Artificial Intelligence

Search

Marc Toussaint
University of Stuttgart
Winter 2019/20

(slides based on Stuart Russell's AI course)

Motivation:

Search algorithms are a core tool for sequential decision making! They a standard method of choice for complex domains, like the game of Go, or certain POMDPs, but Mixed-Integer Programming or other combinatorial problems. Search algorithms provide a robust framework for decision making – the real smarts is often within the heuristic that guides and prioritizes search. Major parts of classical AI research is about intelligent heuristics. And machine learning methods are often used to learn heuristics or evaluation functions. This embeds learning capabilities in a robust decision making framework.

Learning about search tree algorithms is an important background for several reasons:

- The concept of decision trees, which represent the space of possible future decisions and state transitions, is generally important for thinking about decision problems.
- In probabilistic domains, tree search algorithms are a special case of Monte-Carlo methods to estimate some expectation, typically the so-called Q-function. The respective Monte-Carlo Tree Search algorithms are the state-of-the-art in many domains.
- Tree search is also the background for backtracking in CSPs as well as forward and backward search in logic domains.
- **Heuristics & bounds** are fundamental in AI. Abstraction & simplification is often best formalized in terms of heuristics & bounds. Related concepts are: Heuristics in A*, Branch & Bound for Mixed-Integer Programming, Multi-Bound Tree Search in Logic Geometric Programming, Angelic Semantics, Optimism (=lower bound) in the face of Uncertainty. Learning about tree search is the starting point to appreciate such concepts.

We will cover the basic tree search methods (breadth, depth, iterative deepening) and eventually A*

What is AI?

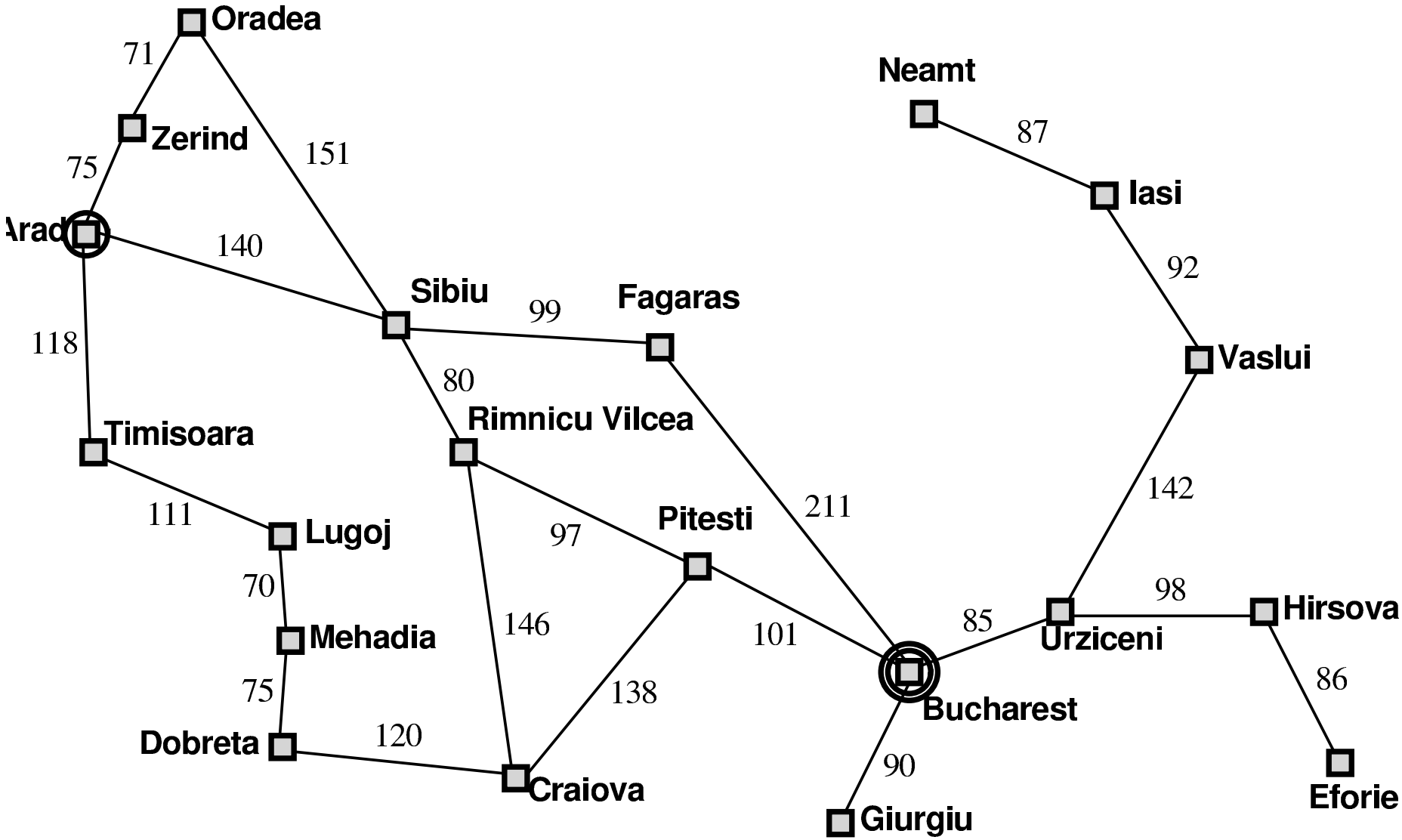
- AI is a research field
- AI research is about systems that take decisions
  - **Here: sequential decisions**
- ... systems that take optimal/desirable decisions
  - **Here: sequential decisions towards a "goal"**
- ... systems that take optimal decisions on the basis of all available information
  - **Here: The decision process is fully known**
    (later in the lecture: Reinforcement Learning)

Outline

- Problem formulation & examples
- Basic search algorithms

Problem Formulation & Examples

Example: Romania



Example: Romania

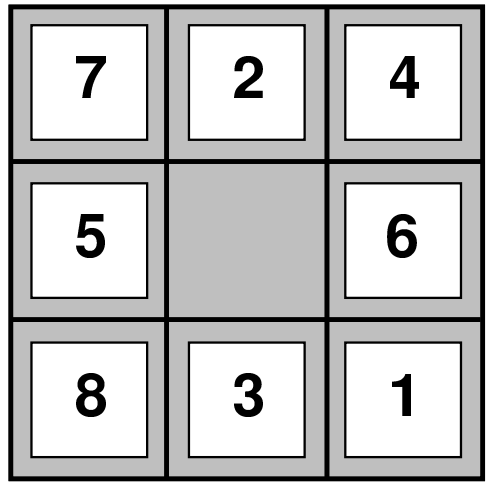
- initial state: $s_0 =$Arad
- goal: be in Bucharest, $\mathcal{S}_{\text{goal}} = \{\text{Bucharest}\}$
- problem formulation:
  - states: various cities, $\mathcal{S} = \{\text{Arad, Timisoara, } \dots\}$
  - decisions: drive between cities, $\mathcal{A} = \{\text{edges between states}\}$
  - problem: find sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest minimize costs with cost function, $(s, a) \mapsto c$

Deterministic, fully observable search problem

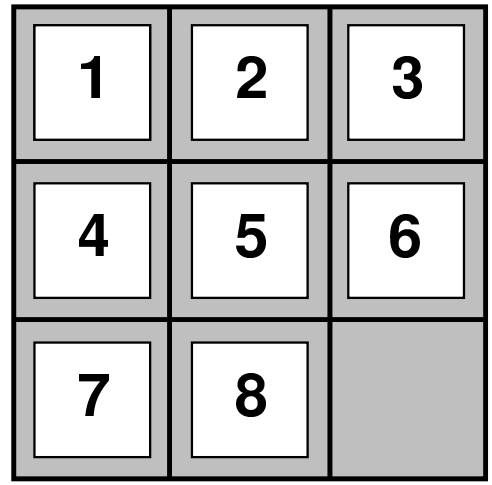
- A deterministic, fully observable search problem is defined by four items:
  - initial state $s_0 \in \mathcal{S}$ e.g., $s_0 =$ Arad
  - successor function $succ : \mathcal{S} \to \mathcal{S}^*$ maps to set of decisions
    e.g., $succ(\text{Arad}) = \{\text{Sibiu, Timisoara, Zerind}\}$
  - goal states $\mathcal{S}_{\text{goal}} \subseteq \mathcal{S}$, e.g., $\mathcal{S}_{\text{goal}} = \{\text{Bucharest}\}$
  - step cost function $cost(s, s')$, assumed to be ≥ 0,
    e.g., traveled distance, number of actions executed, etc.
    the path cost is the sum of step costs

- A solution is a sequence of decisions leading from s_0 to a goal
- An optimal solution is a solution with minimal path costs

Example: The 8-puzzle



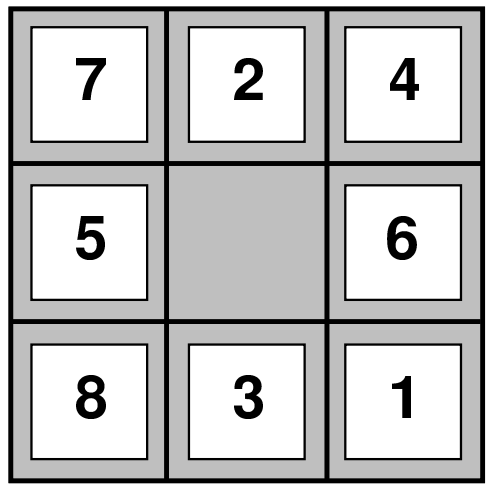
Start State



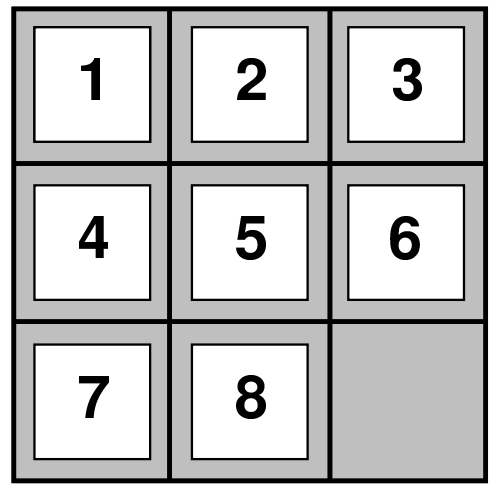
Goal State

states??:

