search: Outline

- ◇ Best-first search
- ◇  $A^*$  search
- ◇ Heuristics



## Best-first search

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

⇒ Expand most desirable unexpanded node

Implementation:

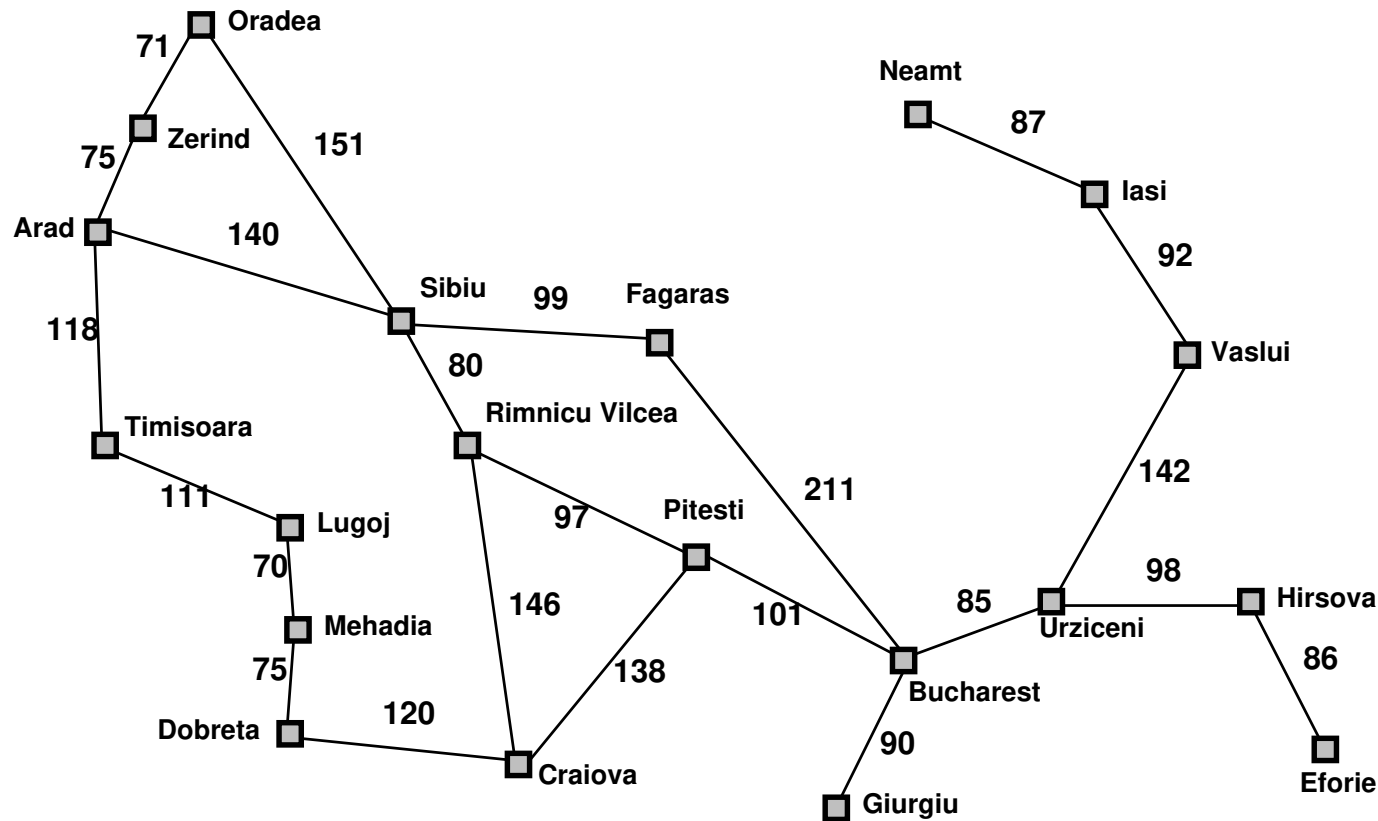
*fringe* is a queue sorted in decreasing order of desirability

