Lecture 2b: Informed Search

CSCI 360

Introduction to Artificial Intelligence

USC

What have we learned so far?

- What is AI?
- Problem-solving agent
- Uninformed search
- Informed search

What have we learned so far?

- **What is AI?**
  - Acting rationally
    - a function that maps percepts to actions
- Problem-solving agent
- Uninformed search
- Informed search

What have we learned so far?

- What is AI?
- **Problem-solving agent**
  - Formulating the problem
    - States, initial state, actions, transition model, goal test
- Uninformed search
- Informed search

What have we learned so far?

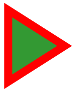
- What is AI?
- Problem-solving agent
- **Uninformed search**
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Informed search

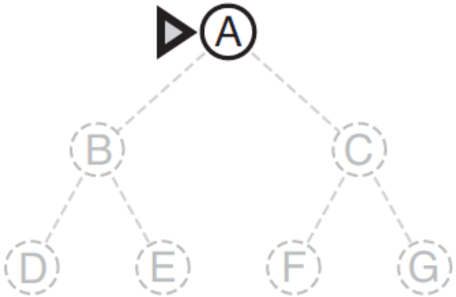
Recap: Breadth-first search (BFS)

- The “**shallowest**” node in frontier is chosen for expansion
  - The frontier is implemented using a FIFO queue

Recap: Breadth-first search (algorithm)

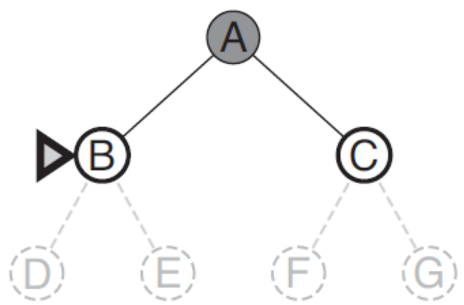
At each step, the node to be expanded next is indicated by the marker



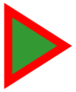


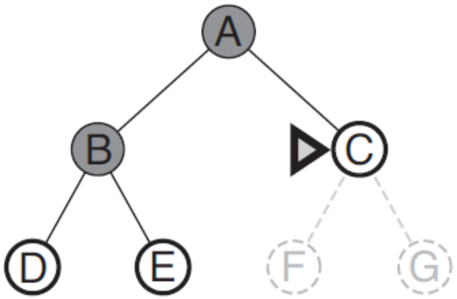
Recap: Breadth-first search (algorithm)

At each step, the node to be expanded next is indicated by the marker



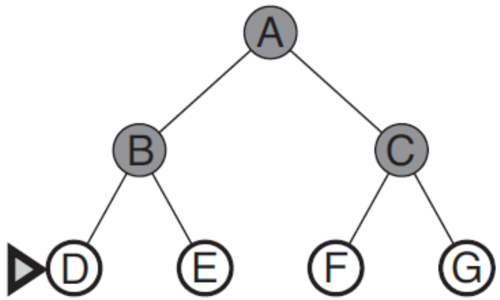
Recap: Breadth-first search (algorithm)

At each step, the node to be expanded next is indicated by the marker 



Recap: Breadth-first search (algorithm)

At each step, the node to be expanded next is indicated by the marker



Recap: BFS (performance)

- Complete?
  - Yes -- If the shallowest goal node is at some finite depth, ***d***, then BFS will eventually find the goal node

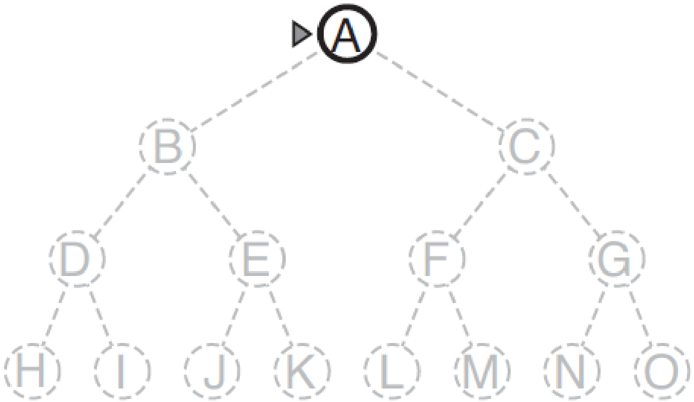
- Optimal?
  - Yes -- If the path cost is a non-decreasing function

- Time complexity?
  - $b + b^2 + b^3 + \ldots + b^d = \boldsymbol{O(b^d)}$

- Space complexity?
  - $\boldsymbol{O(b^d)}$

Recap: Depth-first search (DFS)

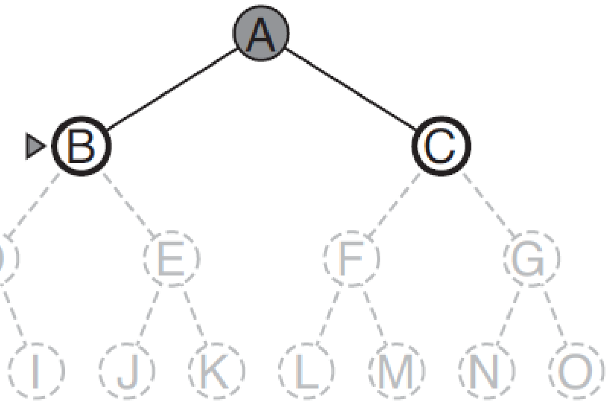
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)



-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

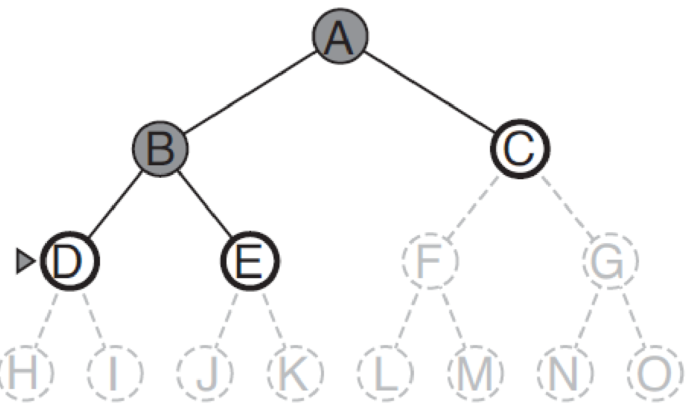
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

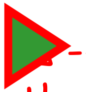


-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

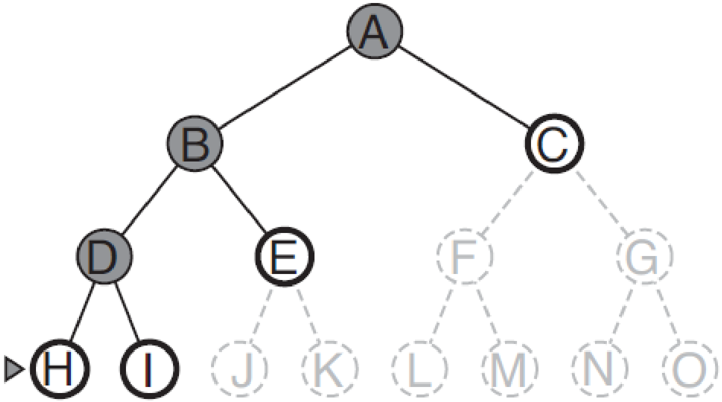
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)




-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

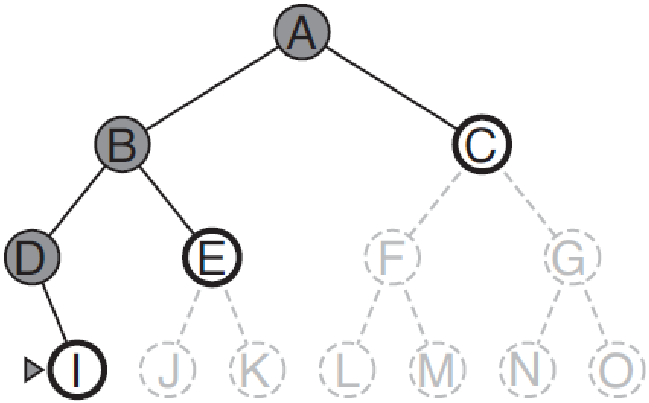
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

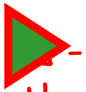


-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

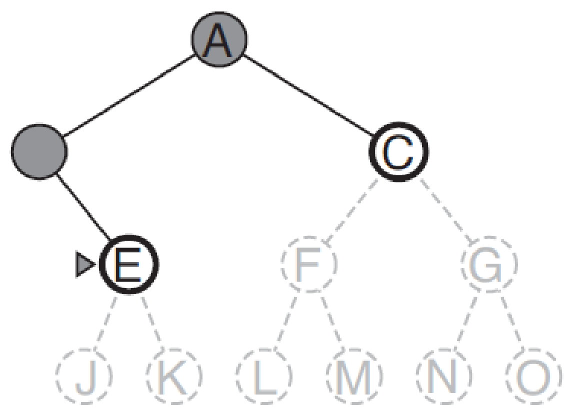
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)




-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

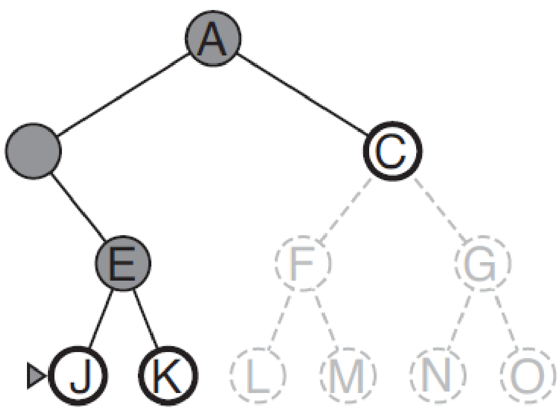
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

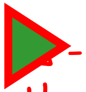


-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

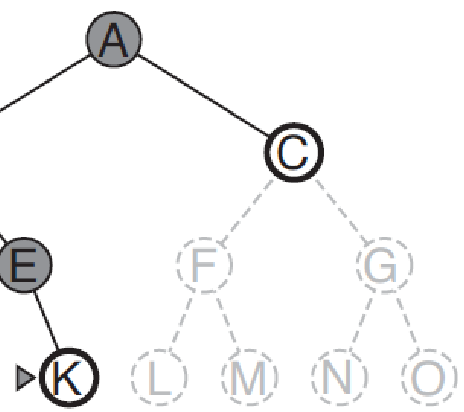
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)




-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

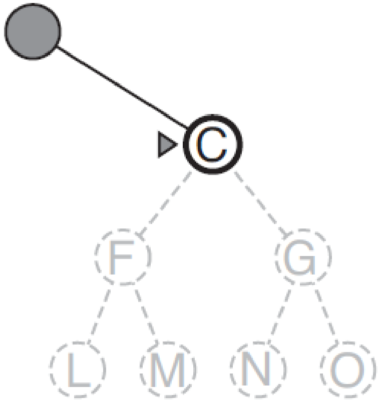
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

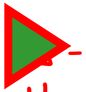


-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

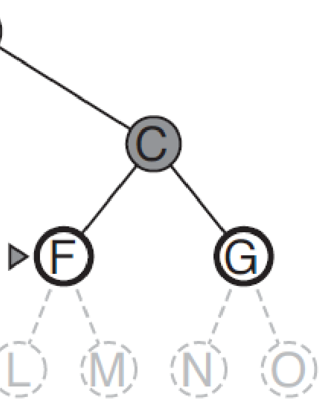


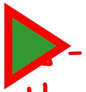


-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

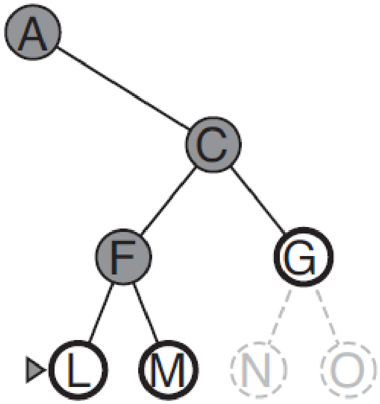
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