Example: The 8-puzzle



Start State

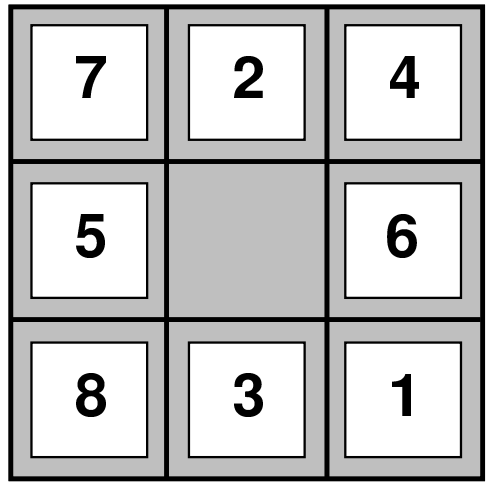


Goal State

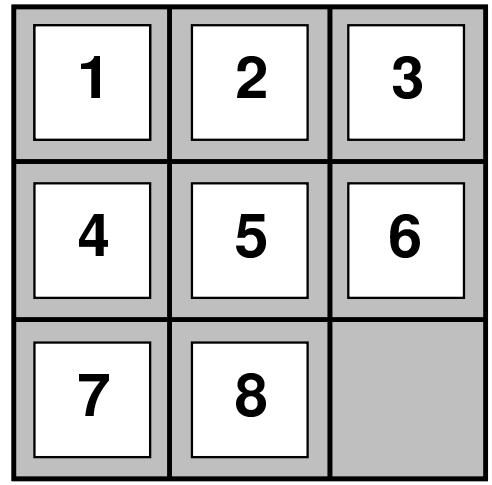
states??: integer locations of tiles (ignore intermediate positions)

decisions??:

Example: The 8-puzzle



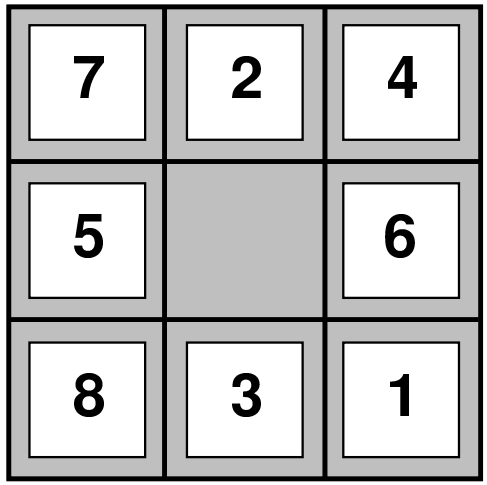
Start State



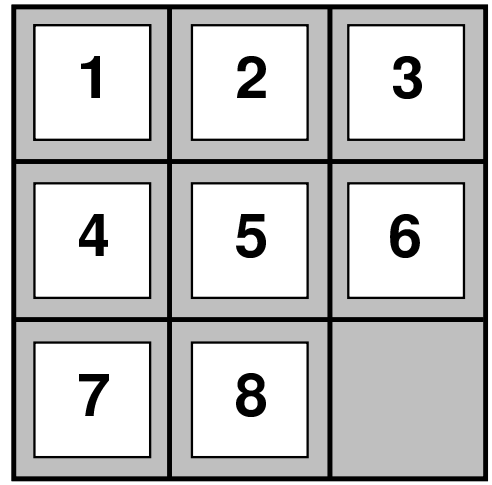
Goal State

states??: integer locations of tiles (ignore intermediate positions)
decisions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??:

Example: The 8-puzzle



Start State



Goal State

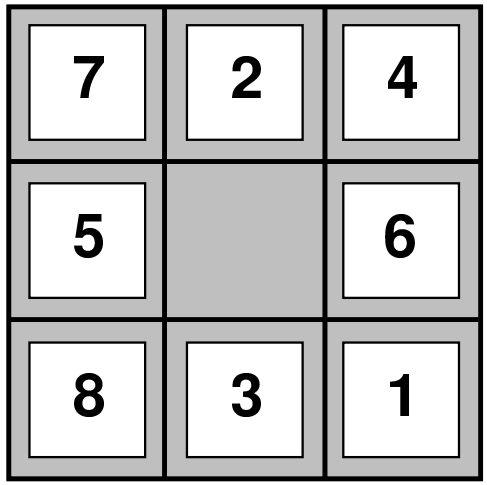
states??: integer locations of tiles (ignore intermediate positions)

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

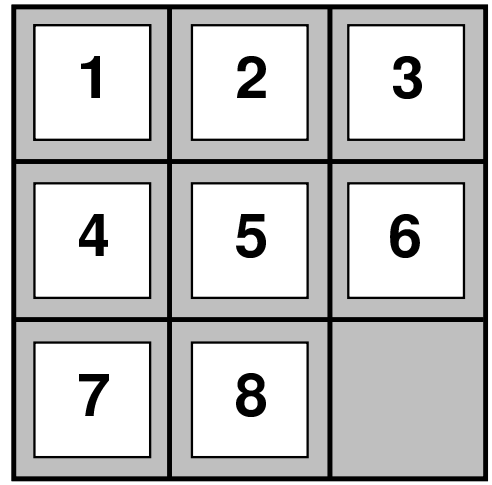
goal test??: = goal state (given)

path cost??:

Example: The 8-puzzle



Start State



Goal State

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

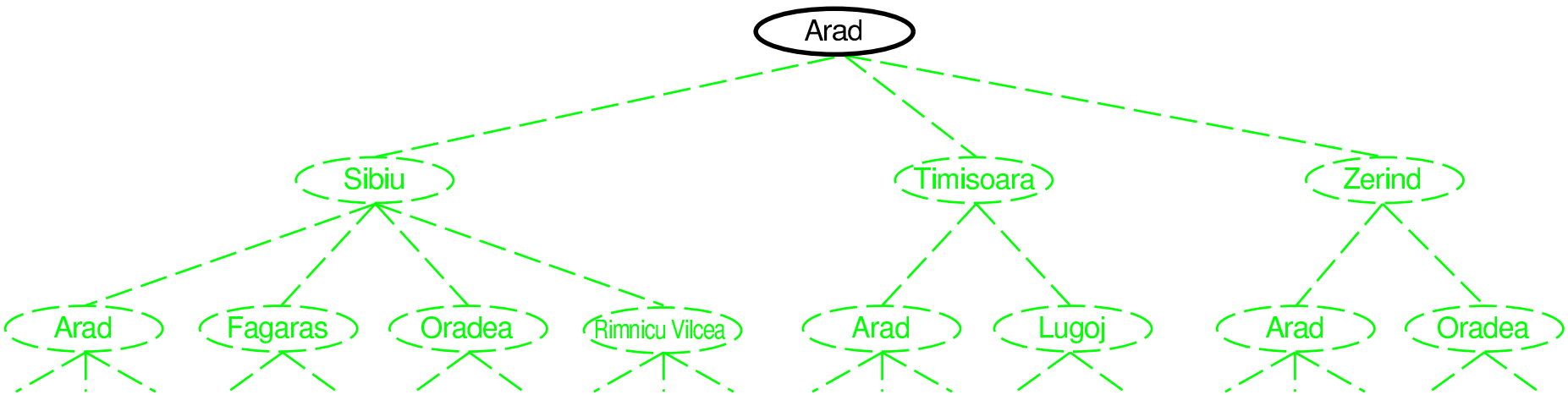
decisions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

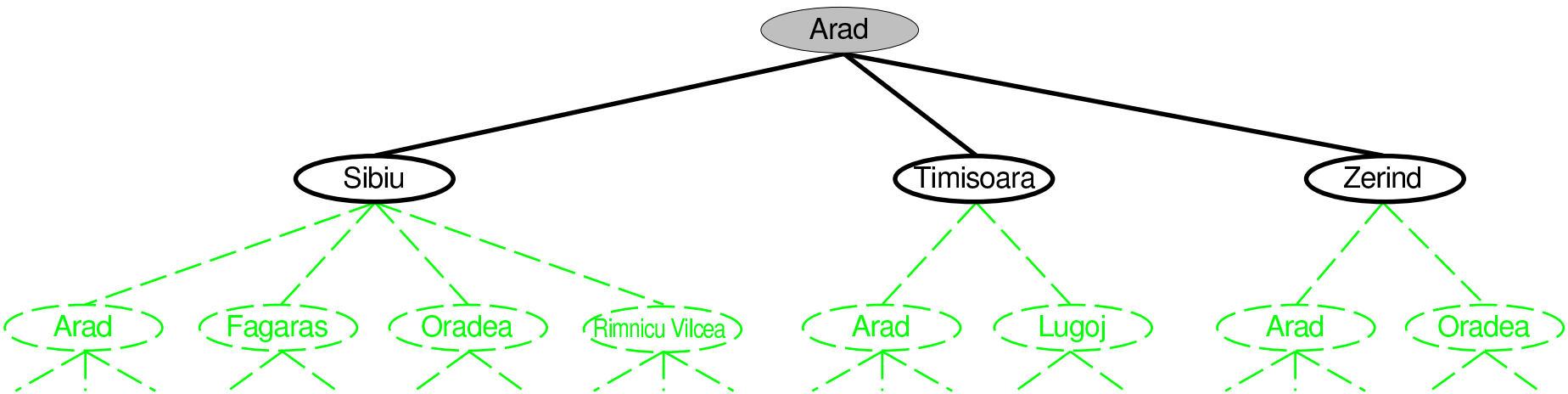
path cost??: 1 per move

Basic Tree Search Algorithms

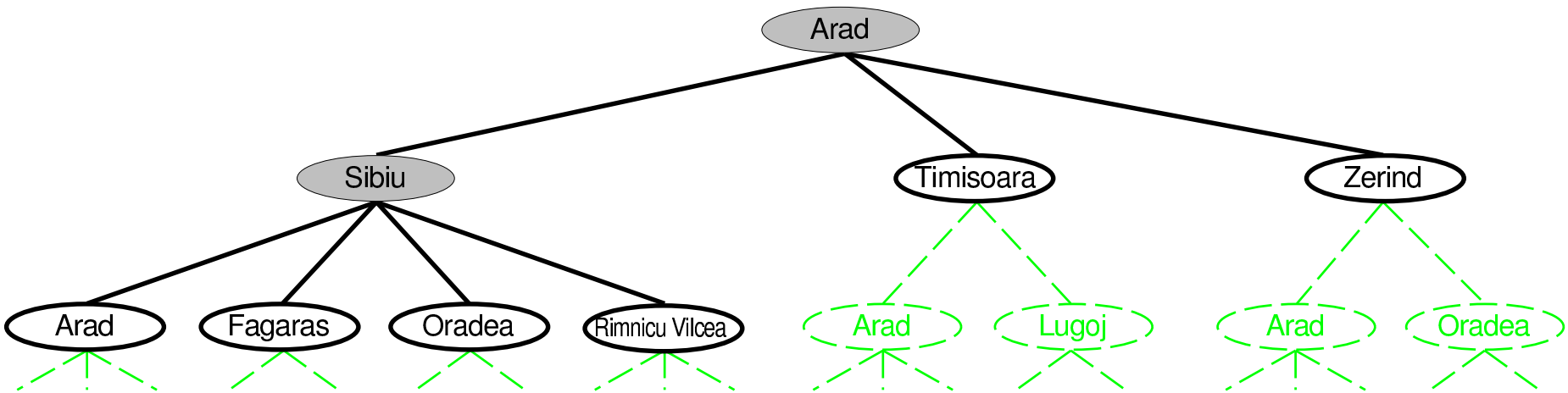
Tree search example



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 $\textsc{Expand}$ function creates new nodes, filling in the various fields and using the $\textsc{SuccessorFn}$ 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
```

Implementation: general tree search

```
1:  procedure TreeSearch(s_0, S_goal, succ, cost)
2:      root ← Node(state=s_0, parent=nil, cost=0, depth=0)
3:      fringe ← ⟨root⟩
4:      while fringe is not empty do
5:          n ← fringe.pop()                          // pop node from fringe
6:          if(n.state∈ S_goal) return n              // goal check
7:          fringe.insert( Expand(n, succ, cost) )    // expand
8:          optional: if n.children={}, destroy it and also check ancestors
9:      end while
10:     return nil
11: end procedure

12: procedure Expand(n, succ, cost)
13:     successors ← {}
14:     for s ∈ succ(n.state) do
15:         n' ← Node(state=s, parent=n, cost=n.cost+c(n.state, s), depth=n.depth+1)
16:         successors.append(n')
17:     end for
18:     return successors
19: end procedure
```

Search strategies

- A strategy is defined by picking the *ordering of the fringe*
- 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 ∞)

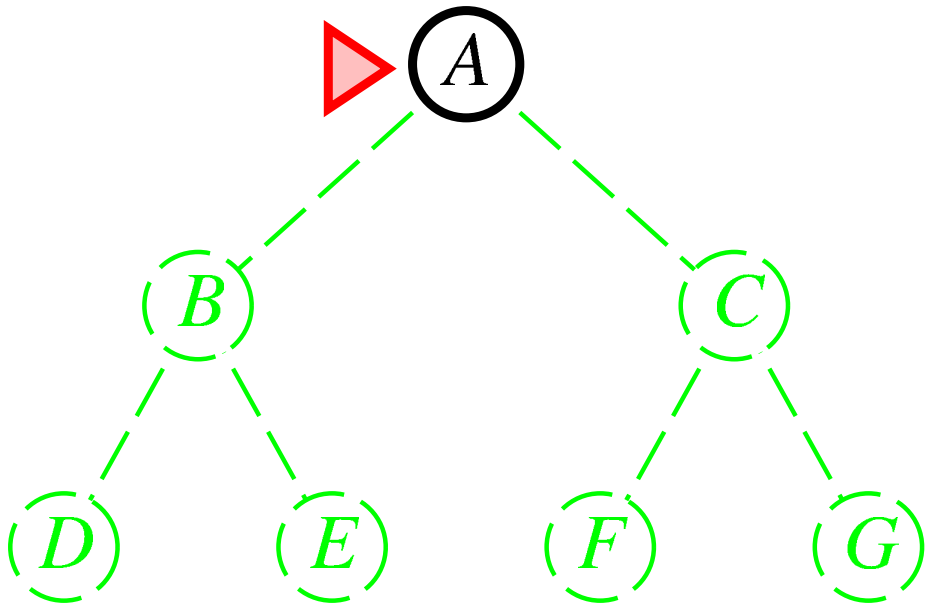
Summary of Search Strategies

- Breadth-first: fringe is a FIFO
- Depth-first: finge is a LIFO
- Iterative deepening search: repeat depth-first for increasing depth limit
- Uniform-cost: sort fringe by g
- A*: sort by $f = g + h$

Breadth-first search

- Pick shallowest unexpanded node
- *Implementation*:

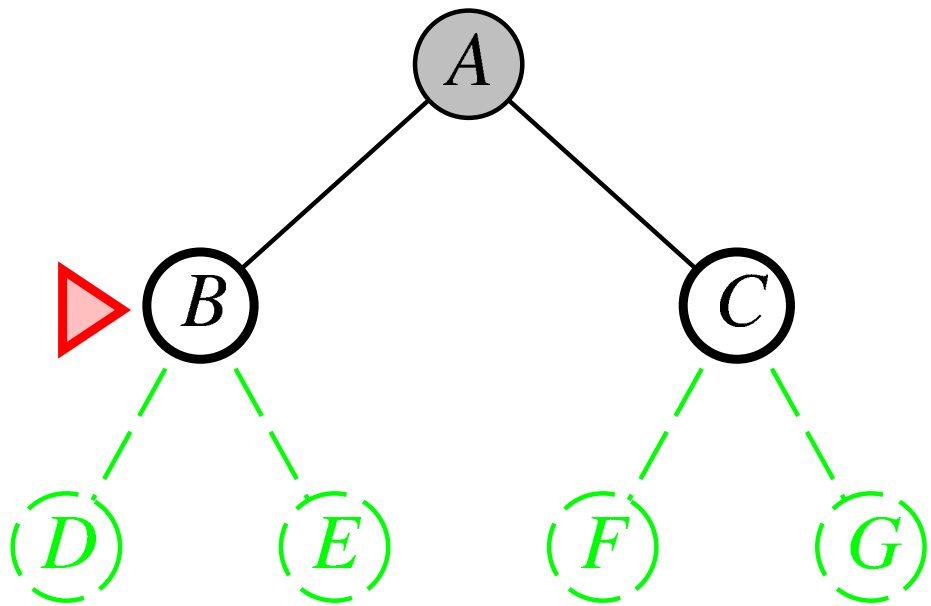
  fringe is a **FIFO** queue, i.e., new successors go at end



Breadth-first search

- Pick shallowest unexpanded node
- *Implementation*:

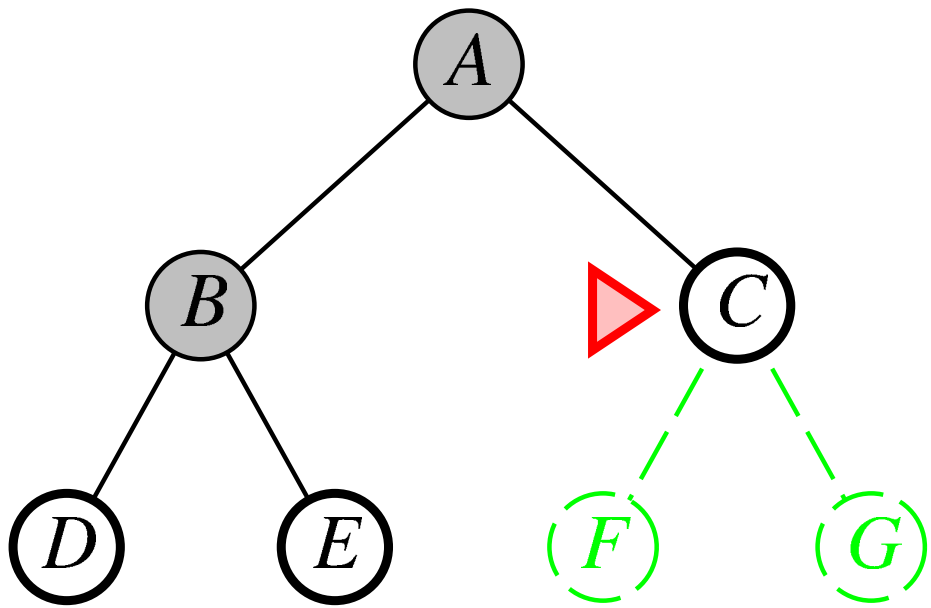
  fringe is a **FIFO** queue, i.e., new successors go at end



Breadth-first search

- Pick shallowest unexpanded node
- *Implementation*:

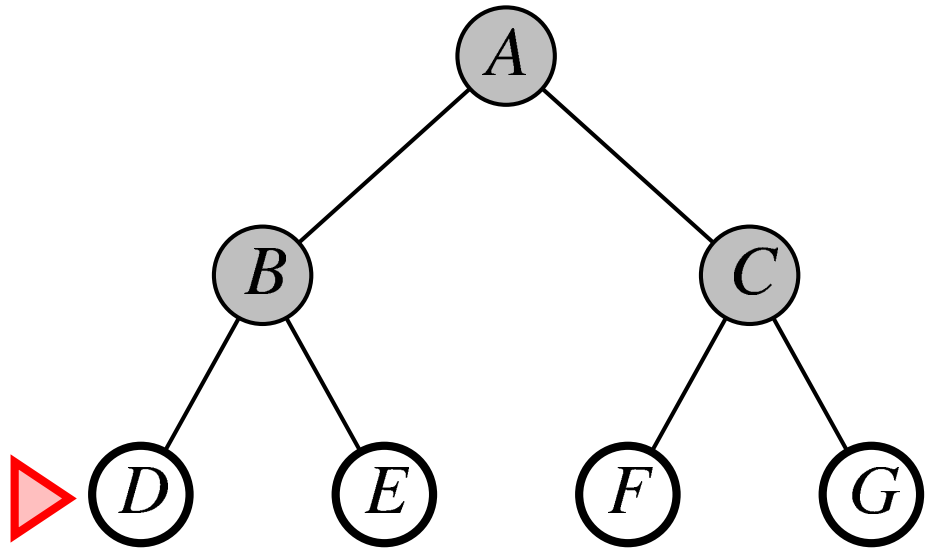
  fringe is a **FIFO** queue, i.e., new successors go at end



Breadth-first search

- Pick shallowest unexpanded node
- *Implementation*:

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



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space?? $O(b^{d+1})$ (keeps every node in memory)

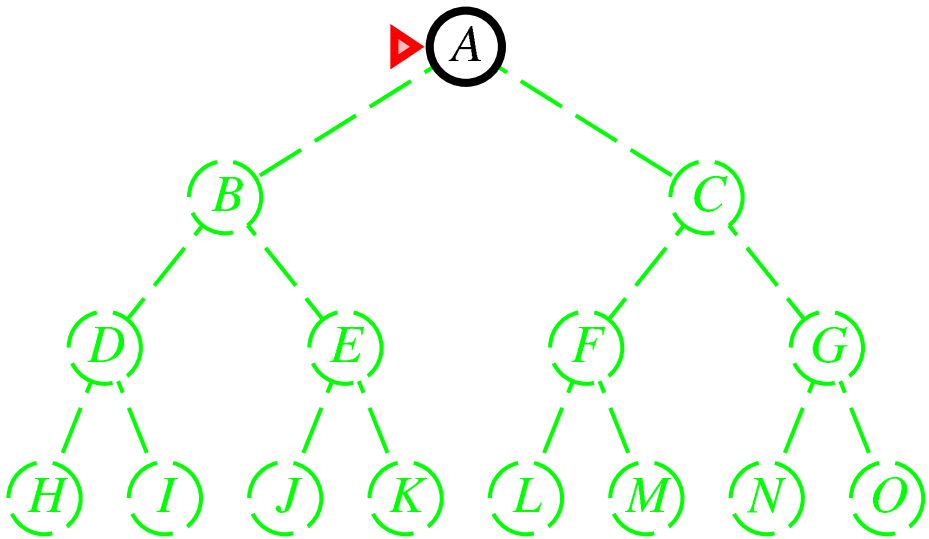
Optimal?? Yes, if cost-per-step=1; not optimal otherwise

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

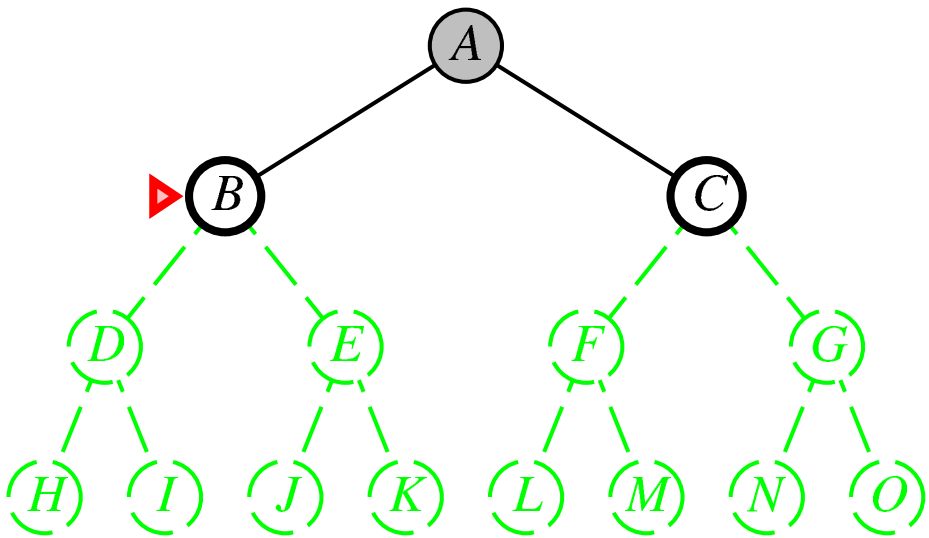
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

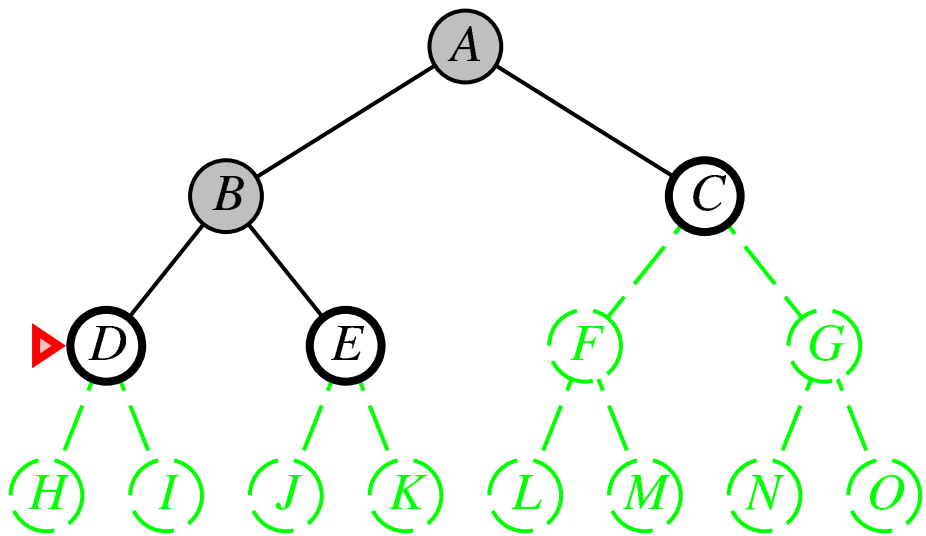
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

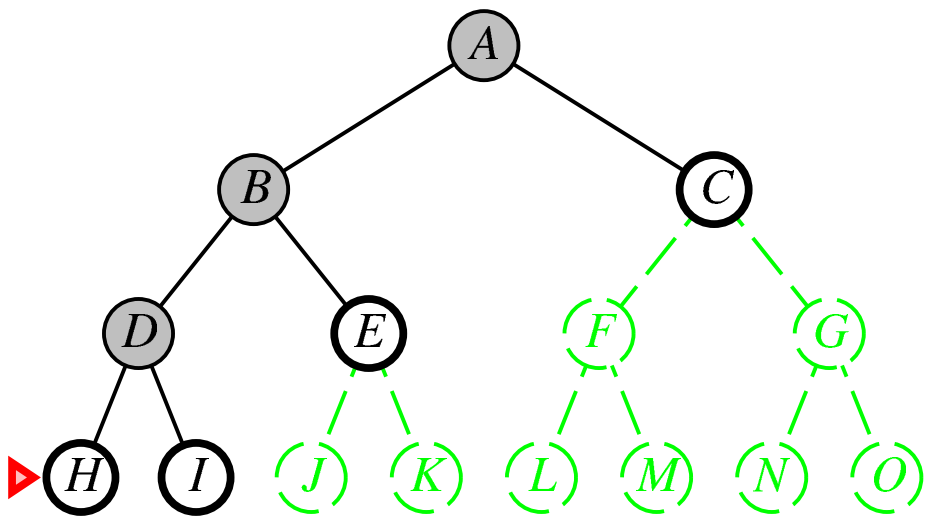
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

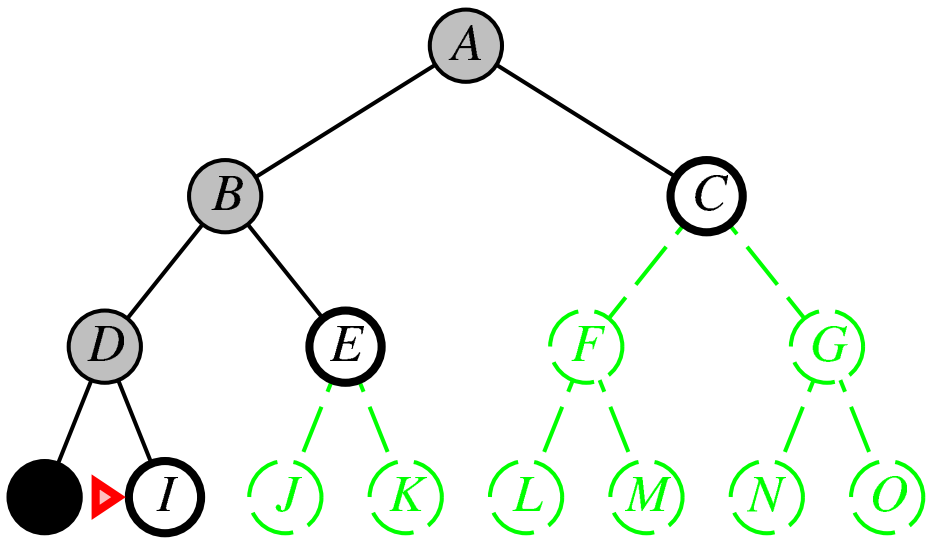
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

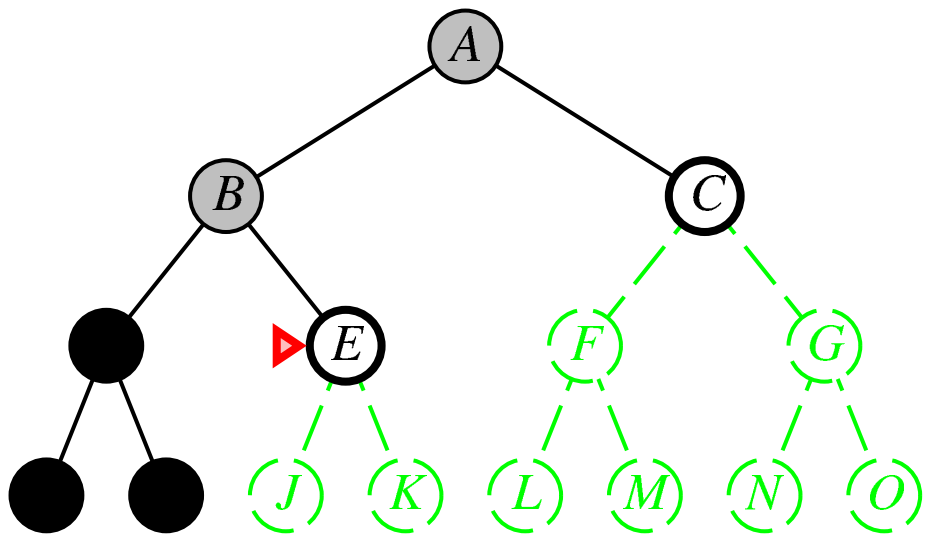
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

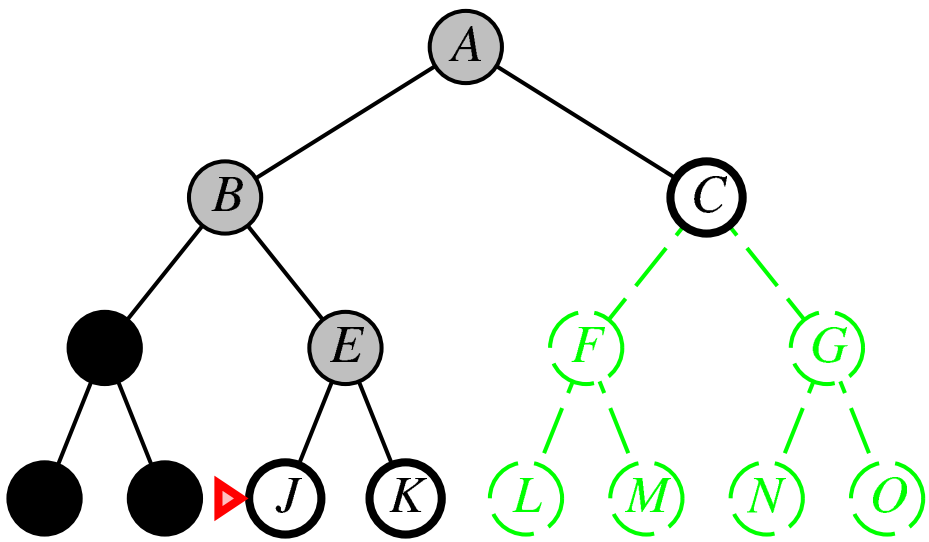
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

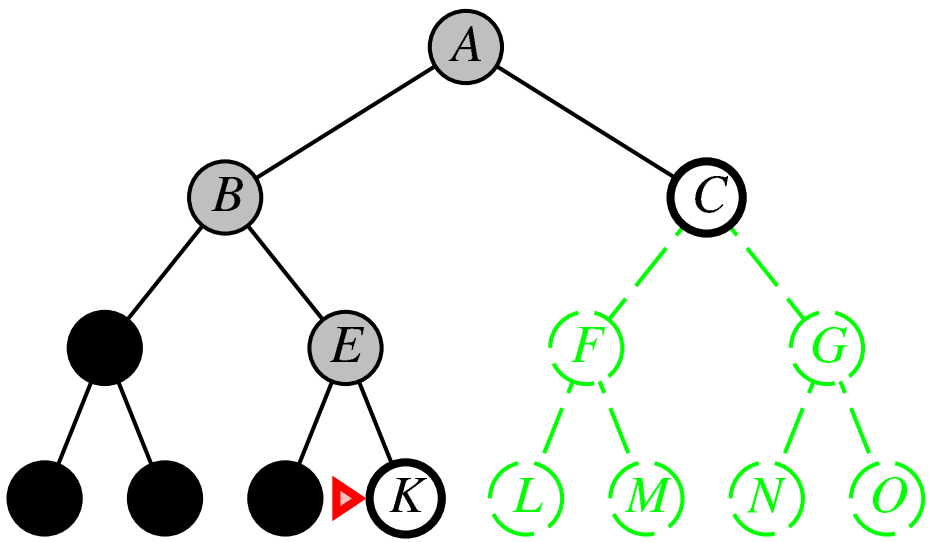
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

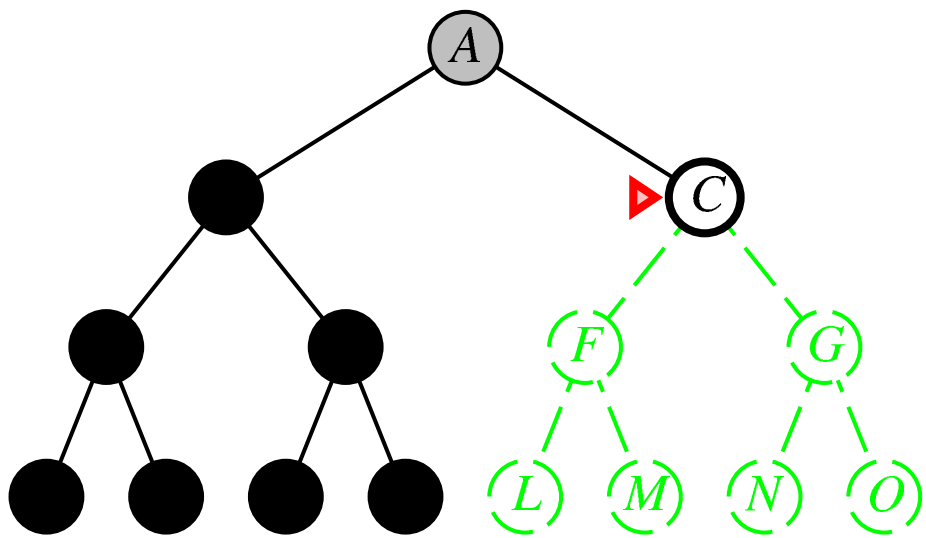
  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Depth-first search

- Pick deepest unexpanded node
- *Implementation*:

  fringe = **LIFO** queue, i.e., put successors at front
- ***Note: The pictures push nodes in non-alphabetical order!***



Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? Yes for $m < \infty$, no otherwise (e.g. problems with loops)

Time??

Properties of depth-first search

Complete?? Yes for $m < \infty$, no otherwise (e.g. problems with loops)

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??

Properties of depth-first search

Complete?? Yes for $m < \infty$, no otherwise (e.g. problems with loops)

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??

Properties of depth-first search

Complete?? Yes for $m < \infty$, no otherwise (e.g. problems with loops)

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
- Optional: *Recursive implementation* using the stack as LIFO:

```
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? ← 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 ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ 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 ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

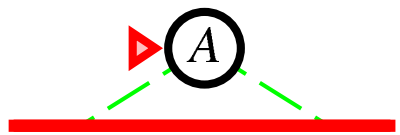
Iterative deepening search

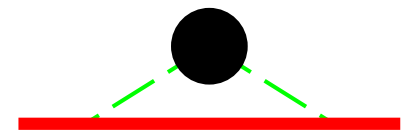
```
1: procedure IterativeDeepeningSearch(s_0, S_goal, succ, cost)
2:     for depth d = 0, 1, .. do
3:         n ← DepthLimitedSearch(d)
4:         if n ≠nil return n
5:     end for
6: end procedure
```

- DepthLimitedSearch(d) is the same as TreeSearch but where **Expand** returns $\{\}$ for n.depth=d, and cleanup is done.

Iterative deepening search

Limit = 0

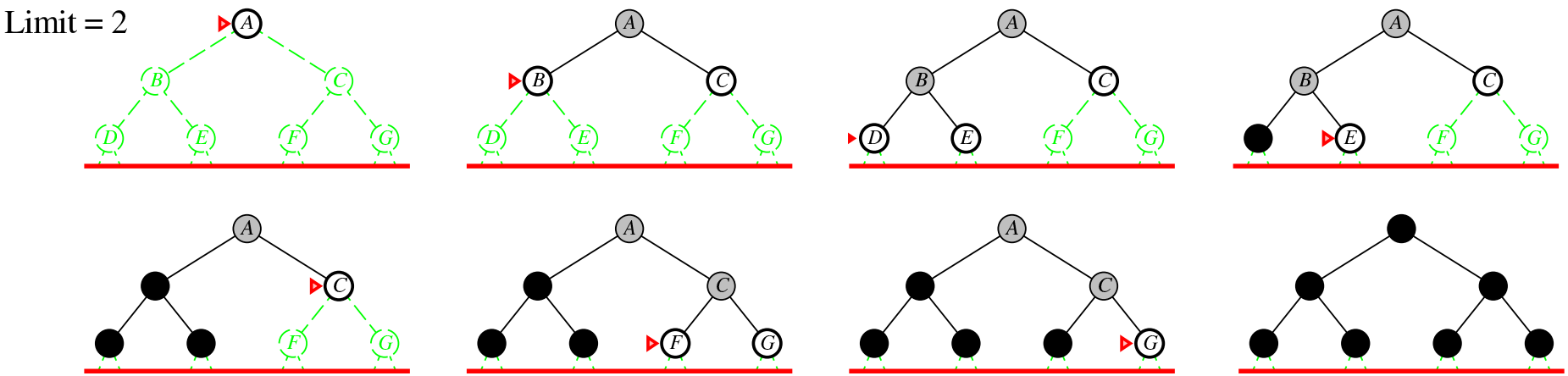




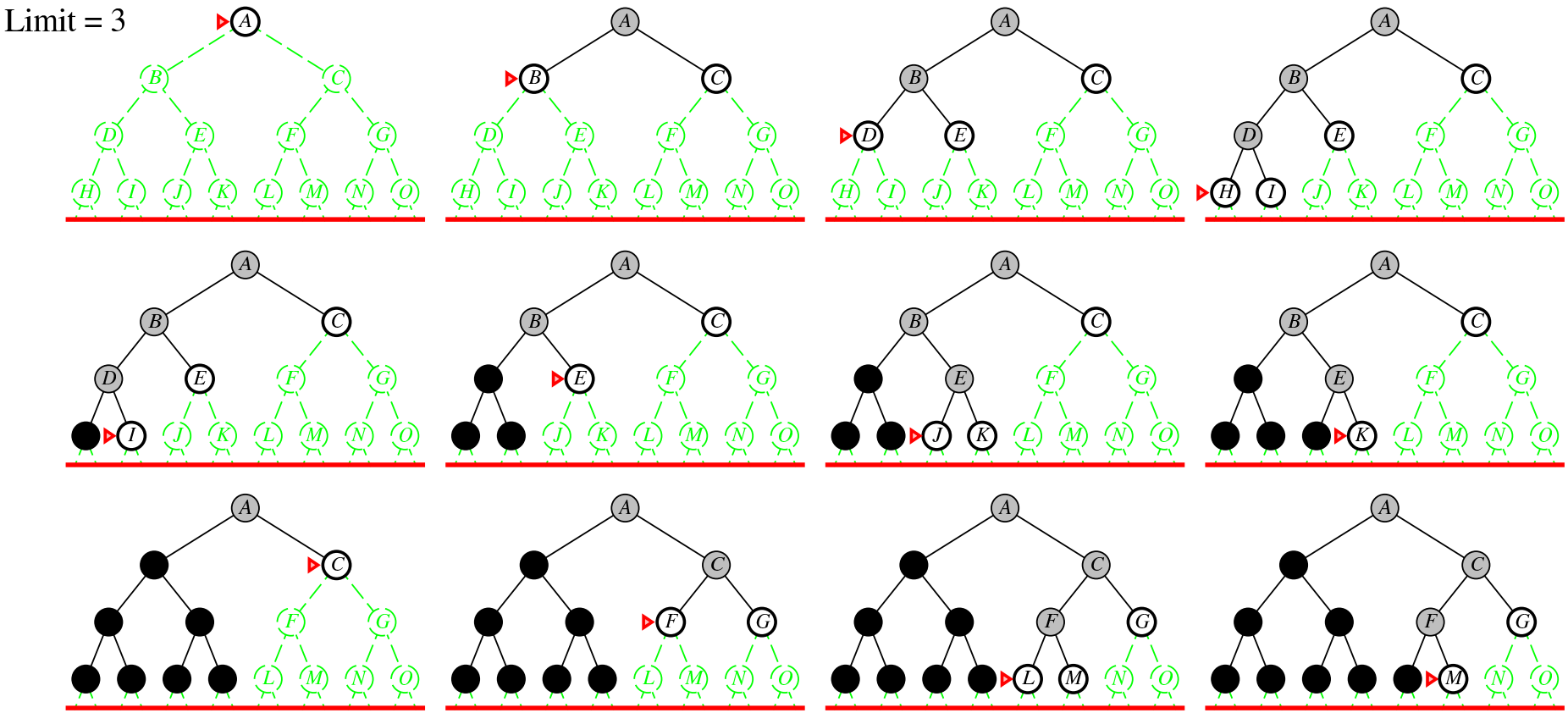
Iterative deepening search

Limit = 1

Iterative deepening search



Iterative deepening search



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

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??

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??

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

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

$$N(\text{IDS}) = 50 + 400 + 3\,000 + 20\,000 + 100\,000 = 123\,450$$
$$N(\text{BFS}) = 10 + 100 + 1\,000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,100$$

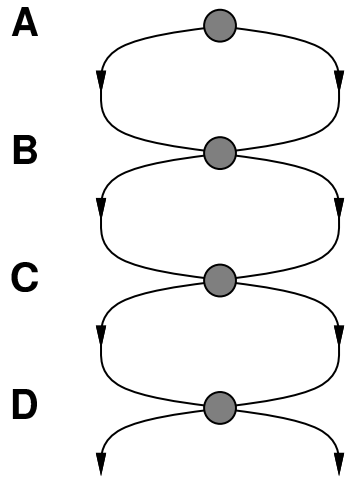
- IDS does better because other nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is *generated*

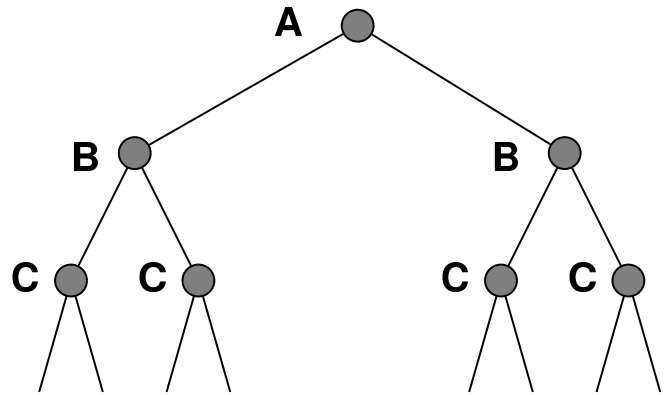
Summary of algorithms

Criterion	Breadth-First	Depth-First	Iterative Deepening
Complete?	Yes (if $b < \infty$)	if $m < \infty$	Yes (if $b < \infty$)
Time	b^{d+1}	b^m	b^d
Space	b^{d+1}	bm	bd
Step-Optimal?	Yes	No	Yes

Loops: Repeated states

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





Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    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
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

But: storing all visited nodes leads again to exponential space complexity (as for BFS)

Graph search

- Slight modification of TreeSearch: lines 7 and 8

```
1:  procedure GRAPHSEARCH(s_0, S_goal, succ, cost)
2:      root ← Node(state=s_0, parent=nil, cost=0, depth=0)
3:      fringe ← ⟨root⟩
4:      closed ← {}
5:      while fringe is not empty do
6:          n ← fringe.pop()                          // pop node from fringe
7:          if n.state ∉ closed then
8:              closed.add( n.state )
9:              if(n.state∈ S_goal) return n          // goal check
10:             fringe.insert( EXPAND(n, succ, cost) )  // expand
11:         end if
12:         optional: if n.children={}, destroy it and also check ancestors
13:     end while
14:     return nil
15: end procedure
```

But: storing all visited nodes leads again to exponential space complexity (as for BFS)

Summary

- In BFS (or uniform-cost search), the fringe propagates layer-wise, containing nodes of similar distance-from-start (cost-so-far), leading to optimal paths but exponential space complexity $O(b^{d+1})$
- In DFS, the fringe is like a deep light beam sweeping over the tree, with space complexity $O(bm)$. Iteratively deepening it also leads to optimal paths.
- Graph search can be exponentially more efficient than tree search, but storing the visited nodes leads to exponential space complexity as BFS.

A* Search

Best-first Search (Prioritized Search)

- Use a priority $f(n)$ to sort nodes in the fringe
  - actually $f(n)$ is neg-priority: nodes with lower $f(n)$ have higher priority
- $f(n)$ should reflect which nodes *could* be on an optimal path
  - *could* is optimistic – the lower $f(n)$ the more optimistic you are that n is on an optimal path

  $\Rightarrow$ Pick the node with highest priority
- Implementation:
  - Plain TREESEARCH with fringe being a priority queue (increasing f-value, pop() returns first)

Special Cases

- Two quantities typically define the priority
  - The cost-so-far $g(n)$ = n.cost
  - A heuristic $h(n)$ which estimates cost-to-go

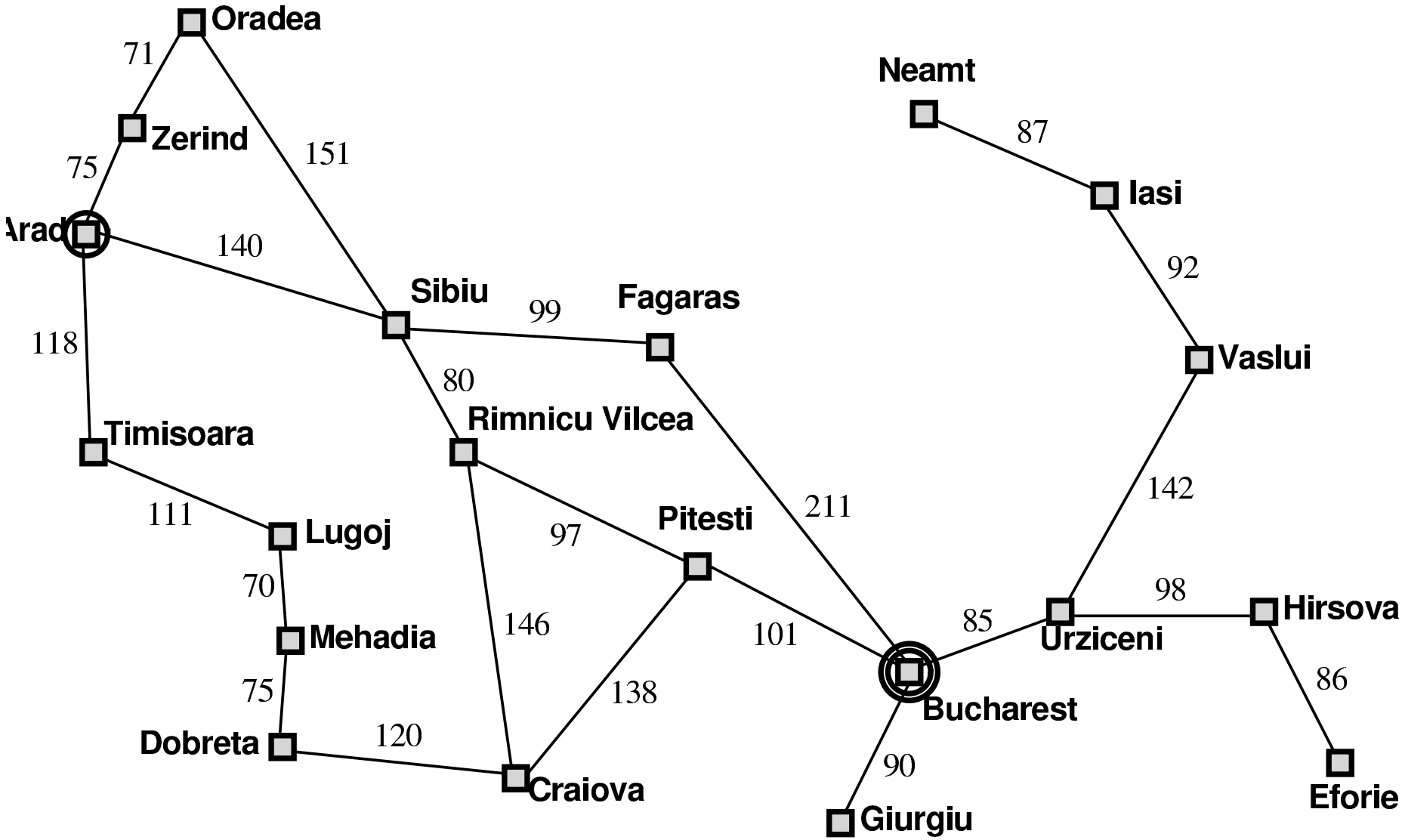
- Special cases of Best-First Search are:
  - uniform-cost search ($f = g$)
  - greedy search ($f = h$)
  - A* search ($f = g + h$)

  (Uniform-cost search is equal to A*when choosing a zero heuristic $h(n) = 0$)

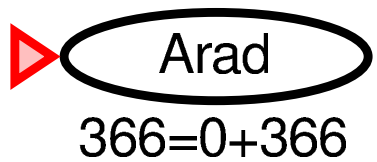
A* search

- *Combine information from the past and the future*
  - neg-priority = cost-so-far + estimated cost-to-go
- The evaluation function is $f(n) = g(n) + h(n)$, with
  - $g(n)$ = cost-so-far to reach n
  - $h(n)$ = estimated cost-to-go from n
  - $f(n)$ = estimated total cost of path through n to goal
- A* search uses an admissible heuristic. h admissible $\iff$
  - $h(n) \leq h^*(n)$, where $h^*(n)$ is the *true optimal* cost-to-go from n
  - $h(n) \geq 0$, and $h(G) = 0$ for any goal G
- E.g., $h_{\text{Euclidean-distance}}(n)$ never overestimates the actual road distance
- Theorem: A* search is optimal

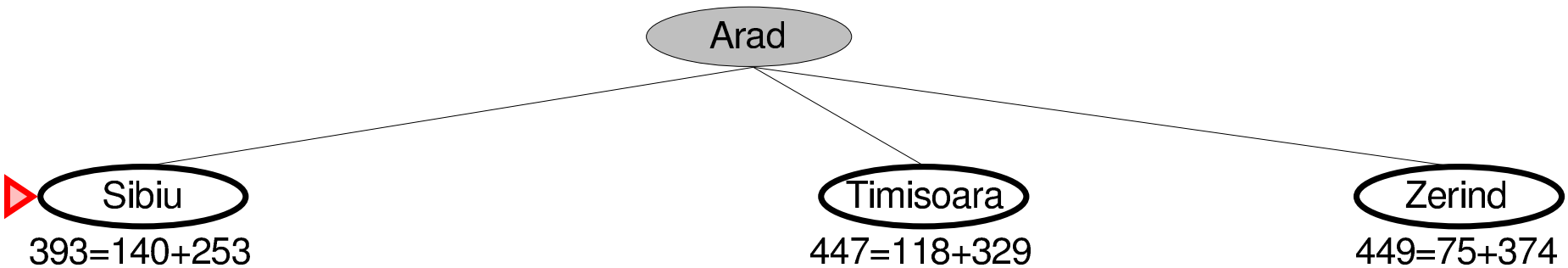
Example: Romania



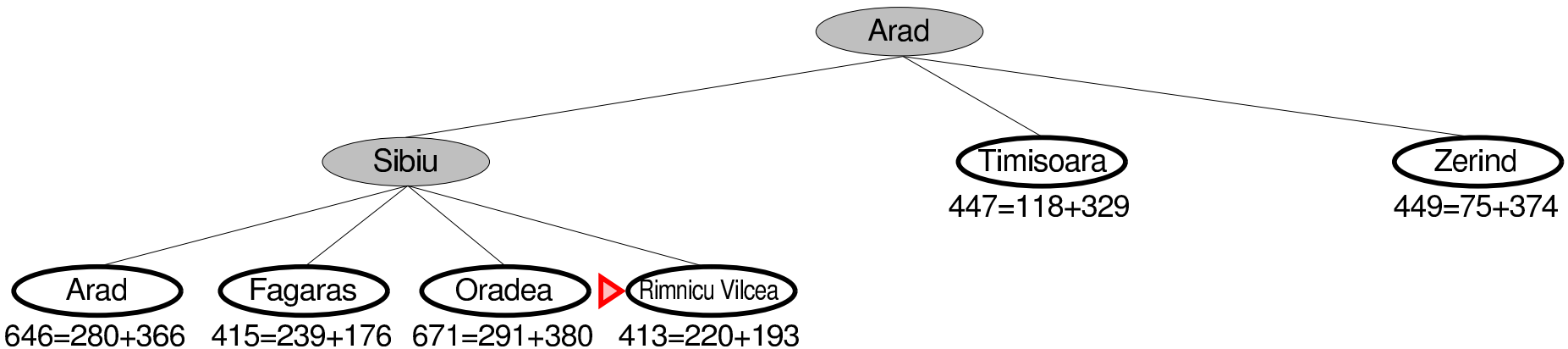
A* search example



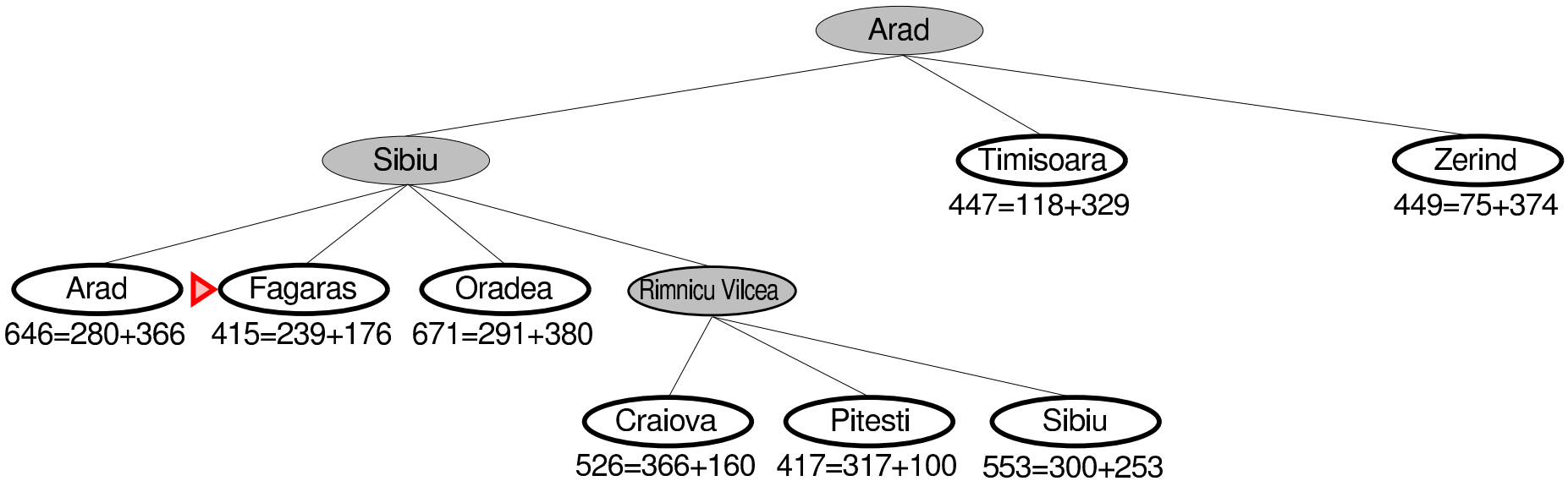
A* search example



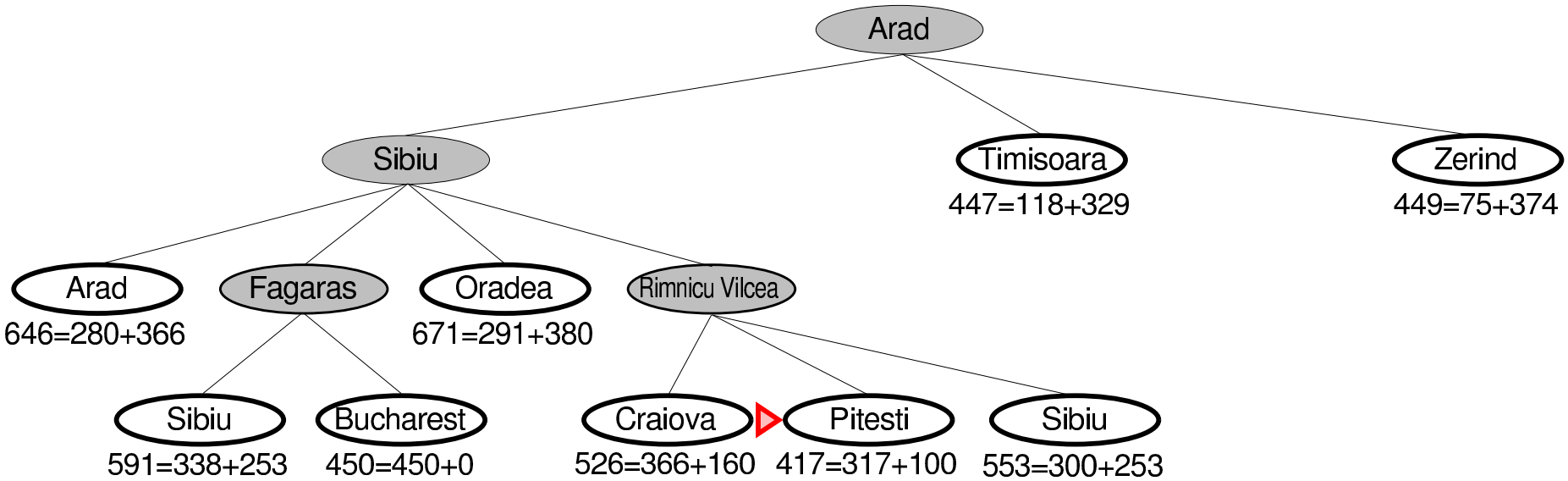
A* search example



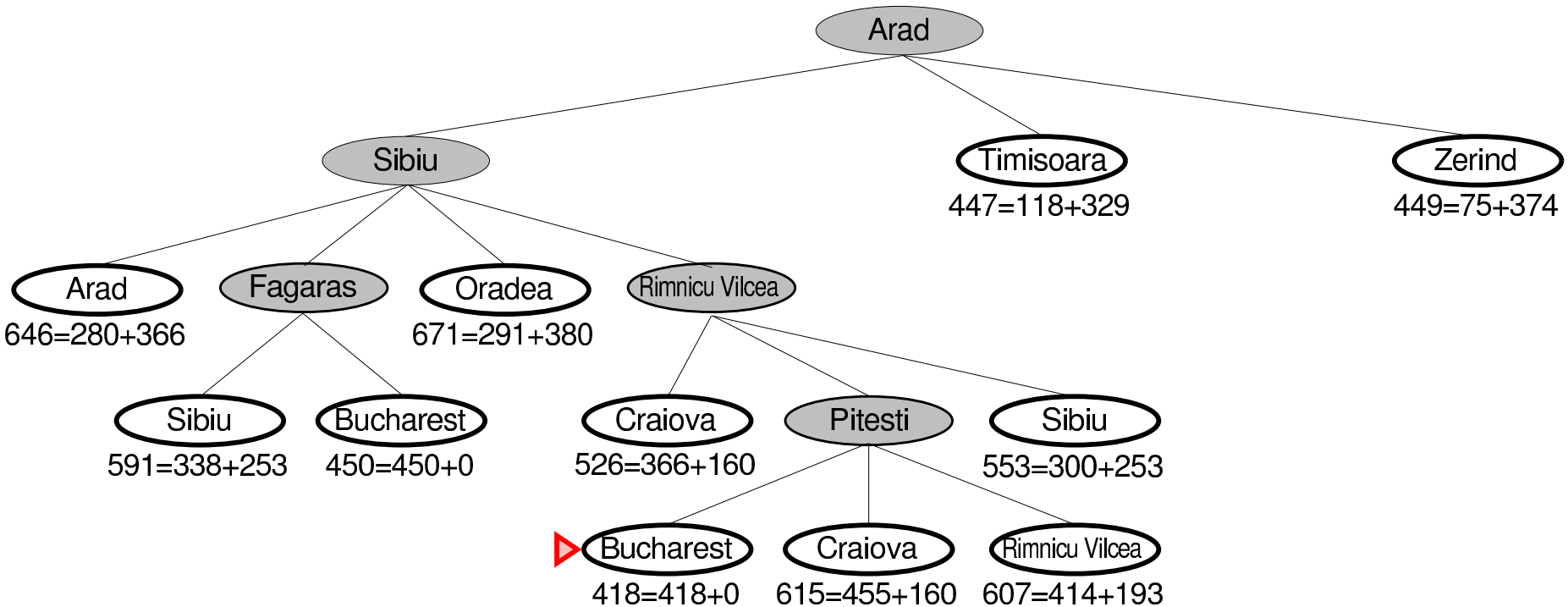
A* search example



A* search example

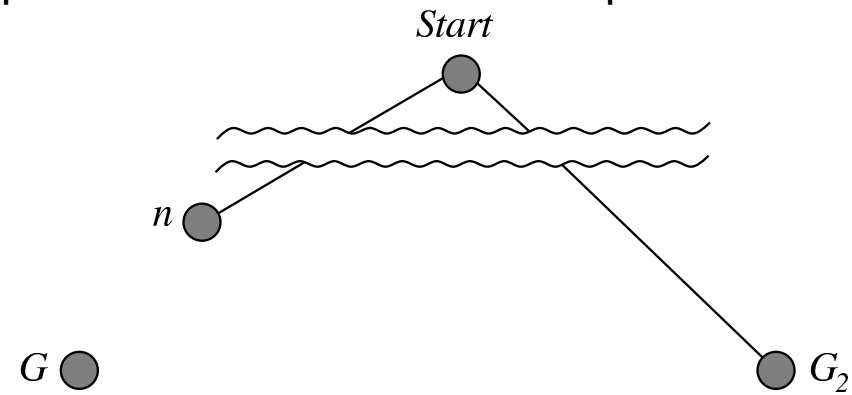


A* search example



Proof of optimality of A*

- Suppose some suboptimal goal G_2 has been generated and is in the fringe (but has not yet been pop'ed and goal tested!). We want to proof: *Any node on a shortest path to an optimal goal G will be expanded before G_2 is pop'ed.*
- Let n be an unexpanded node on a shortest path to G.



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

- Since $f(n) < f(G_2)$, A* will expand n before G_2 is pop'ed. This is true for any n on the shortest path. At some time G is added to the fringe, and since $f(G) = g(G) < f(G_2) = g(G_2)$ it will pop G before G_2.

Properties of A*

Complete??

Properties of A*

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

Time??

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??

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?? Exponential. Keeps all nodes in memory

Optimal??

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?? Exponential. Keeps all nodes in memory

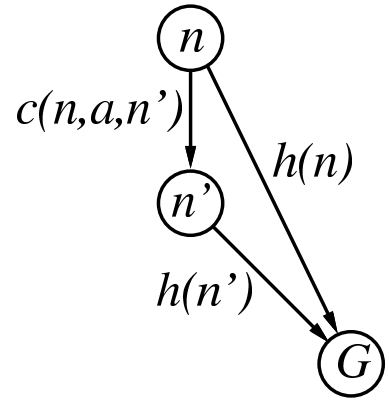
Optimal?? Yes

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

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

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

Consistent (or Monotone) Heuristics



- A heuristic is consistent (or monotone) iff

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

  - Most admissible heuristics are consitent; only slightly more strict

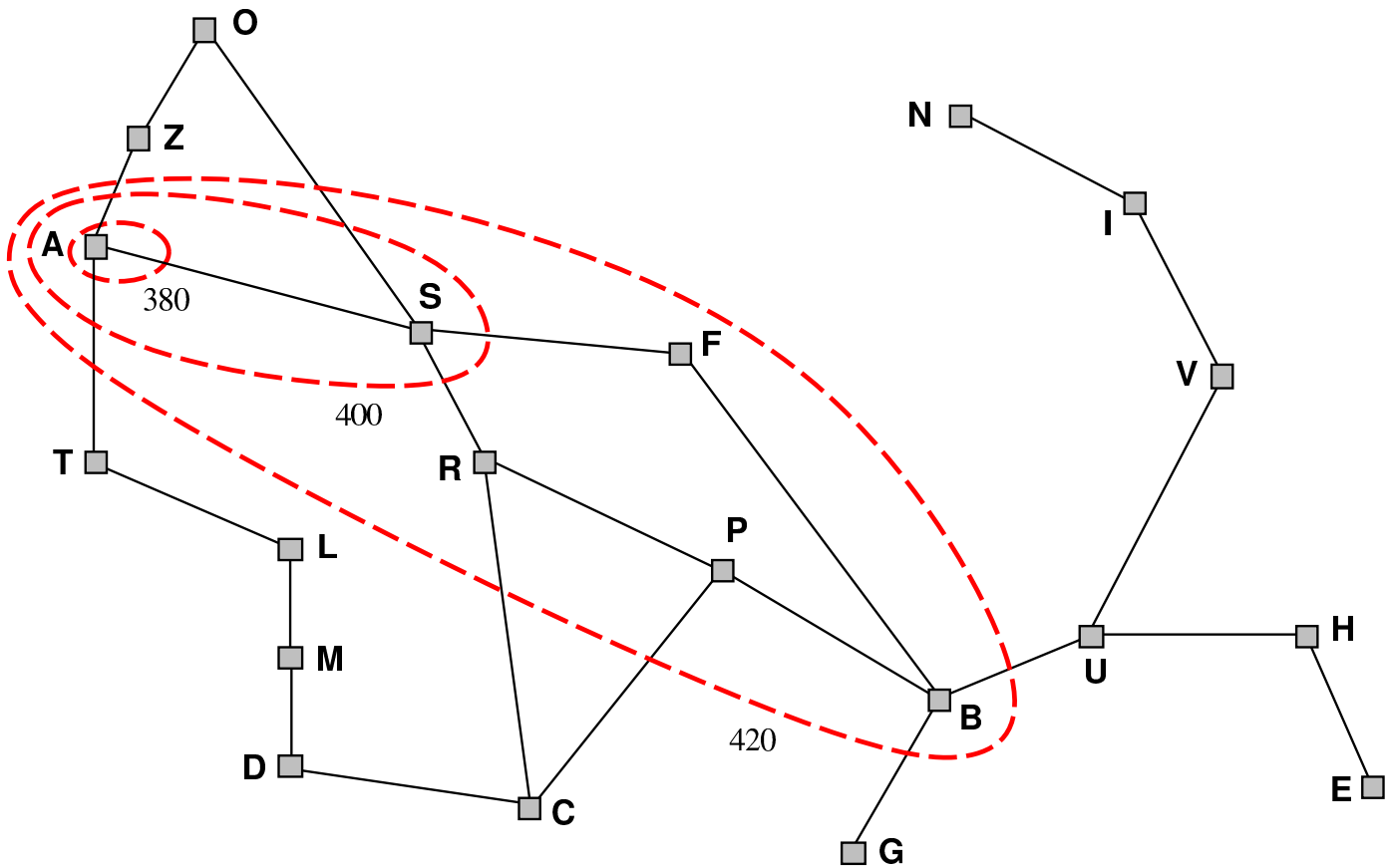
- If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path of the tree (or in the graph, if we do graph search.)

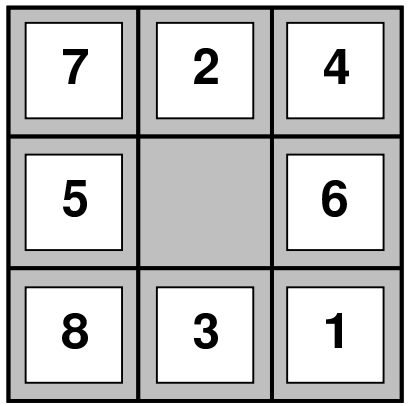
Optimality of A* with Consistent Heuristic

- 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 $f_i < f_{i+1}$

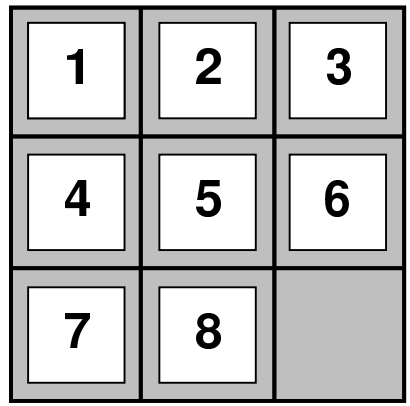


Admissible heuristics

- 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)



Start State

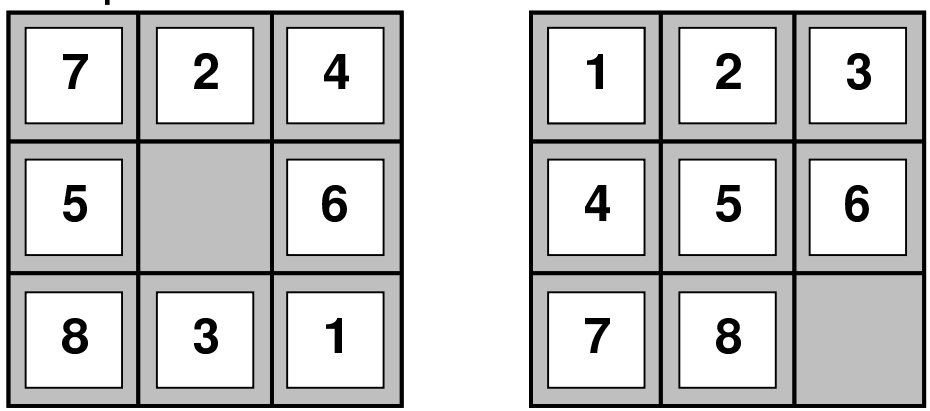


Goal State

$h_1(S)$ =??
$h_2(S)$ =??

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)



$h_1(S) =$?? 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

Heuristics in AI research

`https://ipc2018.bitbucket.io/`

`http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Results/ipc2011-booklet.pdf`

- Comments (as an outsider!)
  - The state-of-the-art planners are often just A* (or similar), but with crazily clever heuristics! The smarts is all in the heuristics!
  - State-of-the-art heuristics automatically analyze the problem description for structure, such as landmarks. Based on this structure, they define heuristics.

Branch & Bound

- In A*, $f(n)$ is a *lower* bound of the *partial solution* represented by n
- In Branch & Bound we also want to have an upper bound. If that bound is higher than the lowest full solution cost so far, we don't expand anymore
- Otherwise it is essentially A*

Memory-bounded A*

- As with BFS, A* has exponential space complexity
- Iterative-deepening A*, works for integer path costs, but problematic for real-valued
- (Simplified) Memory-bounded A* (SMA*):
  - Expand as usual until a memory bound is reach
  - Then, whenever adding a node, remove the *worst* node n' from the tree
  - worst means: the n' with highest $f(n')$
  - To not loose information, *backup* the measured step-cost $cost(\tilde{n}, a, n')$ to improve the heuristic $h(\tilde{n})$ of its parent

  SMA* is complete and optimal if the depth of the optimal path is within the memory bound

Outlook

- Tree search with *partial observations*
  - we discuss this in a fully probabilistic setting later
- Tree search for *games*
  - minimax extension to tree search
  - probabilistic Monte-Carlo tree search methods for games