Special cases:

greedy search

A\* search

# Romania with step costs in km



Straight-line distance  
to Bucharest

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

## Greedy search

Evaluation function  $h(n)$  (**h**euristic)

= estimate of cost from  $n$  to the closest goal

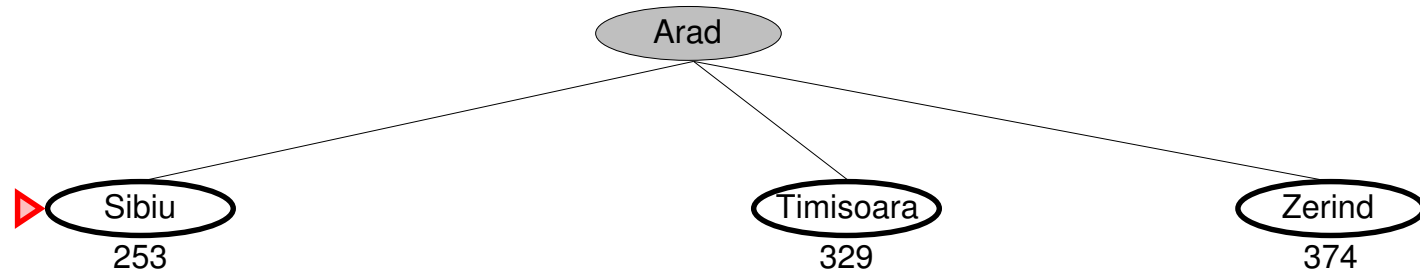
E.g.,  $h_{SLD}(n)$  = straight-line distance from  $n$  to Bucharest