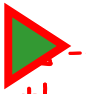



-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

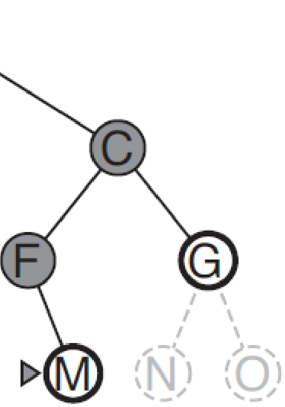
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)




-- node to be expanded next is indicated by the marker

Recap: Depth-first search (DFS)

- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)



-- node to be expanded next is indicated by the marker

Recap: DFS (performance)

- Complete?
  - No -- If the shallowest goal node is at some finite depth, ***d***, and the state space is infinite, then DFS may never find the goal node

- Optimal?
  - No

- Time complexity?
  - $b + b^2 + b^3 + \ldots + b^m = \boldsymbol{O(b^m)}$

- Space complexity?
  - ***O(bm)***

Recap: DFS's space complexity

- Depth-first tree search is the "**basic workhorse**" of many areas of AI because of its memory efficiency

- Require storage of only *O(bm)* nodes

Depth	Nodes	Time	Memory	DFS
2	110	.11 milliseconds	107 kilobytes	
4	11,110	11 milliseconds	10.6 megabytes	
6	10^6	1.1 seconds	1 gigabyte	
8	10^8	2 minutes	103 gigabytes	
10	10^{10}	3 hours	10 terabytes	
12	10^{12}	13 days	1 petabyte	
14	10^{14}	3.5 years	99 petabytes	
16	10^{16}	350 years	10 exabytes	156 kilobytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Recap: Having test of Both Worlds (DFS+BFS)?

- Iterative deepening DFS

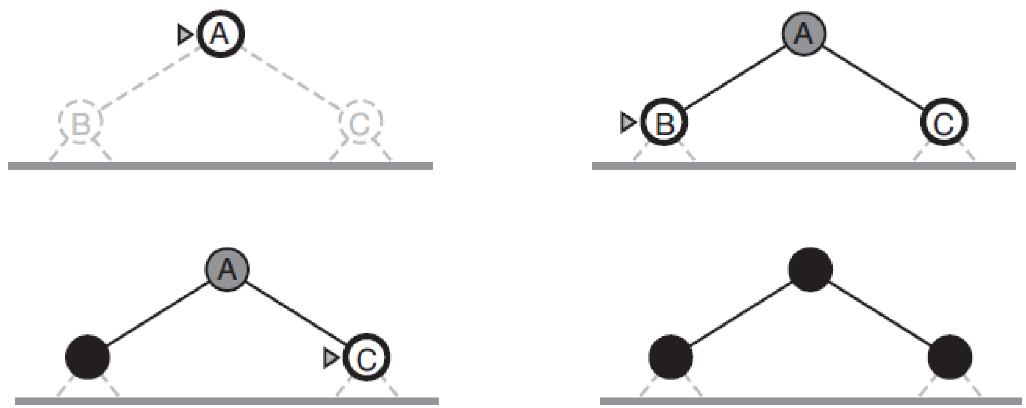
function ITERATIVE-DEEPENING-SEARCH($problem$) **returns** a solution, or failure
  for $depth = 0$ **to** ∞ **do**
    $result \leftarrow$ DEPTH-LIMITED-SEARCH($problem$, $depth$)
    if $result \neq$ cutoff **then return** $result$

- Advantages
  - **Complete** and **Optimal** (just like BFS)
  - **O(bd)** storage requirement (just like DFS)

Recap: Iterative Deepening (example)

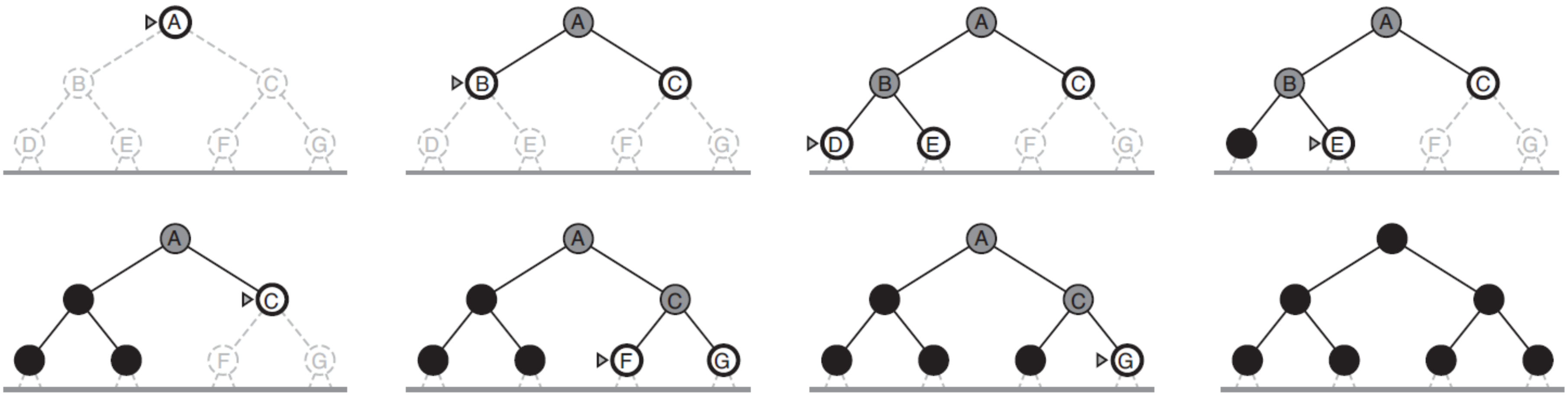
- Limit = 0

- Limit = 1



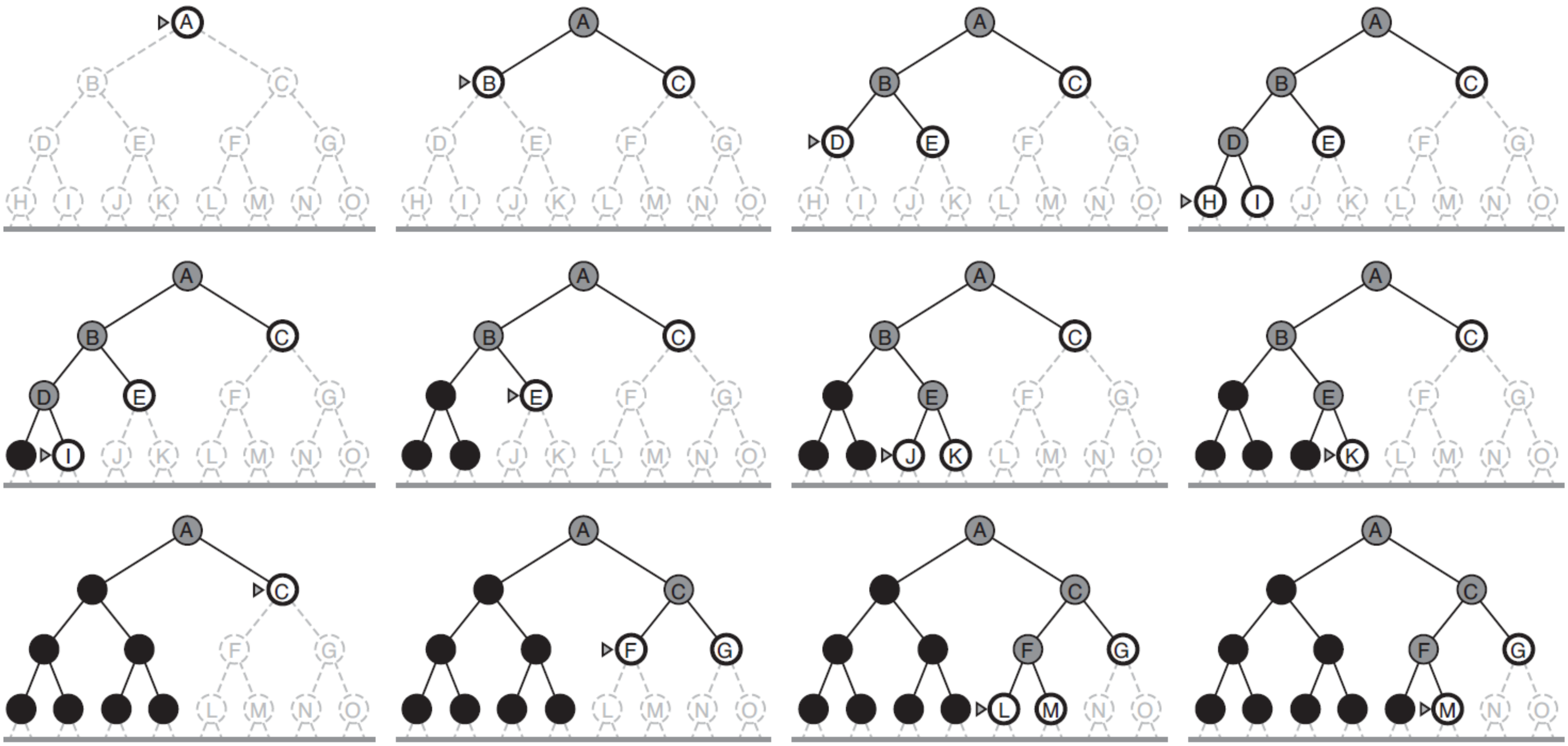
Recap: Iterative Deepening (example)

- Limit = 2



Recap: Iterative Deepening (example)

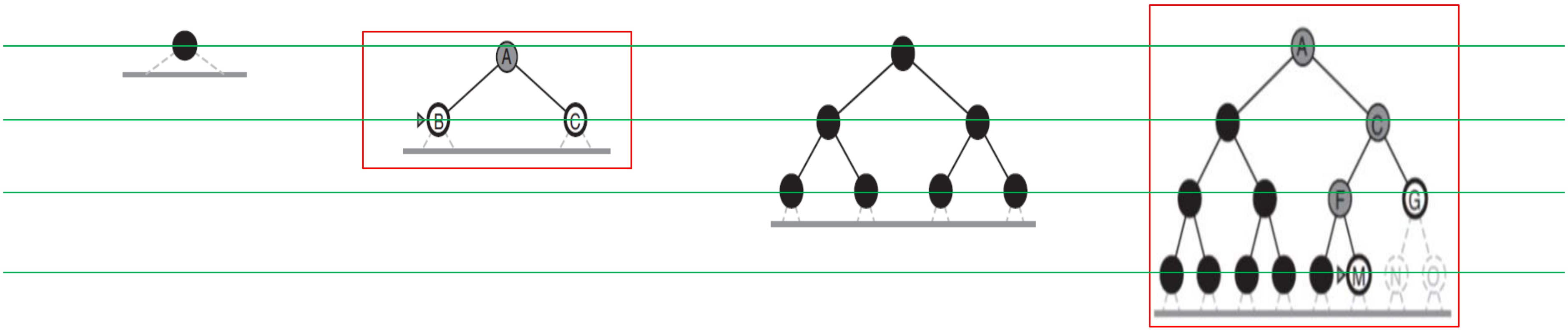
- Limit = 3



Recap: Iterative Deepening (Time Complexity)

- Except for (*limit = d*), each level is visited multiple times

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$



Recap: Iterative Deepening (Time Complexity)

- Except for (*limit = d*), each level is visited multiple times

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

- Compare to BFS with shallowest goal depth (d) → *$O(b^d)$*

- ***Example:*** *Let (b=10) and (d=5)*

$$\boxed{\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 \end{aligned}}$$

Recap: Iterative Deepening (conclusion)

- **Iterative deepening** is the preferred uninformed search method for many AI applications
  - E.g., when (1) the search space is large and (2) the depth of the solution is not known

Comparison of uninformed search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Outline for Today

- What is AI?
- Problem-solving agent
- Uninformed search
- **Informed search**
  - Greedy best-first search
  - A* search
  - Memory-bounded heuristic search
  - Heuristic functions

Uninformed vs. Informed Search

- **Uninformed search** (or blind search): the agent does not know whether one state is "more promising"
  - All the agent can do is to (1) generate successor states and
  - (2) check if a state is a goal state

- **Informed Search**: the agent ***has some idea*** whether one state is "more promising" than another state

Best-First Search

- Node is chosen for expansion based on an **evaluation function, *f(n)***
  - Returns an estimated, total solution cost at node ***n***

- Pseudocode of “best-first search” is identical to “uniform-cost search” except that
  - **Uniform-cost**: priority queue ordered by actual path cost ***g(n)***
  - **Best-first**: priority queue ordered by combined ***f(n) = g(n) + h(n)***

What is ***h(n)***?

Heuristic Function *h(n)*

- The estimated cost of the cheapest path from the state at node, n, to a goal state

total cost

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

– The **actual path cost** from the initial state to the state at node n

The **estimated cost** from the state at node n to a goal state

Heuristic Function *h(n)*

- The estimated cost of the cheapest path from the state at node, n, to a goal state

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

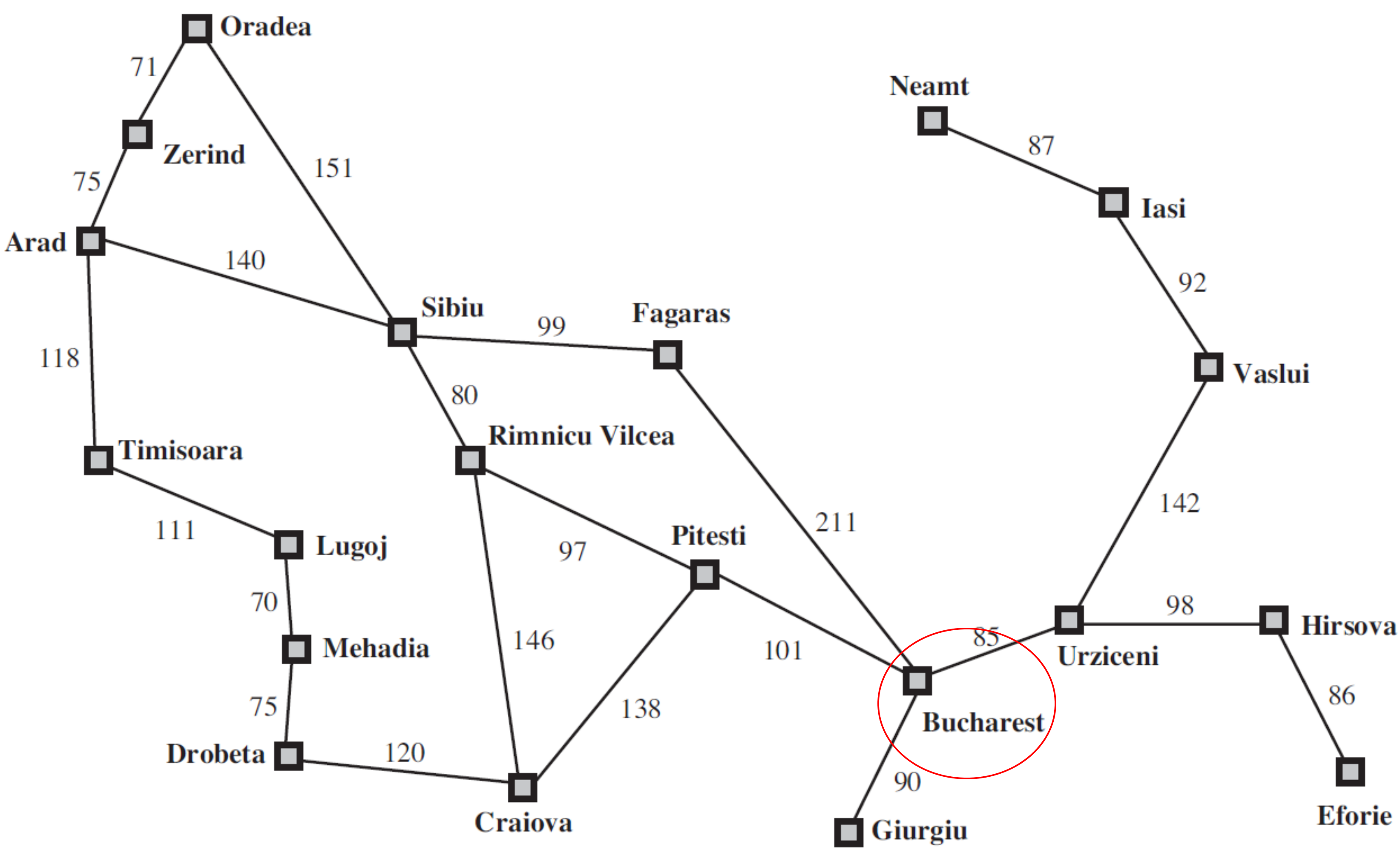
- Heuristic functions are not derived from the **problem description**, but from "**additional knowledge**"

Greedy Best-first Search

- **Intuition:** Expands node that "appears" to be **closest to the goal** since it's likely to lead a solution quickly
  - i.e., $f(n) = h(n)$
  - In other words, assume the past cost $g(n) = 0$

- **Example:** In Map of Romania, if the goal is Bucharest, we can use the "*straight-line distance to Bucharest*" as the heuristic function *h(n)*

Question: *How to compute “straight-line distances”?*



Correct answer: **You Can't!**

Question: *How to compute the "straight-line distances"?*

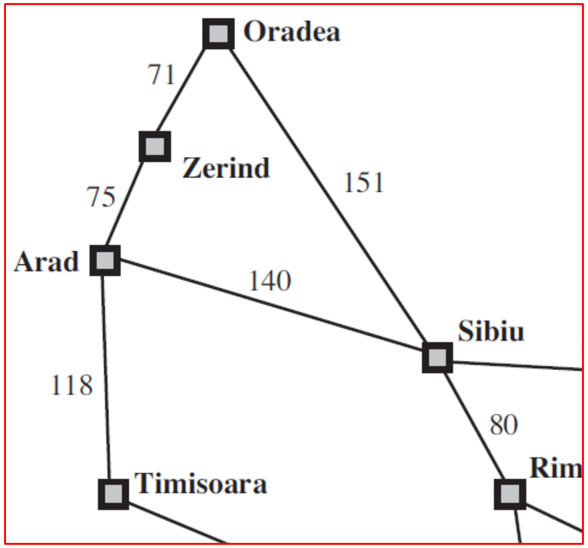
The values of ***h(n)*** cannot be computed from the problem description itself; it has to come from "**additional knowledge**"

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Greedy Best-first Search (example run)

- Starting at “Arad”, there are three actions
  - *Go(Zerind)*
  - *Go(Sibiu)*
  - *Go(Timisoara)*

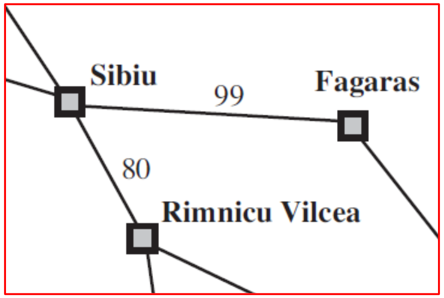


- After that, which of the new nodes should be chosen for expansion?

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best-first Search (example run)

- Starting at “Sibiu”, there are two actions
  - *Go(Fagaras)*
  - *Go(Rimnicu Vilcea)*



- After that, which of these two new nodes should be expanded next?

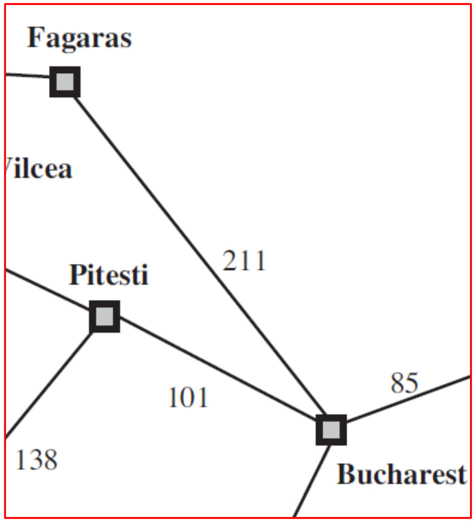
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best-first Search (example run)

- Starting at “Fagaras”, there is only one action
  - *Go(Bucharest)*

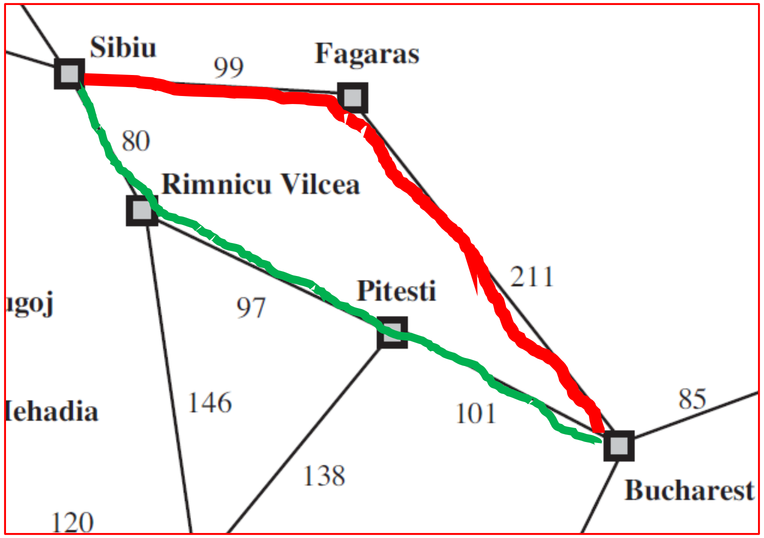
- It’s a goal state!

- But, is this the **optimal** solution?



Greedy Best-first Search (optimality)

- Starting at “Fagaras”, there is only one action
  - *Go(Bucharest)*
- It’s a goal state!
- No, it is **not optimal**

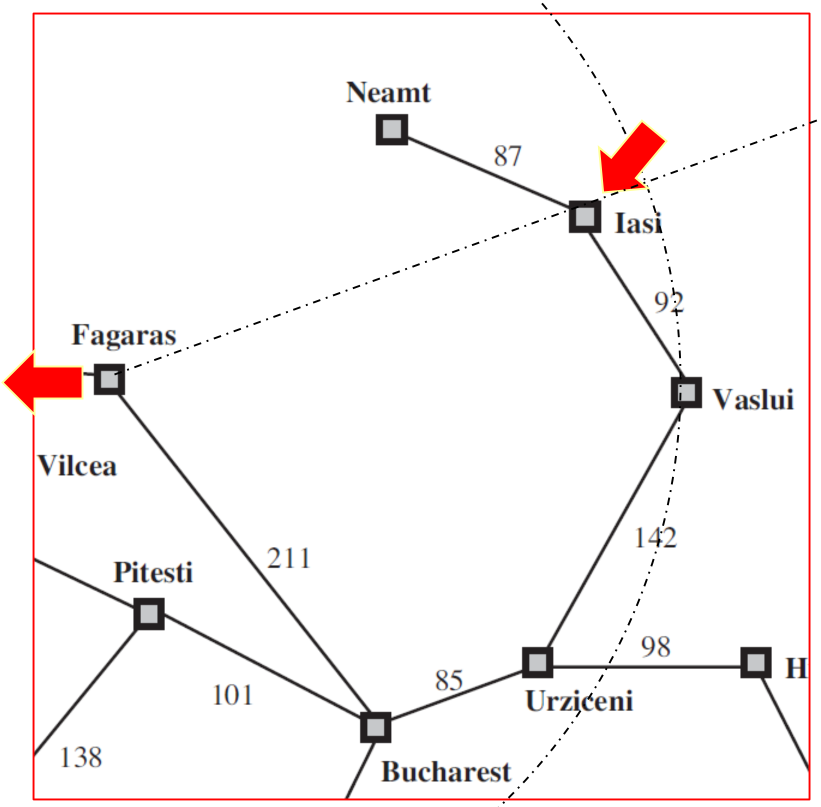


278 kilometers vs. 310 kilometers

Greedy Best-first Search (completeness)

- Tree-Search is not complete even in a finite state space

- Example:
  - From “Iasi” to “Fagaras”
    - *Go(Neamt)*
    - *Go(Vaslui)*
  - **Which node to expand next?**
    - ***Neamt***, since its “straight-line distance” to *Fagaras* is shorter
    - Will never find the solution due to the infinite loop “Neamt – Iasi”



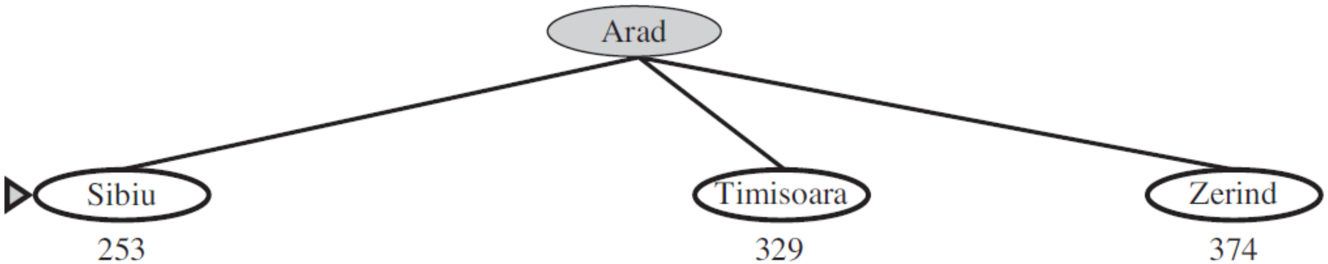
Graph-Search is complete, but only in a finite state space

Greedy Best-first Search (example run)

- When it works, it’s very fast due to “pruning” of state space

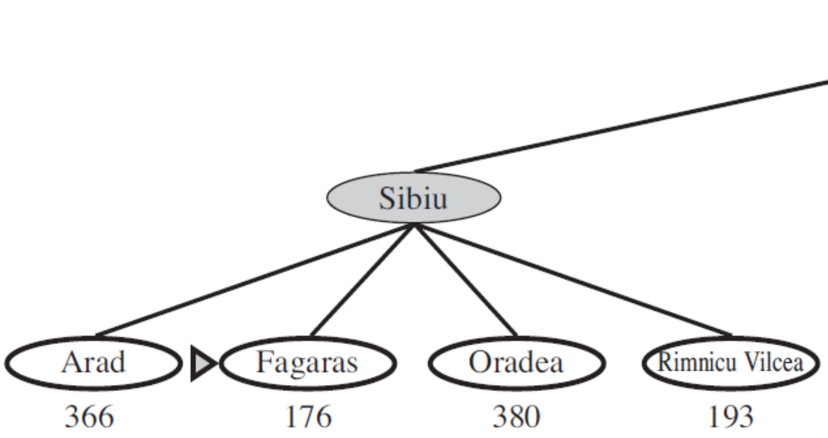
Greedy Best-first Search (example run)

- When it works, it's very fast due to "pruning" of state space



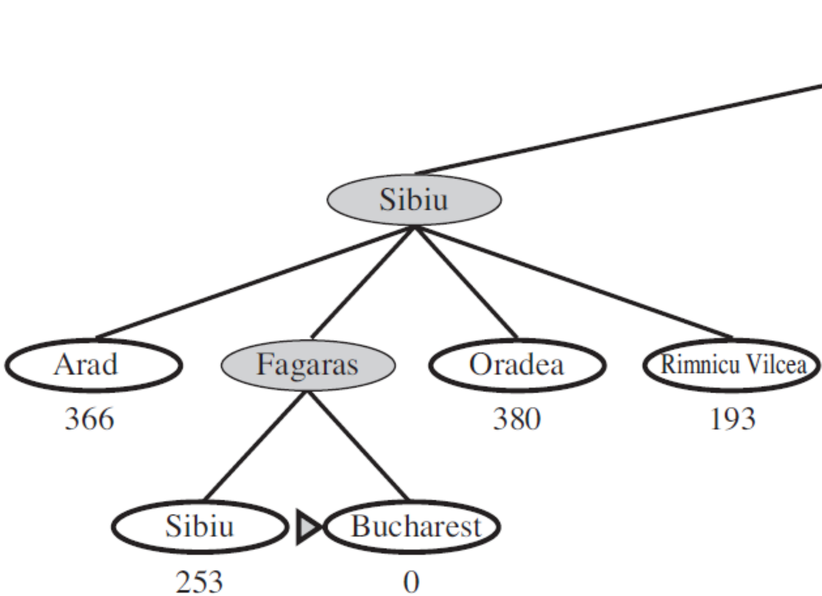
Greedy Best-first Search (example run)

- When it works, it’s very fast due to “pruning” of state space



Greedy Best-first Search (example run)

- When it works, it's very fast due to 'pruning' of state space



Only **3 nodes** are expanded (Arad, Sibiu, Fagaras)

Timisoara is never expanded

Summary of Greedy Best-first Search

- Complete?
  - No – can get stuck in loops
- Optimal?
  - No
- Time?
  - Exponential, but a way to get dramatic improvement
- Space?
  - Keeps all nodes in memory

Outline for Today

- What is AI?
- Problem-solving agent
- Uninformed search
- **Informed search**
  - Greedy best-first search
  - **A* search**
  - Memory-bounded heuristic search
  - Heuristic functions

Question: How to make “best-first search” optimal and complete?

- **Answer:** Use both ***g(n)*** and ***h(n)*** to compute total cost ***f(n)***

total cost

$$\boxed{f(n) = g(n) + h(n)}$$

- The **actual path cost** from the initial state to the state at node n

The **estimated cost** from the state at node n to a goal state

A* search – minimizing the total estimated cost

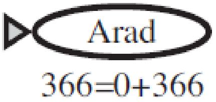
- Most widely known form of “best-first search”
  - **Goal:** find the cheapest solution
  - **At each step**: expand node with lowest value of the total estimated solution cost: f(n) = g(n)+h(n)

- Pseudo code is identical to “Uniform-cost Search” except for using ***(g+h)*** instead of ***(g)*** to set the priority queue

Key idea: *Avoid expanding along paths that are already known to be expensive*

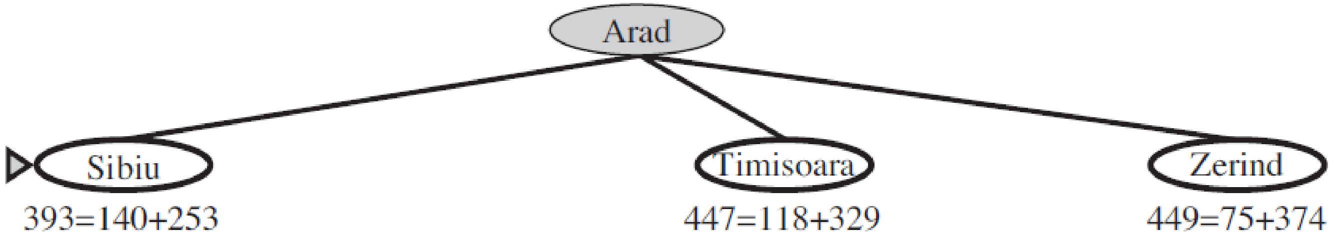
A* search (example run)

- Total estimated solution cost: *f(n) = g(n) + h(n)*



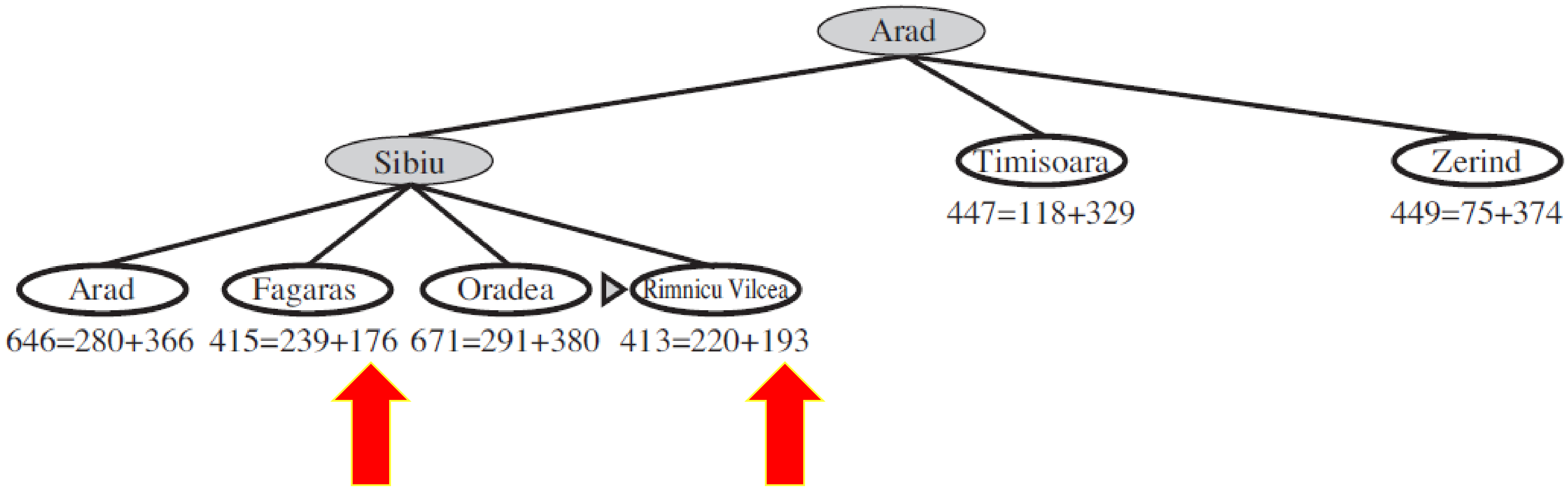
A* search (example run)

- Total estimated solution cost: *f(n) = g(n) + h(n)*



A* search (example run)

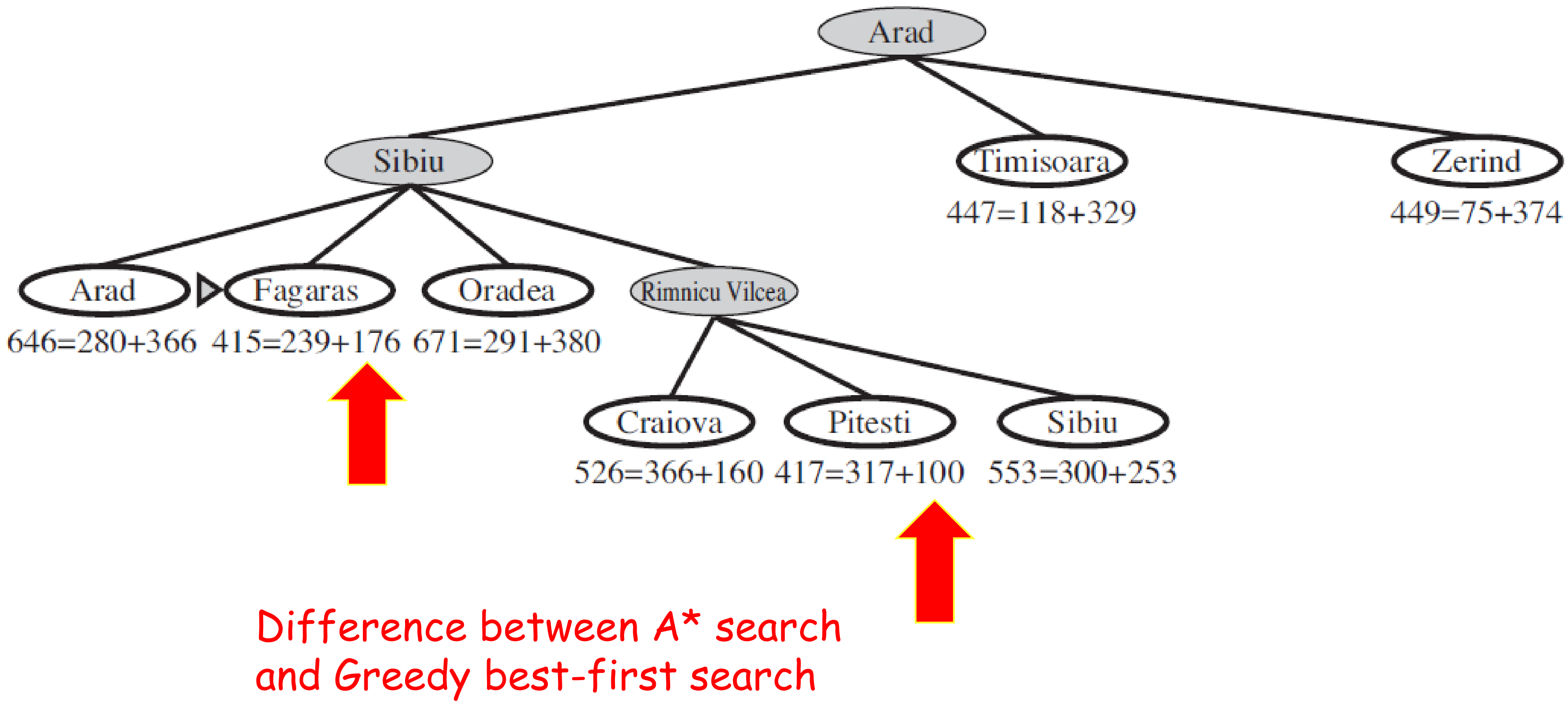
- Total estimated solution cost: *f(n) = g(n) + h(n)*



Difference between A* search
and Greedy best-first search

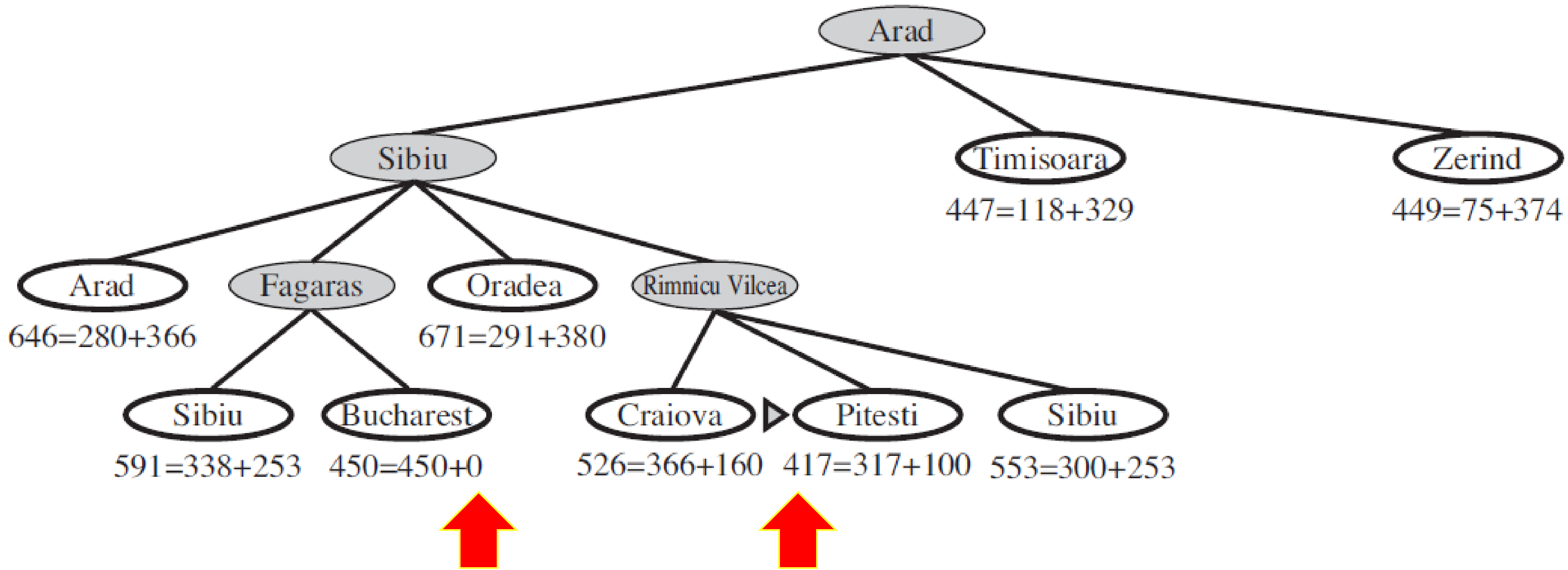
A* search (example run)

- Total estimated solution cost: $f(n) = g(n) + h(n)$



A* search (example run)

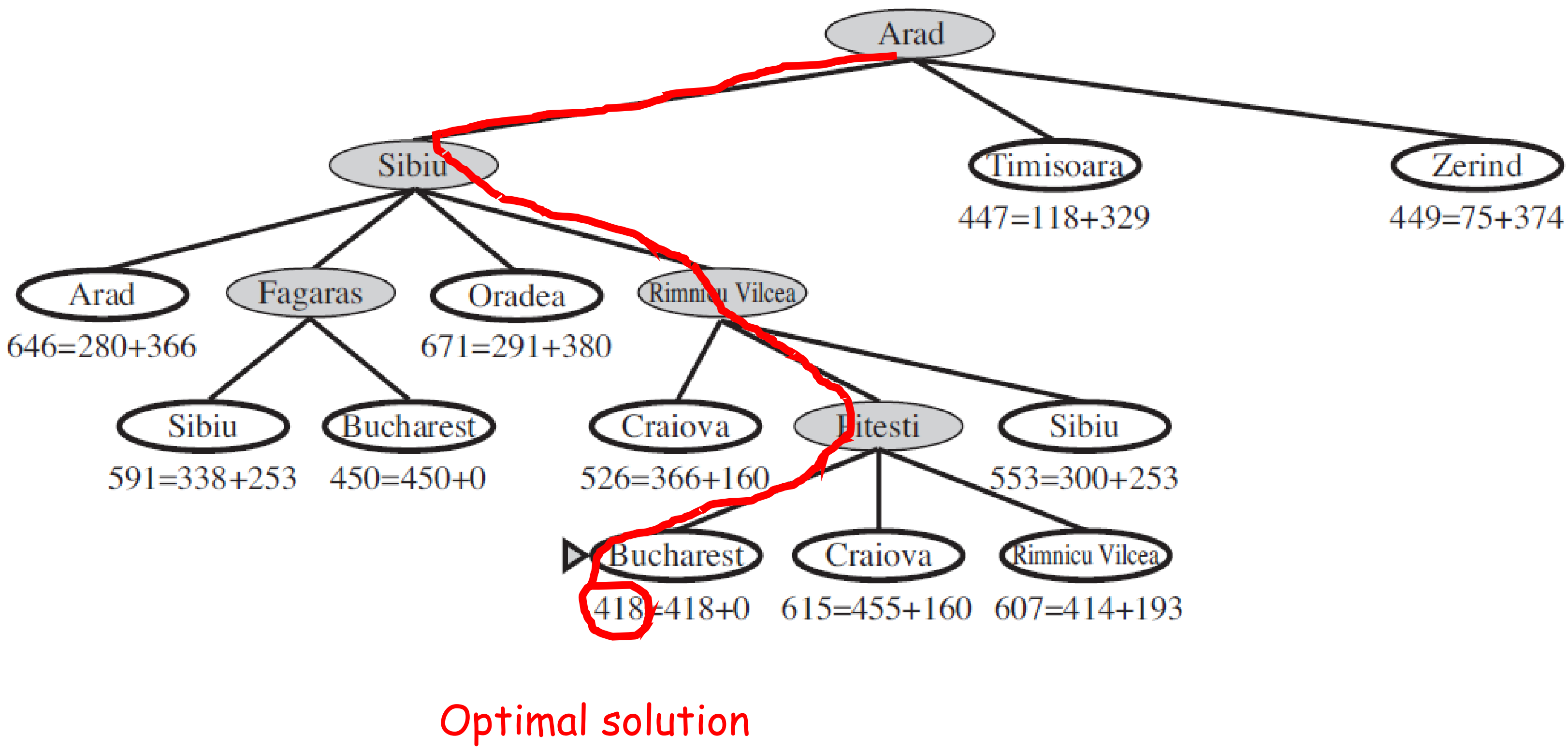
- Total estimated solution cost: *f(n) = g(n) + h(n)*



Greedy best-first search would have stopped at this point

A* search (example run)

- Total estimated solution cost: *f(n) = g(n) + h(n)*



Pseudo code (same as Uniform-cost Search)

- Node with the "**lowest path cost**" ➔ node with the "**lowest total cost**"

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Is A* complete and optimal?

- Complete
  - If there is a solution, A* search will find it

- Optimal
  - The solution found by A* search is guaranteed to be the cheapest solution

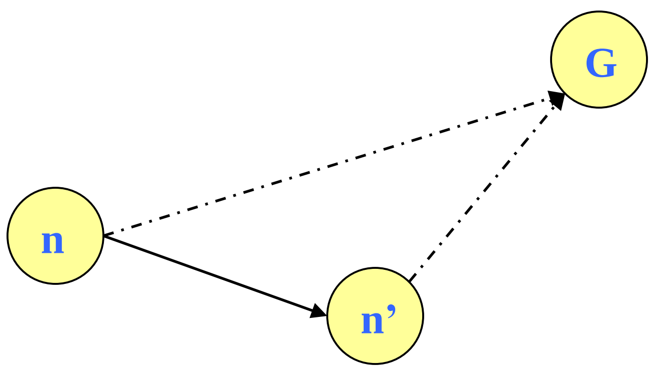
Condition for optimality

- Heuristic function ***h(n)*** is **admissible**
- An admissible heuristic is one that *never overestimates* the true cost to reach the goal
  - Since ***f(n) = g(n) + h(n)*** and
  - ***g(n)*** is the actual cost,
  - If ***h(n)*** never overestimates the cost from ***n*** to the goal, then ***f(n)*** never overestimates the total cost of a solution along the current path through node ***n*** either

Stronger condition for optimality

- **Consistency** (or **monotonicity**) is required only for applications of A* to ***graph search***

- Heuristic function ***h(n)*** is consistent if, for any successor ***n'*** of ***n*** generated by any action ***a***, we have
  - $h(n) \le c(n,a,n') + h(n')$



Optimality of A*

- The graph-search version of A* search is optimal if ***h(n)*** is consistent.

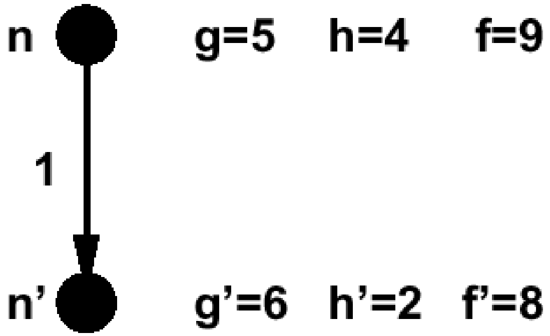
- (Less useful) proof sketch:
  - First, if h(n) is consistent, the values of f(n) along any path are non-decreasing.
    - By the definition of "consistency"
  - Second, whenever A* selects a node (n) for expansion, the optimal path to that node has been found
    - Assume this is not true, there would have to be another frontier node (n') on the optimal path from the start node to n; since f(n) is non-decreasing, f(n') would have a lower value than f(n), which means n' would have been selected for expansion

Reminder: *What's the meaning of **f(n)**?*

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

(not consistent)

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

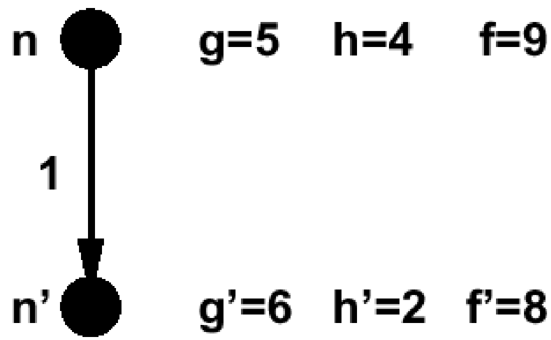


Reminder: *What's the meaning of **f(n)**?*

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

(not consistent)

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



But this throws away information!

$f(n) = 9 \Rightarrow$ true cost of a path through n is ≥ 9

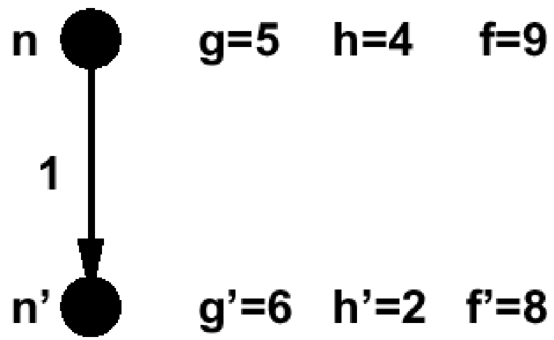
Hence true cost of a path through n' is ≥ 9 also

Reminder: *What's the meaning of* ***f(n)***?

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

(not consistent)

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



But this throws away information!

$f(n) = 9 \Rightarrow$ true cost of a path through n is ≥ 9

Hence true cost of a path through n' is ≥ 9 also

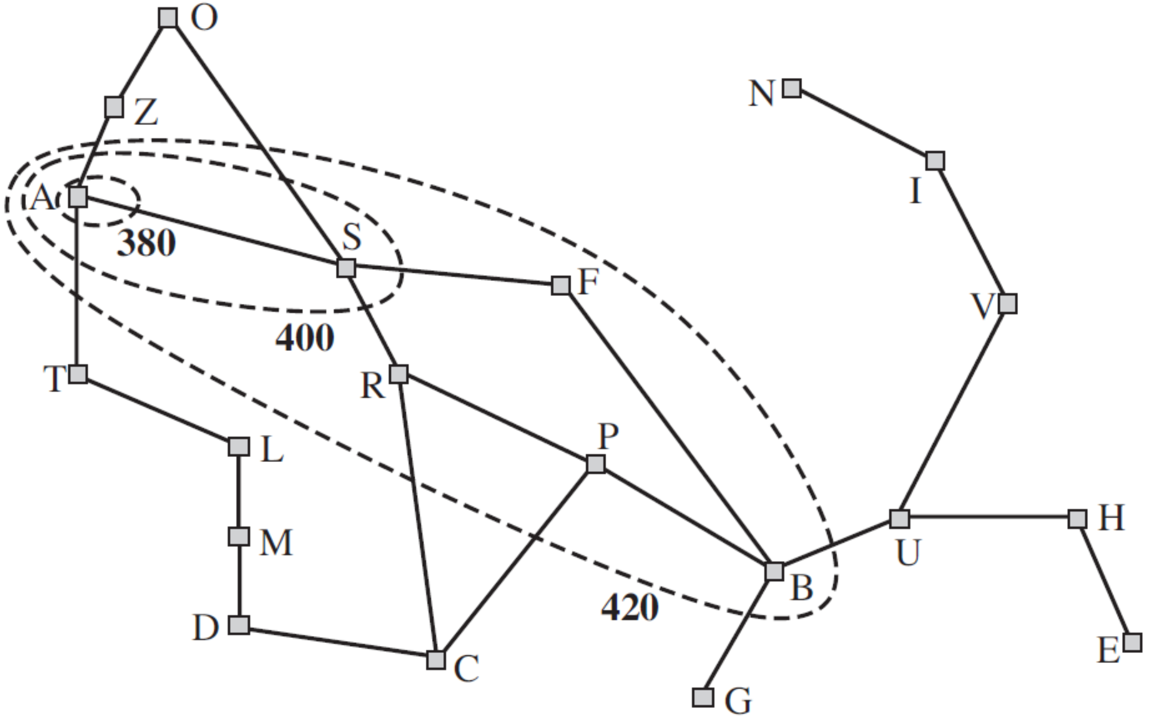
Pathmax modification to A*:

Instead of $f(n') = g(n') + h(n')$, use $f(n') = max(g(n') + h(n'), f(n))$

With pathmax, f is always nondecreasing along any path

Contours in A* search

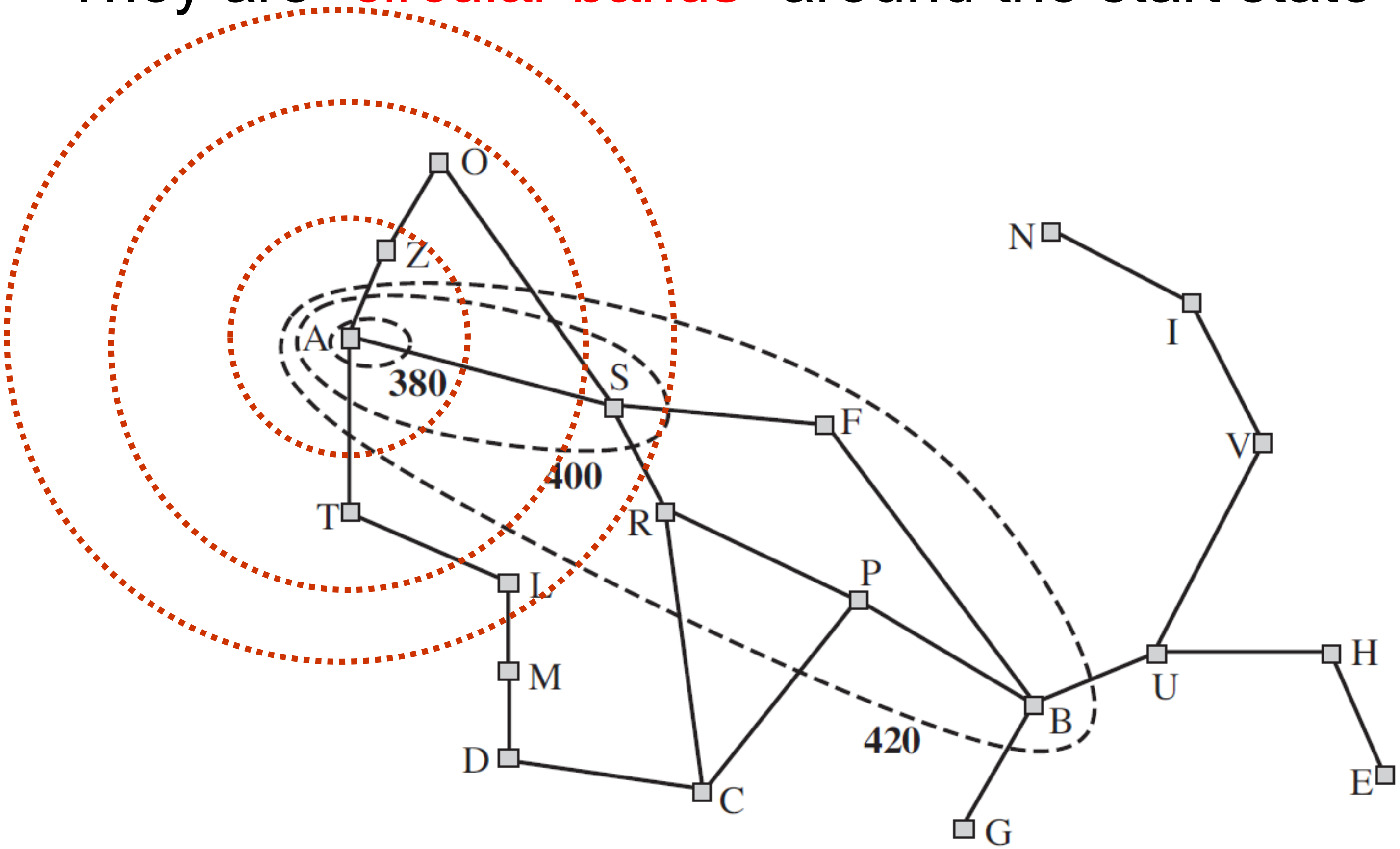
- The values of **f(n)** are non-decreasing along any path from the initial state – they form “contours”



Question: Are there contours in “uniform-cost search”? What do they look like?

Contours in Uniform-cost Search

- They are "circular bands" around the start state

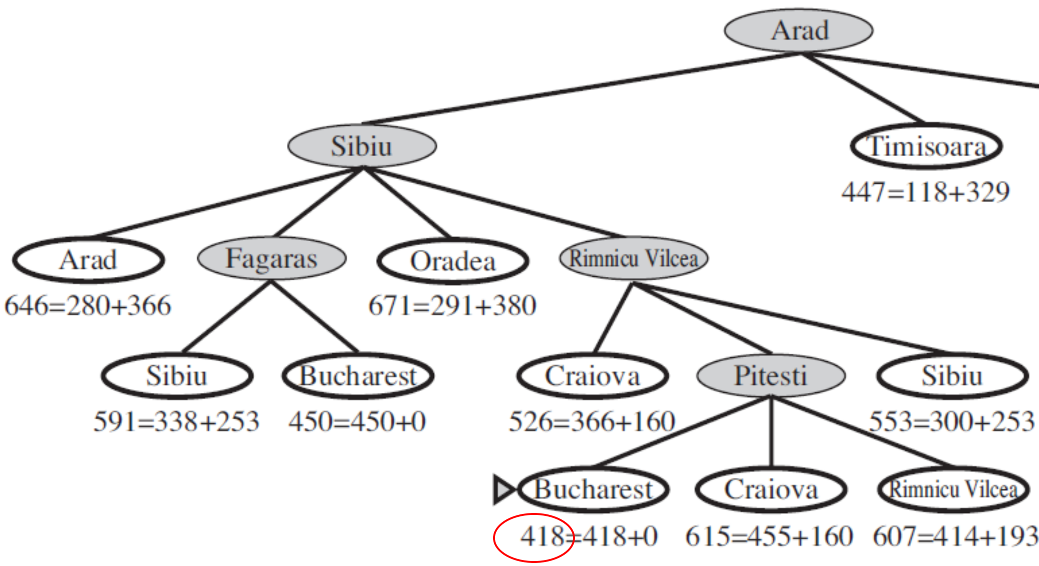


Properties of A* (summary)

- Complete?
  - Yes, unless infinitely many nodes with f≤f(G)
- Optimal?
  - Yes – it cannot expand contour f_{i+1} until f_i is finished
- Time?
  - Exponential
- Space?
  - Keep all nodes in memory

A* is also "optimally efficient"

- No other optimal algorithm is guaranteed to expand fewer nodes than A*
  - First, A* expands no nodes with ***f(n) > C*,*** where ***C**** is the cost of the optimal solution
  - Second, any other algorithm that does not expand all nodes with ***f(n)<C**** runs the risk of missing the optimal solution



Never expand nodes with cost > 418

Space complexity of A* is a bigger problem

- A* usually runs out of space long before it runs out of time
  - Because it keeps all generated nodes in memory
  - And the number of generated nodes is exponential in C*

- Solution?
  - Memory-bounded heuristic search

Outline for Today

- What is AI?
- Problem-solving agent
- Uninformed search
- **Informed search**
  - Greedy best-first search
  - A* search
  - **Memory-bound heuristic search**
  - Heuristic functions

Three variants

- Iterative-deepening A* (IDA*)
- Recursive best-first search (RBFS)
- Simplified memory-bound A* (SMA*)

Three variants

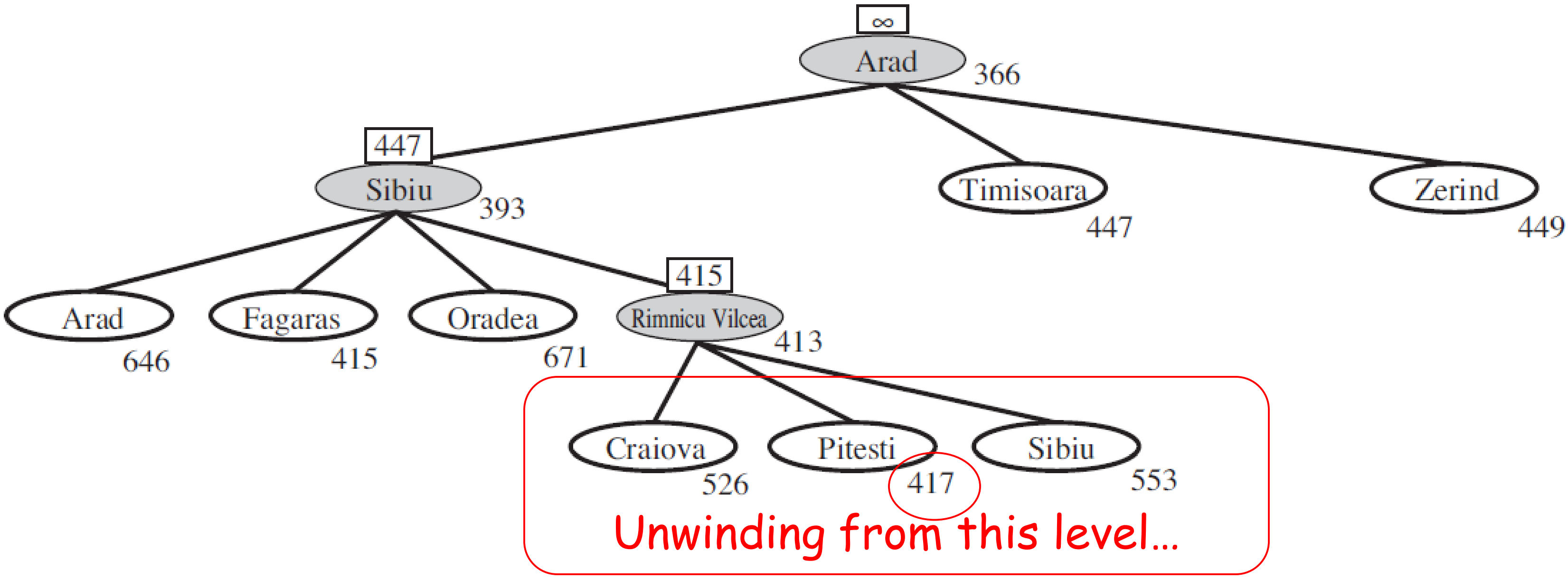
- Iterative-deepening A* (IDA*)
  - Instead of using “depth” as the cutoff, IDA* uses the f-cost (g+h) as the cutoff
  - No need to store the sorted queue of nodes
- Recursive best-first search (RBFS)
- Simplified memory-bound A* (SMA*)

Three variants

- Iterative-deepening A* (IDA*)
- Recursive best-first search (RBFS)
  - Similar to recursive DFS with depth limit, but uses ***f-limit*** to keep track of the total cost of the best alternative path available from any ancestor of the current node
    - If the cost of the current node (*f-cost*) exceeds this limit (*f-limit*), the recursion unwinds back to the alternative path
- Simplified memory-bound A* (SMA*)

RBFS (recursive best-first search)

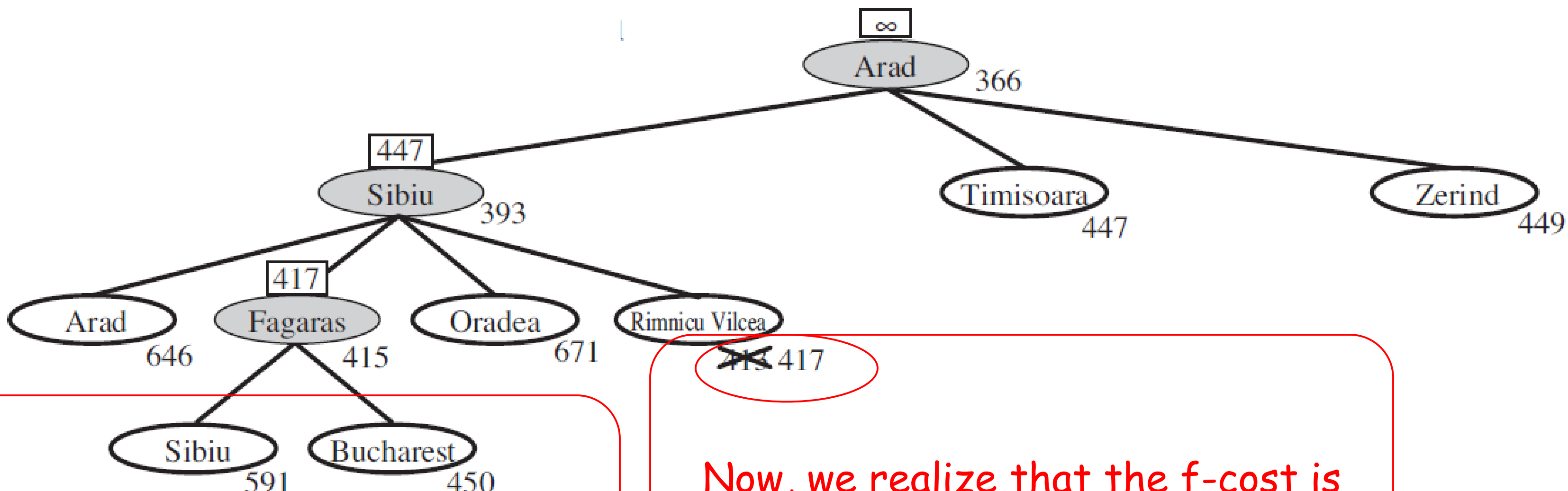
- **Above a node**: the f-limit for each recursive call on the node
- **Under a node**: f-cost associated with the node



- Before expanding "Sibiu", it computes the cheapest alternative path from sibling cities "Timisoara" and "Zerind" (and the f-limit is 447)
- Before expanding "Rimnicu Vilcea", it compute the cheapest alternative path from sibling cities "Arad", "Fargaras" and "Oradea" (and the f-limit is 415)

RBFS (recursive best-first search)

- **Above a node**: the f-limit for each recursive call on the node
- **Under a node**: f-cost associated with the node



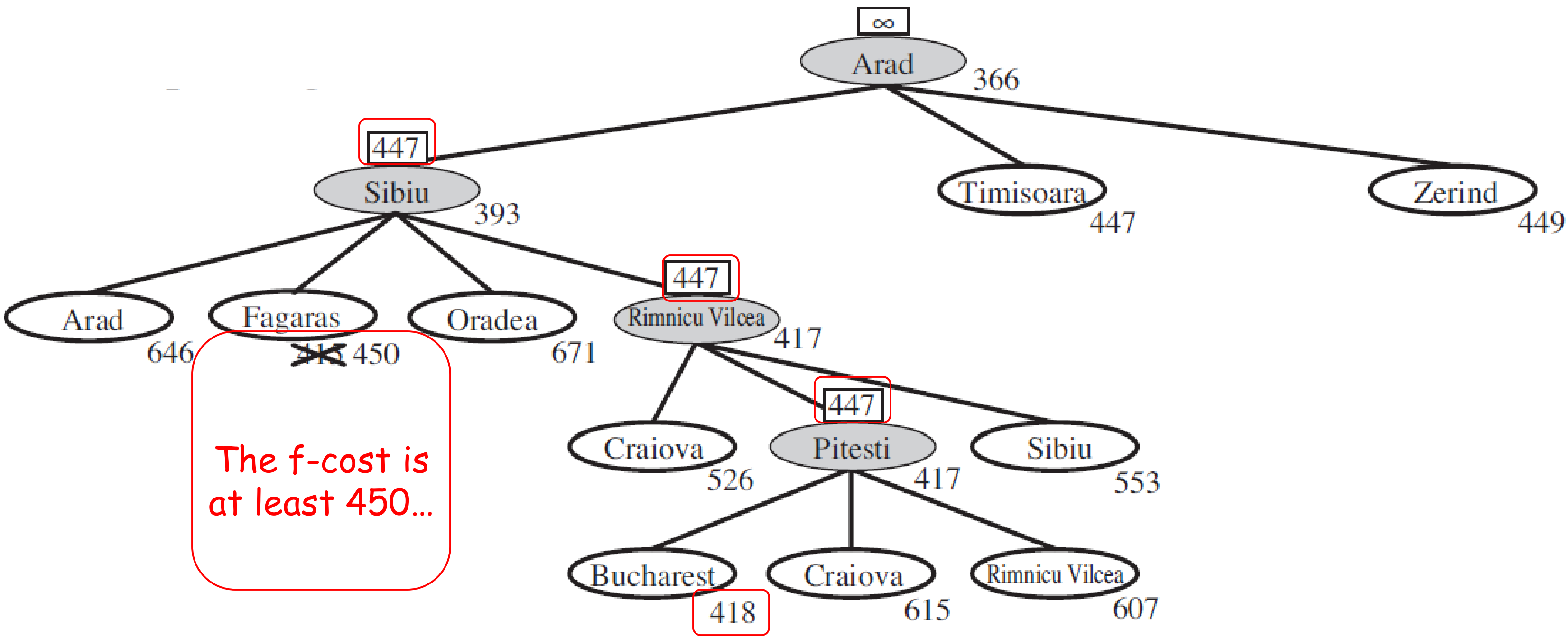
(450 is worse than 415, the cost we had expected; Unwinding …

Now, we realize that the f-cost is worst than 413, it's at least 417…

- Before expanding "Fagaras", the f-limit is set to 417 (newly computed from "Rimnicu Vilcea")
- When "Bucharest" has f-cost=450, which is greater than f-limit=417, it unwinds again…

RBFS (recursive best-first search)

Question: why is this "space efficient"?



Performance of RBFS

- Space complexity is **linear** in the depth of the deepest optimal solution

- **Optimal** if the heuristic function h(n) is admissible

- Time complexity is rather difficult to characterize
  - Depending on the accuracy of the heuristic function and how often the best path changes as the nodes are expanded

Three variants

- Iterative-deepening A* (IDA*)
- Recursive best-first search (RBFS)
- Simplified memory-bound A* (SMA*)
  - SMA* starts just like A*, expanding the best leaf until memory is full
  - At this point, it adds a new node to the search tree only after dropping an old one (the worst leaf node, with the highest f-cost)

Performance of SMA*

- Complete if there is any reachable solution
  - That is, the depth of the shallowest goal node is less than the memory size

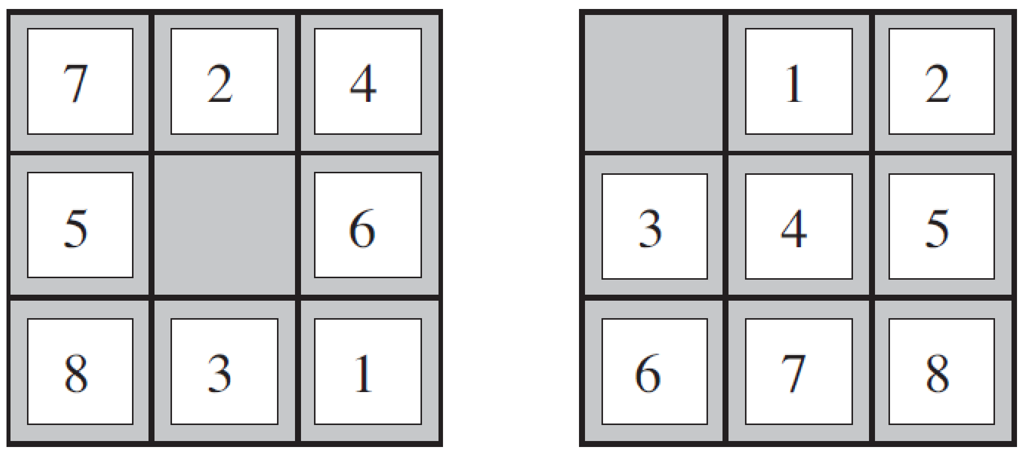
- Optimal if any optimal solution is reachable
  - Otherwise, it returns the best reachable solution

Outline for Today

- What is AI?
- Problem-solving agent
- Uninformed search
- **Informed search**
  - Greedy best-first search
  - A* search
  - Memory-bound heuristic search
  - **Heuristic functions**

How to evaluate heuristic functions?

- Example: 8-puzzle
  - Average solution cost is 22 steps
  - Branching factor < 3
  - Exhaustive tree search to depth 22 ➔ 3^22 states
  - Graph search ➔ 181,440 distinct states



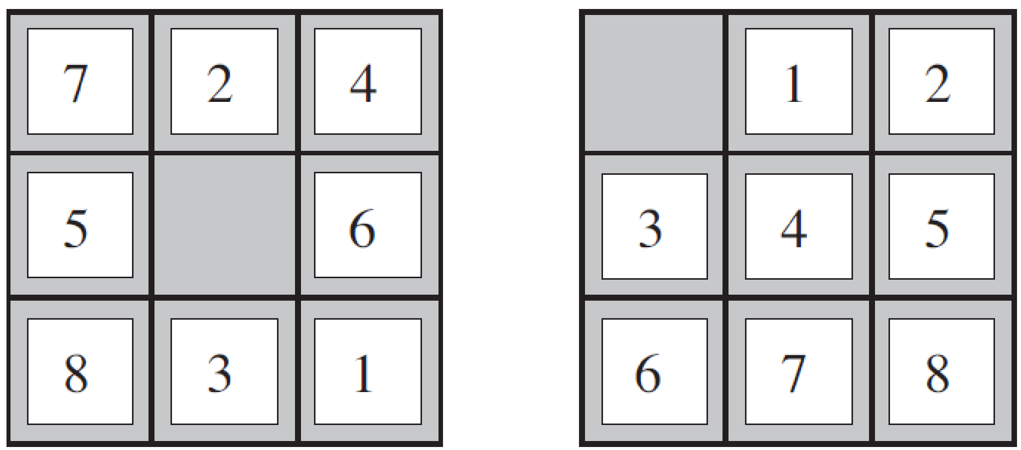
Start State Goal State

Heuristic functions for 8-puzzle

- Two candidates
  - h1(n) = number of misplaced tiles
  - h2(n) = sum of the distances of the titles from their goal positions
    - Manhattan distance: The sum of the horizontal and vertical distances

h1 = ??

h2 = ??



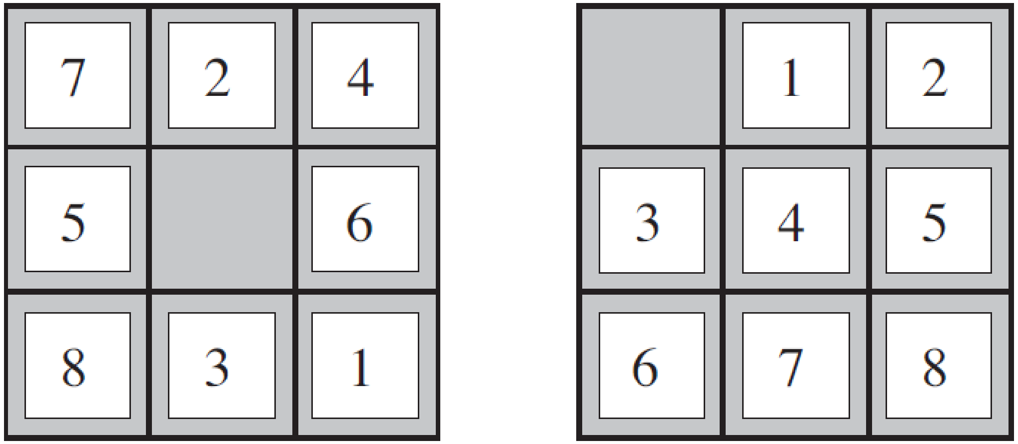
Start State Goal State

Heuristic functions for 8-puzzle

- Two candidates
  - h1(n) = number of misplaced tiles
  - h2(n) = sum of the distances of the titles from their goal positions
    - Manhattan distance: The sum of the horizontal and vertical distances

h1 = 8

h2 = 18



Start State Goal State

How do you know which heuristics is better?

Experimental results: A* with h1 and h2

- Solving 1200 random 8-puzzle problems

	Search Cost (nodes generated)		
d	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	–	539	113
16	–	1301	211
18	–	3056	363
20	–	7276	676
22	–	18094	1219
24	–	39135	1641

The effect of heuristic accuracy

- Effective branching factor b*
  - Assume the heuristic is perfect, A* would explore a total of (N) nodes and the solution depth is d

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

Let (N=52) and (d=5), what would be the value of (b*)?

Answer: b*=1.92

Experimental results: A* with h1 and h2

- Solving 1200 random 8-puzzle problems

	Search Cost (nodes generated)			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

Why Heuristic h2(n) is better than h1(n)?

- By their definitions, for any node ***n***, we have ***h1(n) ≤ h2(n)***
  - We say that h2 dominates h1

- **Domination translates into efficiency:** A* using h2 will never expand more nodes than A* using h1

$$f(n) < C^*$$

$$g(n) + h(n) < C^*$$

$$g(n) < C^* - h(n)$$

Why Heuristic h2(n) is better than h1(n)?

- By their definitions, for any node ***n***, we have ***h1(n) ≤ h2(n)***
  - We say that h2 dominates h1

- **Domination translates into efficiency**: A* using h2 will never expand more nodes than A* using h1

$$f(n) < C^*$$

$$g(n) + h(n) < C^*$$

$$\boxed{g(n) < C^* - h(n)}$$

larger h(n) ➔ smaller g(n)

when h(n)=0, g(n) is only bounded by C* (i.e., no guidance from the goal at all)

How to generate heuristic functions?

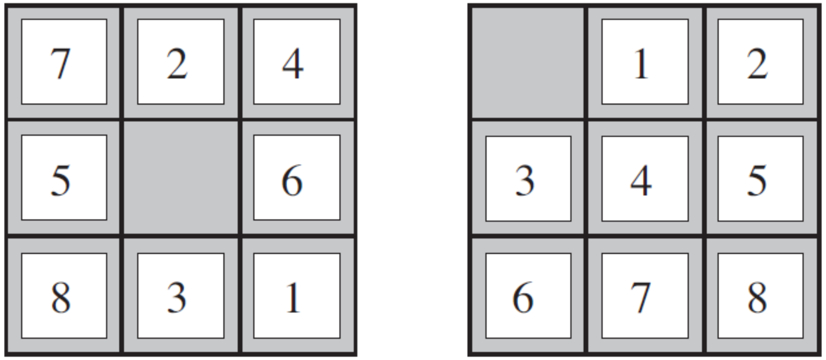
- Admissible heuristics can be derived from the exact solution cost of a “**relaxed**” version of the problem
- By “**relaxed**” we mean it has an “**over-approximation**” of the transition model of the original problem
  - More edges between states, to provide short cuts

Relaxed problems for 8-puzzle

- Original transition model

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B, and B is blank

- Relaxed transition model
  – (a) A tile can move from square A to square B if A is adjacent to B
  – (b) A tile can move from square A to square B if B is blank
  – (c) A tile can move from square A to square B.



Start State

Goal State

Relaxed problems for 8-puzzle

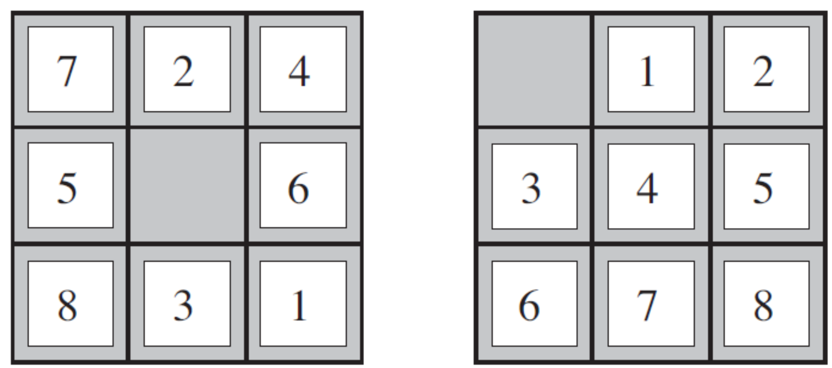
- Original transition model

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B, and B is blank

- Relaxed transition model
  - (a) A tile can move from square A to square B if A is adjacent to B
  - (b) A tile can move from square A to square B if B is blank
  - (c) A tile can move from square A to square B.

Number of misplaced tiles

Manhattan distance



Start State Goal State

Combining heuristic functions

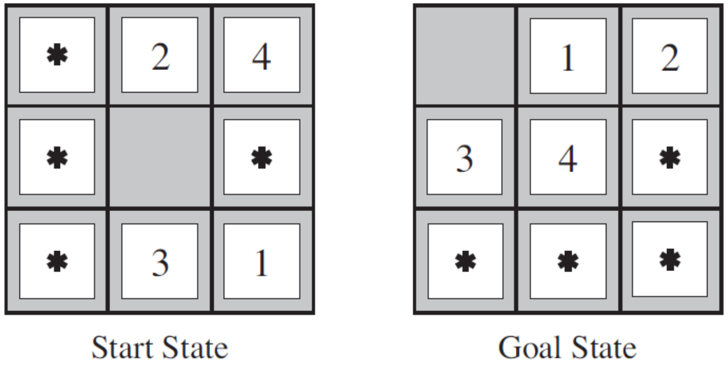
- Given a collection of admissible heuristics, the composite heuristic (which uses whichever function that is the most accurate on the node in question) is also admissible

$$\boxed{h(n) = \max\{h_1(n), \ldots, h_m(n)\}}$$

- Furthermore, h(n) dominates all of these component heuristics

Generating admissible heuristics from subproblems

- **Pattern Database:** store the exact solution costs for every possible subproblem instance
  - Every possible configuration of the first four tiles and the blank
  - Locations of the other four tiles are irrelevant (but moving them still counts toward the solution cost)



- The database can be constructed by searching back from the goal, and recording the cost of each pattern encountered

Learning heuristic functions from experience

- Inductive learning of a heuristic function ***h(n)*** in terms of features $x_1(n)$ and $x_2(n)$

$$\boxed{h(n) = c_1 x_1(n) + c_2 x_2(n)}$$

- Candidate features
  - "number of misplaced tiles"
  - "number of pairs of adjacent tiles that are not adjacent in the goal state

Outline for Today

- What is AI?
- Problem-solving agent
- Uninformed search
- **Informed search**
  - Greedy best-first search
  - A* search
  - Memory-bound heuristic search
  - Heuristic functions
  - **Summary of today’s lecture**

Summary

- Informed search may have access to a heuristic function, h(n), which estimate the cost of a solution from n
  - Greedy best-first search expands nodes with minimal h(n). It is not optimal but is often fast
  - A* search expands nodes with minimal f(n)=g(n)+h(n). It is complete and optimal, provided that h(n) is admissible (or consistent). The space complexity of A* is still high.
  - RBFS and SMA* are robust, optimal algorithms that use limited memory
- The performance of heuristic search algorithms depends on the quality of the heuristic function
  - Good heuristics can be constructed by (1) relaxing the problem definition, (2) storing precomputed solution costs for subproblems in a pattern database, or by (3) learning from experience