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

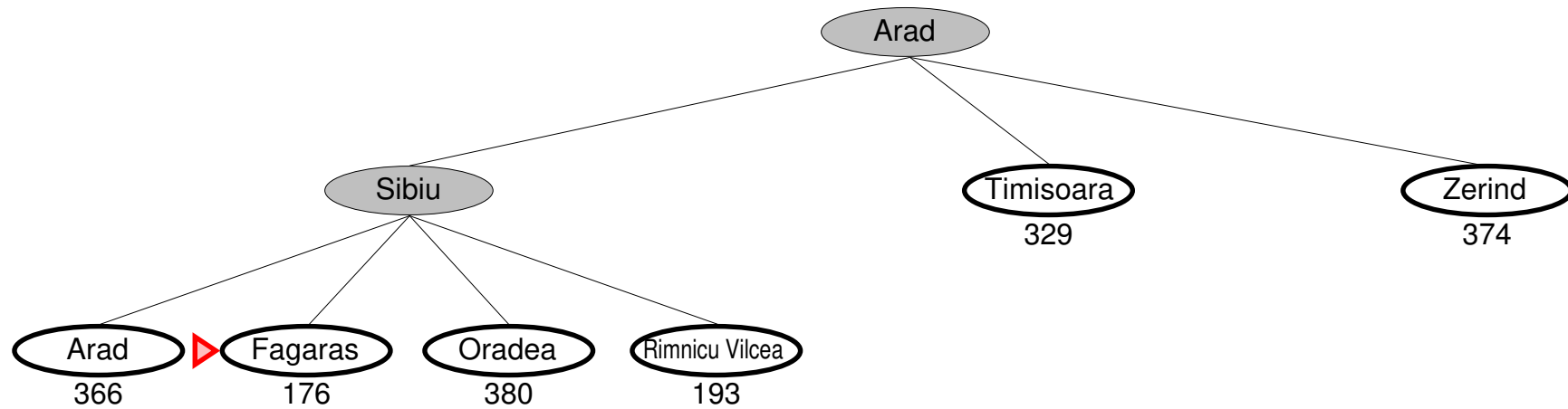
## Greedy search example

▶ Arad  
366

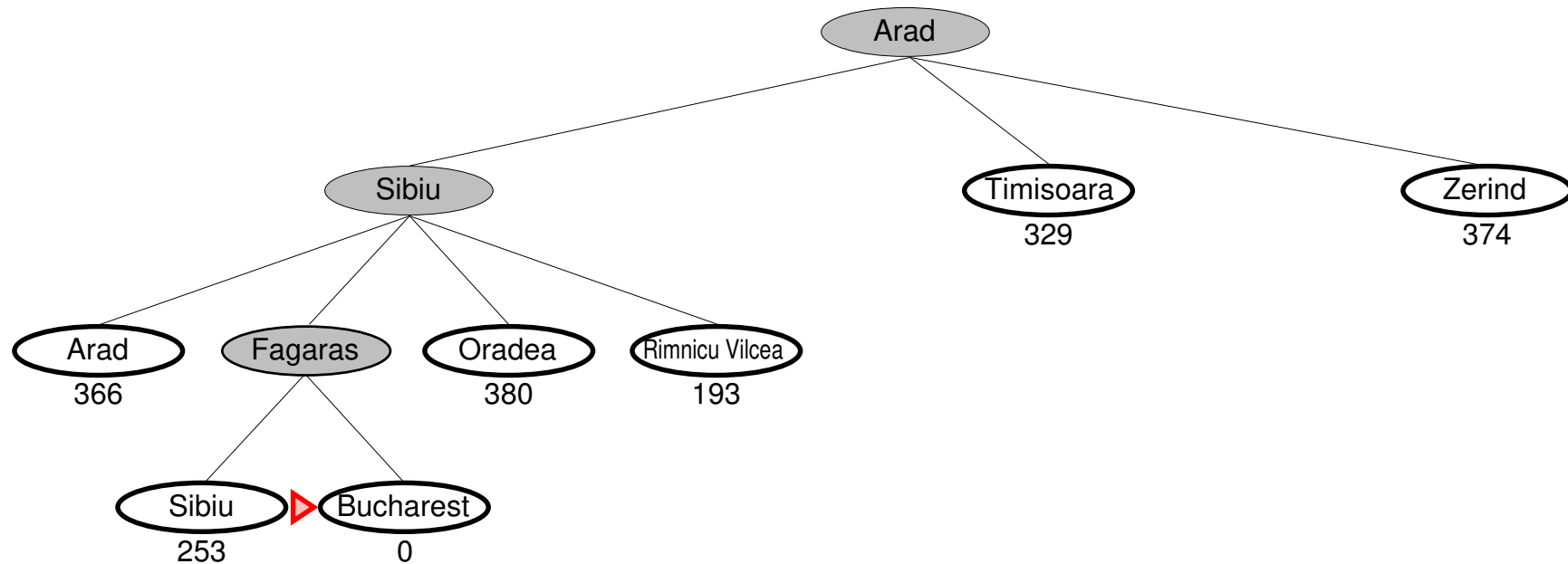
## Greedy search example



# Greedy search example



# Greedy search example



## Properties of greedy search

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

lasi  $\rightarrow$  Neamt  $\rightarrow$  lasi  $\rightarrow$  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  $n$ .

(Also require  $h(n) \geq 0$ , so  $h(G) = 0$  for any goal  $G$ .)

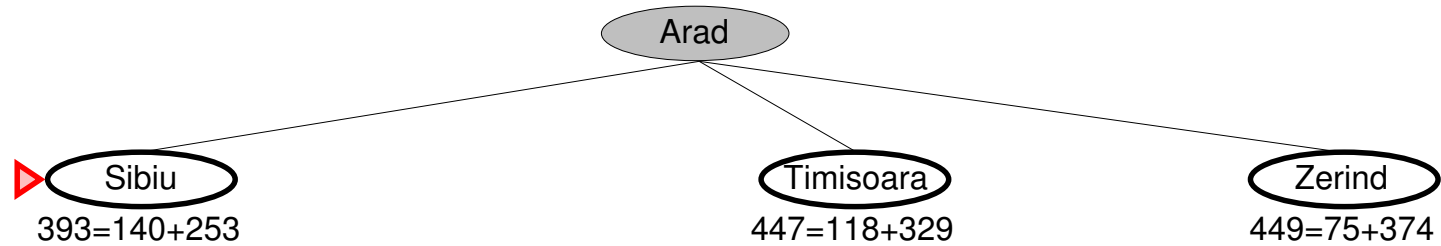
E.g.,  $h_{\text{SLD}}(n)$  never overestimates the actual road distance

**Theorem:** A\* search is optimal

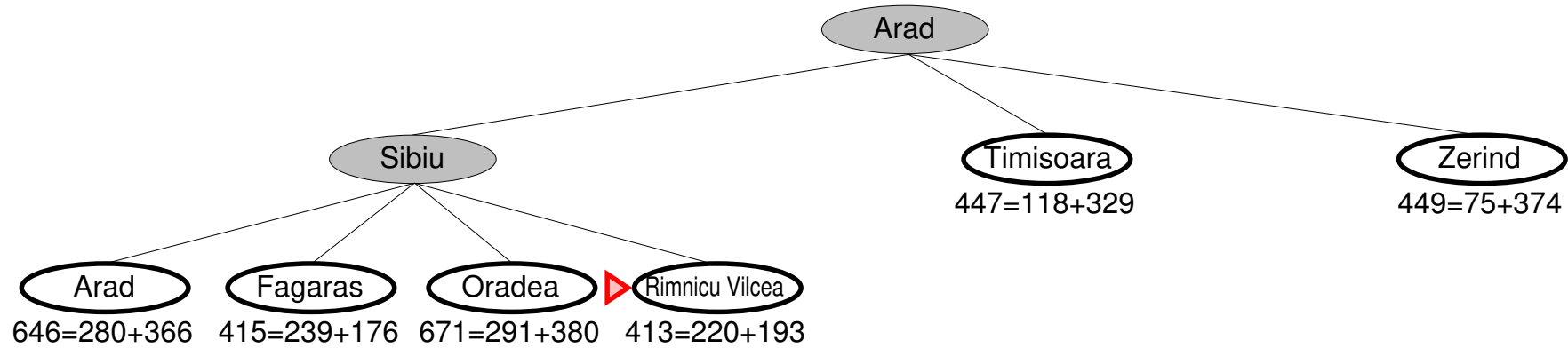
## A\* search example

▶ Arad  
 $366 = 0 + 366$

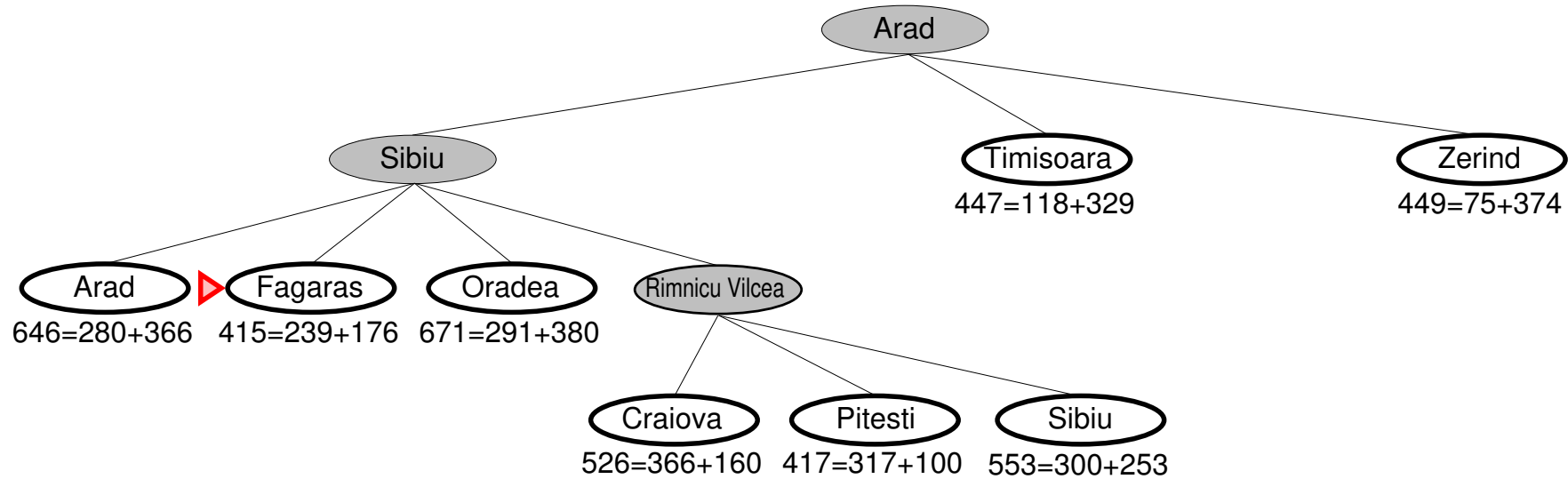
## A\* search example



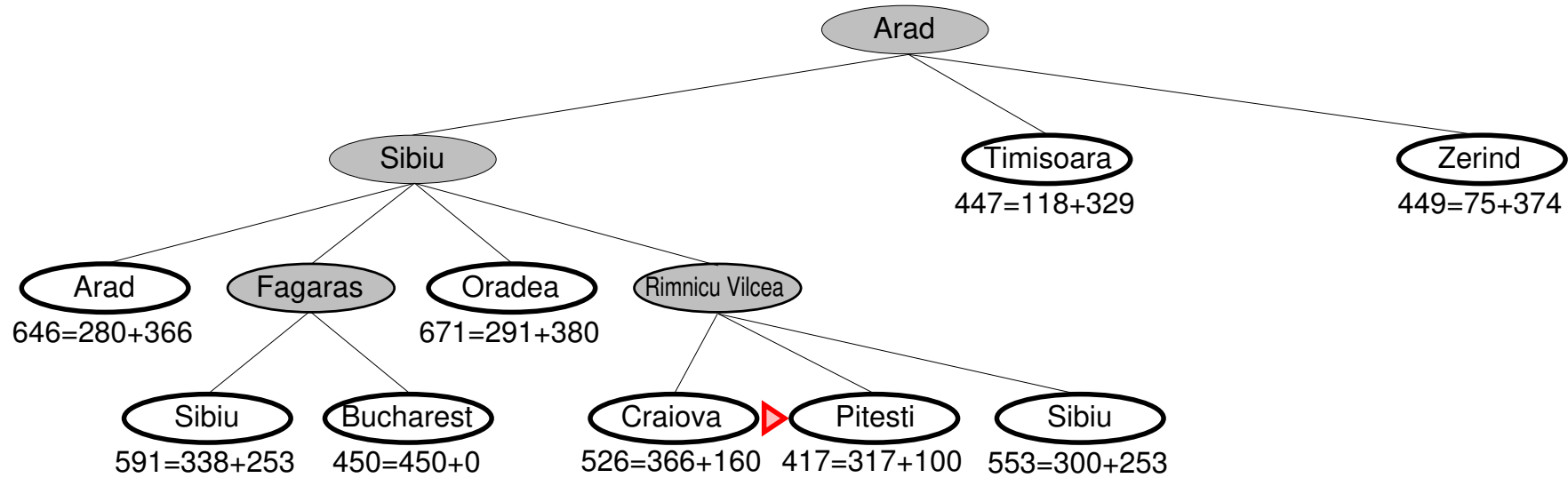
# A\* search example



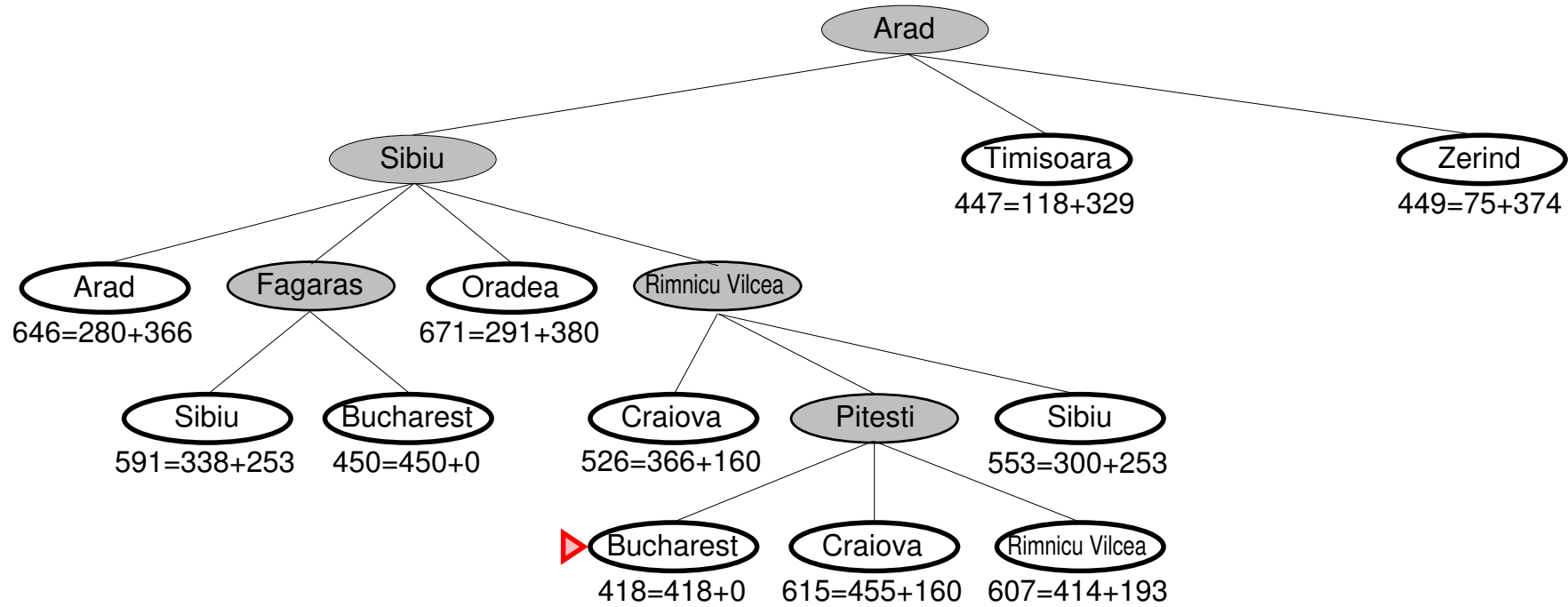
# A\* search example



# A\* search example



# A\* search example



## 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

$A^*$  expands all nodes with  $f(n) < C^*$

$A^*$  expands some nodes with  $f(n) = C^*$

$A^*$  expands no nodes with  $f(n) > C^*$



## Admissible heuristics

E.g., for 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)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$$\underline{h_1(S) = ??}$$

$$\underline{h_2(S) = ??}$$

# Admissible heuristics

E.g., for 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)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$h_1(S) = ??$  6

$h_2(S) = ??$   $4+0+3+3+1+0+2+1 = 14$

## 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

$d = 24$  IDS  $\approx$  54,000,000,000 nodes

$A^*(h_1) = 39,135$  nodes

$A^*(h_2) = 1,641$  nodes

Given any admissible heuristics  $h_a, h_b$ ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates  $h_a, h_b$

## 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

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

## 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

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest  $h$   
– incomplete and not always optimal

A\* search expands lowest  $g + h$   
– complete and optimal  
– also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems