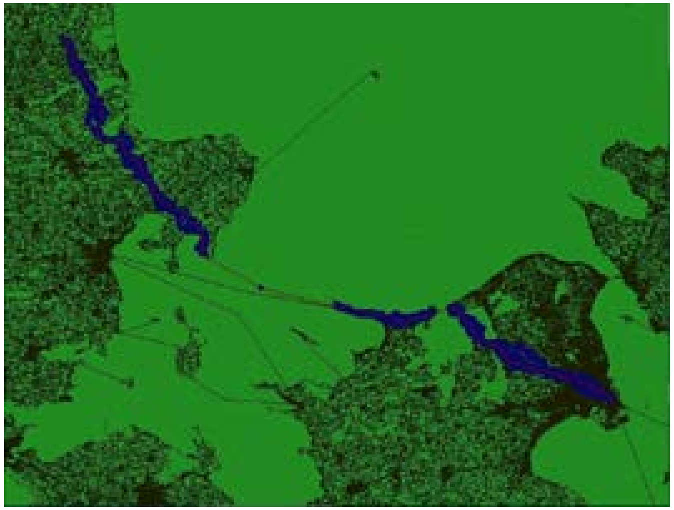
Intelligent Systems Programming

Lecture 2: Heuristic Search

RN13 3 (except 4.1, 4.2, 4.3, 4.6, 4.7, 5.3, 6.3, 6.4)

Today's Program

- 10:00-9:50: Uninformed Search
  - Tree and graph search
  - Depth-Limited Search (DLS)
  - Iterative Deepening Search (IDS)
- 11:00-11:50: Informed Search
  - Best-First Search
    - Greedy best-first search
    - A*
  - Heuristics
    - Domination
    - Relaxations
    - Pattern databases



AI Search Problems

What Characterizes AI Search?

- The search space is implicitly defined
- The search is goal directed
- The search space grows exponentially with problem size
- Search actions are abstract representations of real-world actions
  - Valid: can be decomposed to atomic actions
  - Useful: “atomic” plans between decision points are easy

Assumptions about the search domain

Assumptions

- The environment is static
- Actions are deterministic
- States are observable
- States and actions are discrete

Hold

- Toy puzzles, board games, computers

Don't hold at all

- Anything real unless highly controlled

Search Problem Definition

A search problem has 5 components:

1. An initial state s_0

2. A set of actions. ACTIONS(s) returns the set of actions applicable in state s

3. A transition model. RESULT(s,a) returns the state s' reached from s by applying a

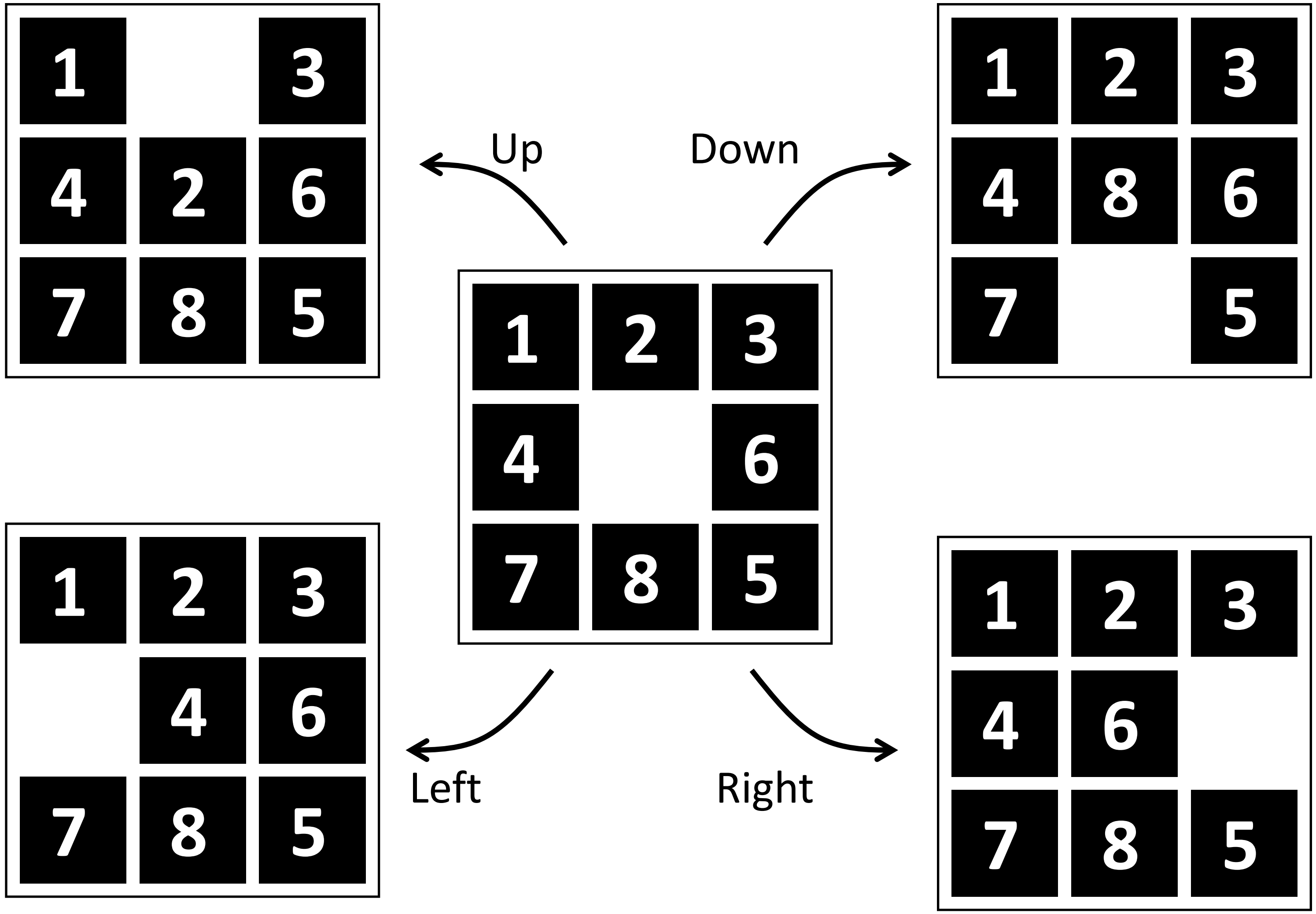
1+2+3 form a **state space**

4. A goal test. GOAL-TEST(s) returns true iff s is a goal state

5. A step cost function. STEP-COST(s,a) > 0 returns the step cost of taking action a to go from state s to state s'

- A solution is a path from the initial state s_0 to a goal state g
- An optimal solution is a path with minimum sum of step costs

Example: 8-Puzzle



Example: 8-Puzzle

- Initial state: s_0 = <1,2,3,4,*,6,7,8,5>
- Goal test function: s = <1,2,3,4,5,6,7,8,*>
- Actions: {Up, Down, Left, Right}
- Transition model: Defined by the rules:

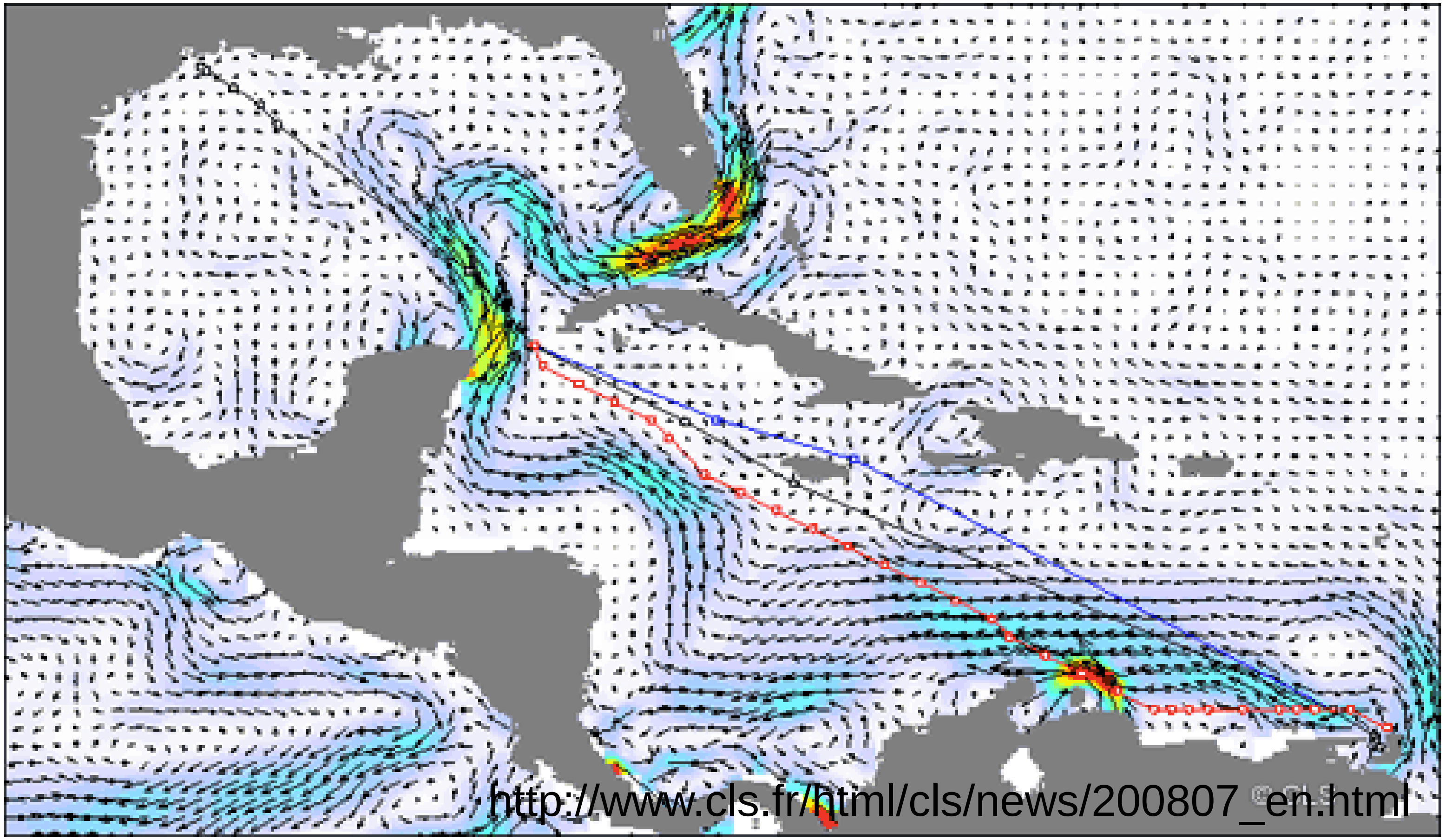
  1: Up (Down): applicable if some tile t above (below) *

  2: Left (Right): applicable if some tile t left (right) side of *

  3: The effect of actions is to swap t and *
- Size of state space: 9!/2
- Cost function: step-cost(s,a) = 1

Real Example. Ship routing

- Path cost: fuel [waves,current,speed] hotel cost

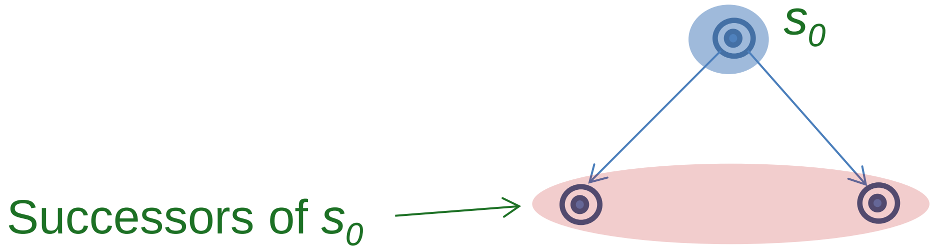


Tree Search

Tree Search

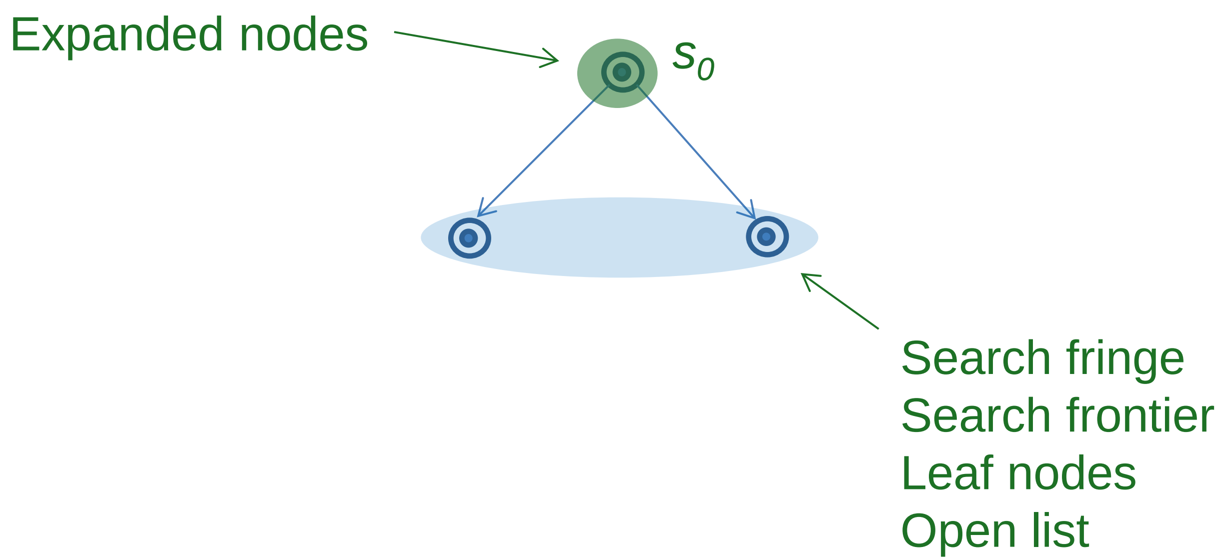
Start from search node with s_0

Tree Search

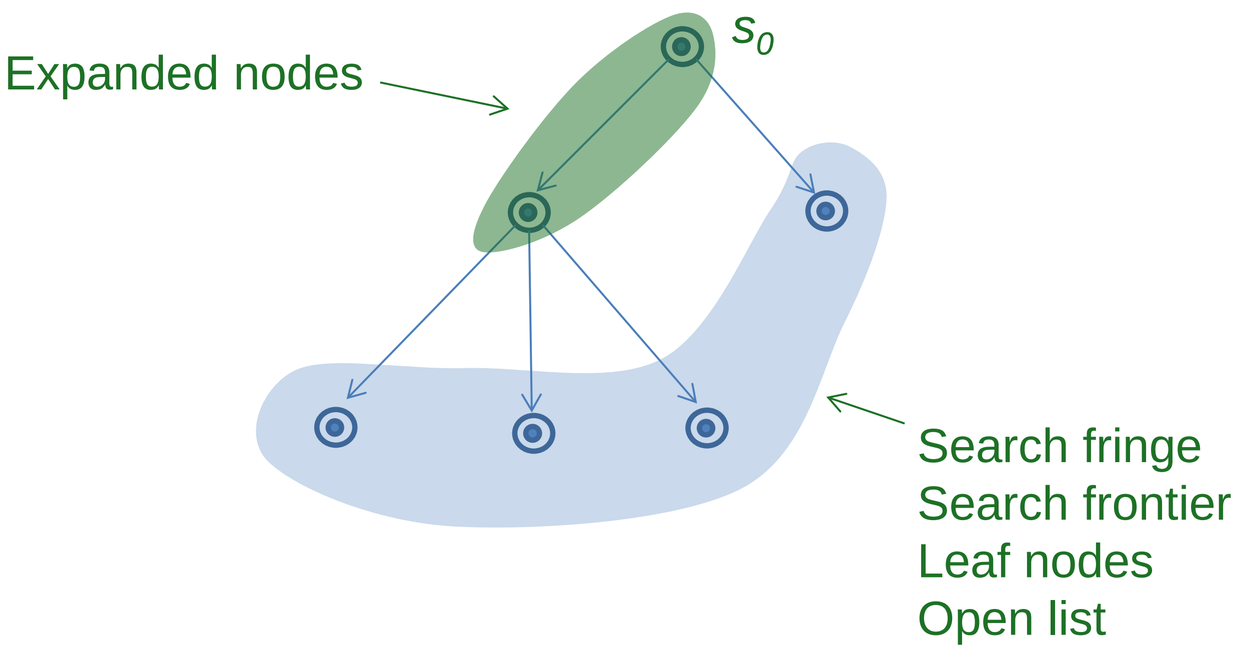


Tree Search

Expand s_0



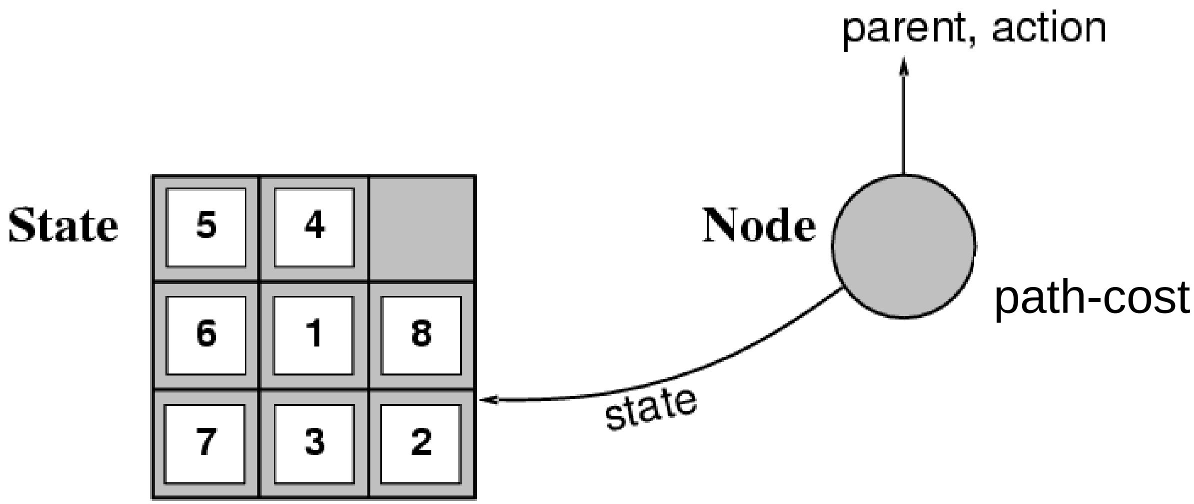
Tree Search



Algorithms vary by which leaf node they expand (search strategy)!

Implementation: General Tree Search

Search node data structure

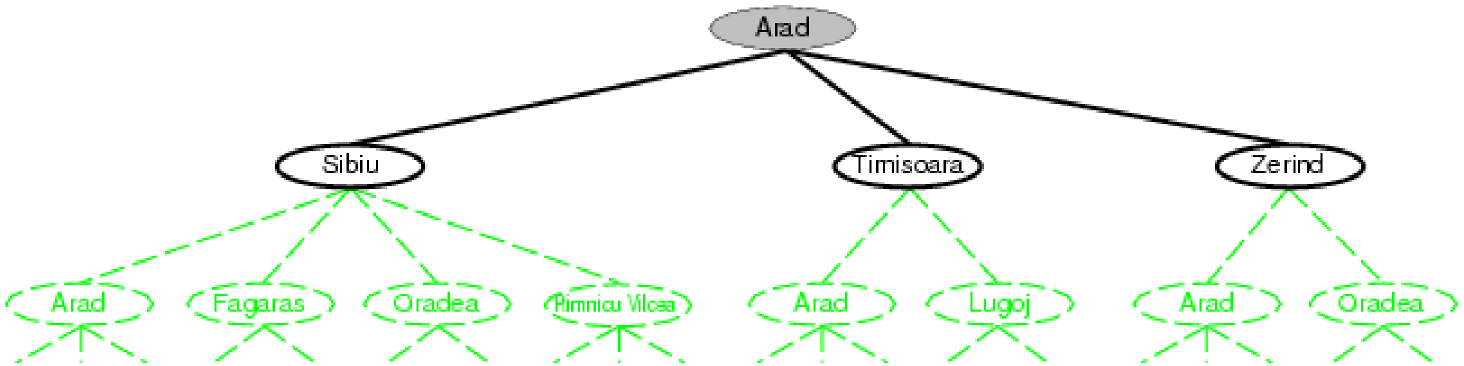


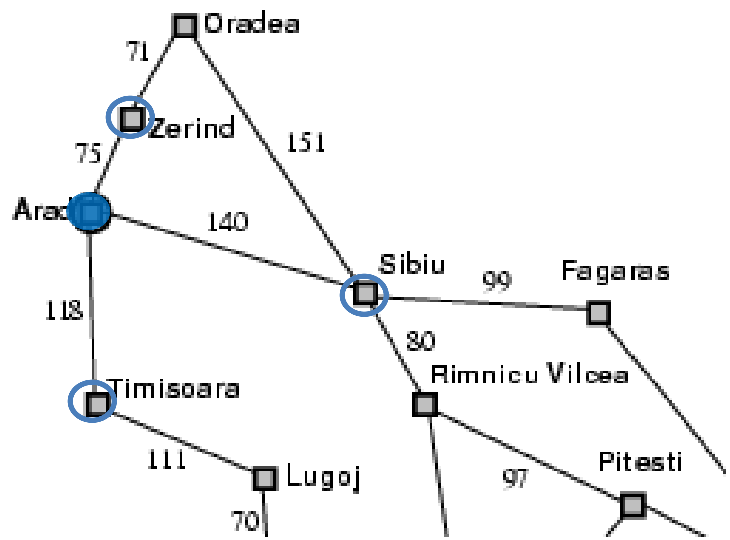
Function CHILD-NODE(*problem*,*parent*,*action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE,*action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE,*action*)

Implementation: General Tree Search

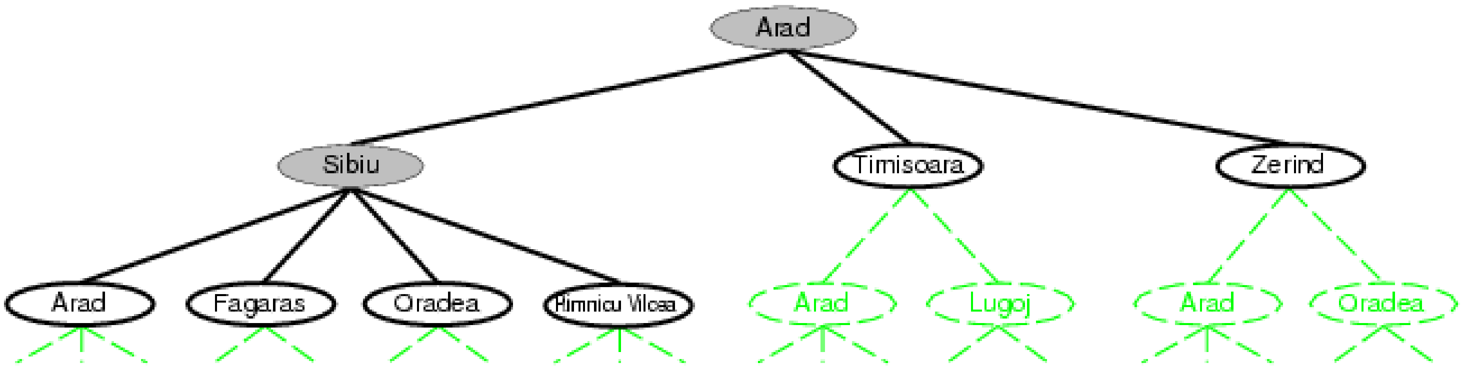
Function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then**
      return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

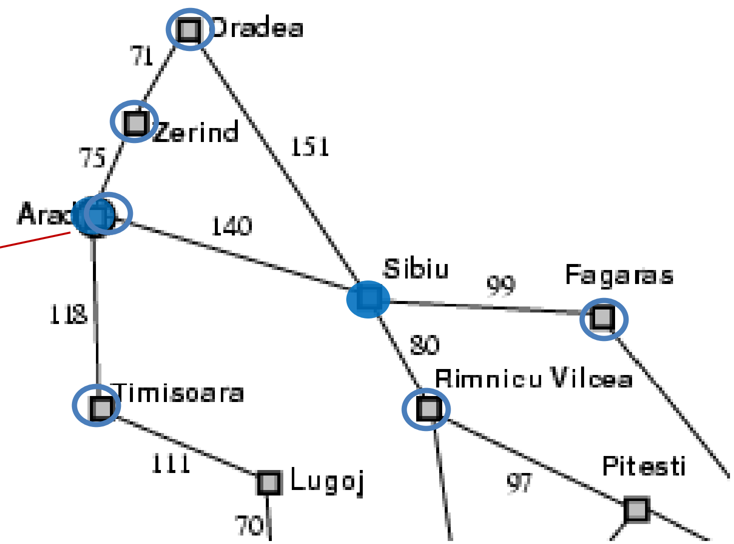
Tree Search Ex.: Routing





Tree Search Ex.: Routing

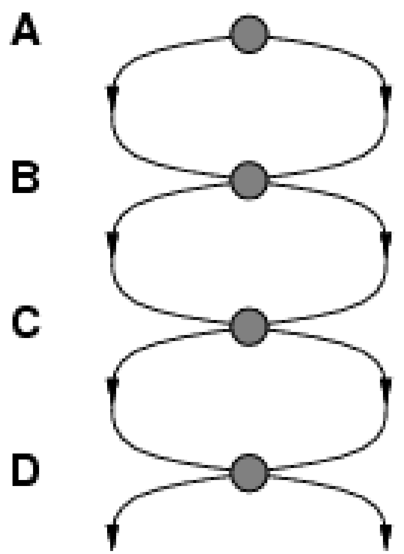


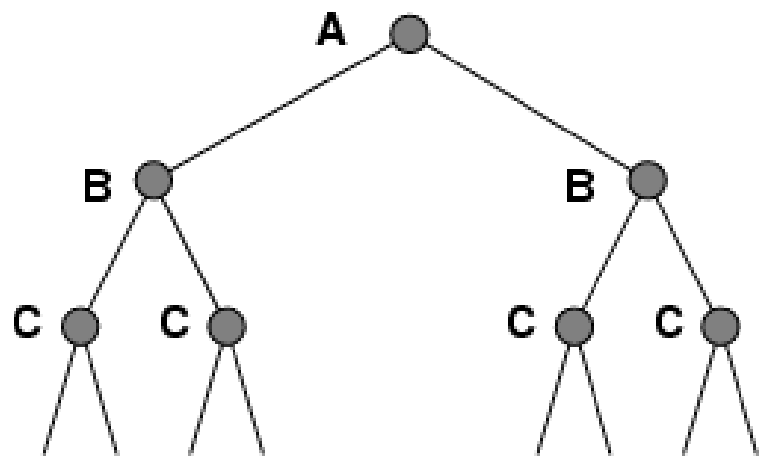


We can disregard paths with loops, why?

Repeated States

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





- Graph search: only keep the first path to a state

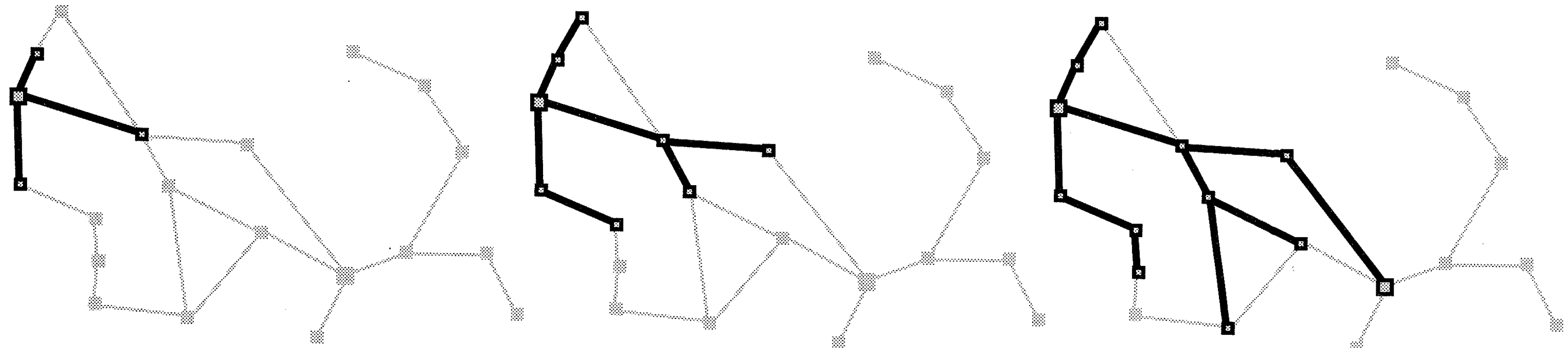
Implementation: General Graph Search

Function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
initialize the frontier using the initial state of *problem*
initialize the explored set to be empty
loop do
if the frontier is empty **then return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state **then**
return the corresponding solution
add the node to the explored set
expand the chosen node, adding the resulting nodes to the frontier
only if not in the frontier or explored set

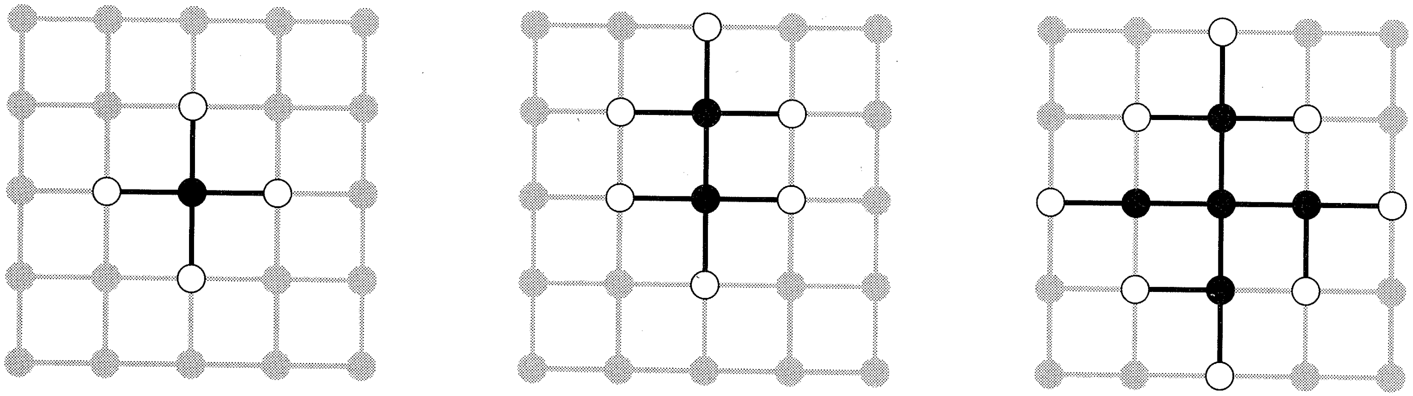
Graph search can overlook an optimal solution, how?

Properties of Graph Search

- Explores the state-space graph



- Satisfies the separation property



Performance characteristic

- time complexity: number of nodes generated
- space complexity: maximum number of nodes in memory
- (soundness: is the produced solution correct)
- completeness: does it always find a solution if one exists
- optimality: does it always find a least-cost solution?

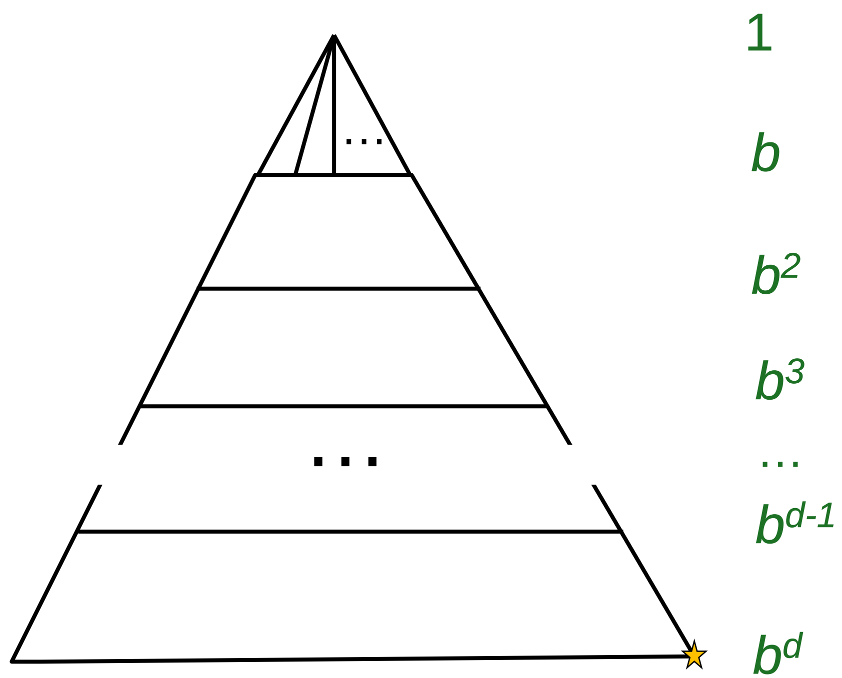
Time and space complexity

Due to the implicit problem representation the number of bits in the input is not a good measure of problem size

Input size is 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 infinite)

Example: Breath-First Tree Search



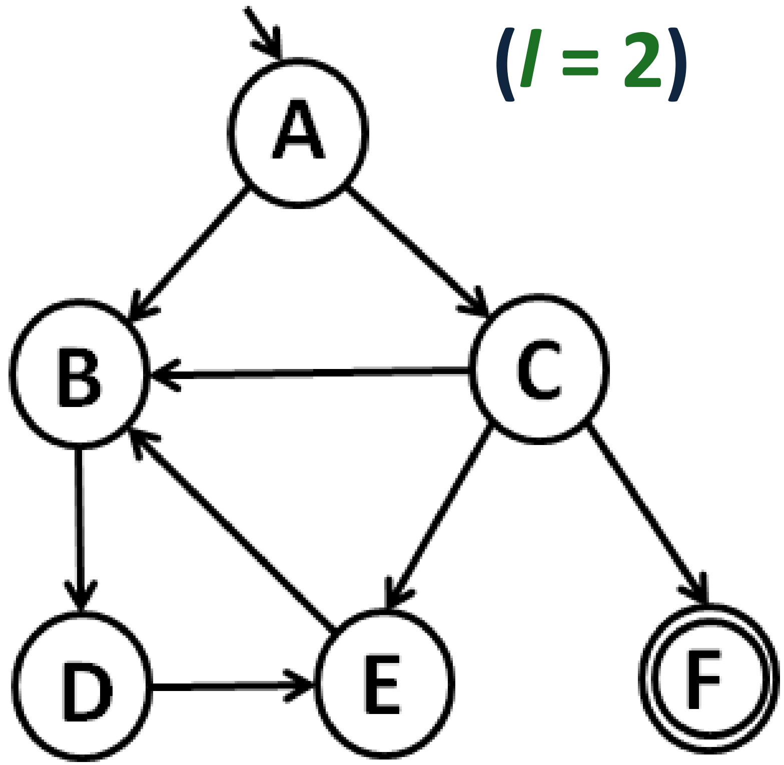
$$O\left(\sum_{i=0}^{d} b^i\right) = O\left(b^d\right)$$

Uninformed Search Algorithms

Depth-limited search (DLS)

- Idea: only do depth first search (space efficient) to a limited depth l (complete within l)
- Example

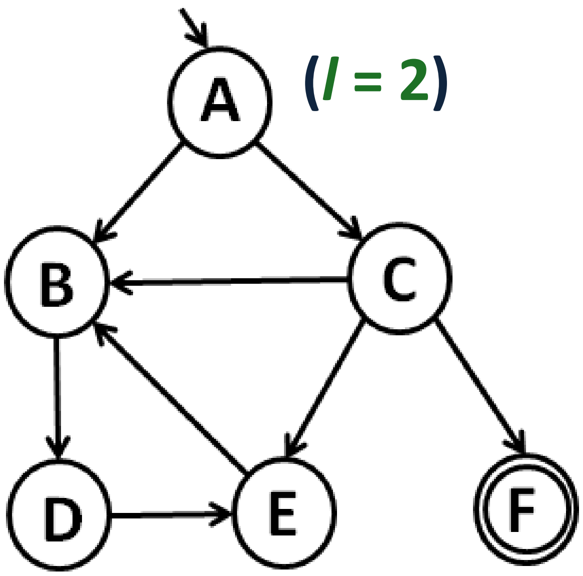
(l = 2)



Depth-limited search (DLS)

Function DEPTH-LIMITED-SEARCH(*problem*,*limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE),*problem*,*limit*)

Function Recursive-DLS(node,*problem*,*limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
   cutoff_occured? ← false
   for each *action* in *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*,*node*,*action*)
    result ← RECURSIVE-DLS(*child*,*problem*,*limit*-1)
    if *result* = *cutoff* **then** *cutoff_occured?* ← true
    else if *result* is a solution **then return** *result*
  if *cutoff_occured?* **then return** *cutoff* **else**
     return *failure*



Properties of depth-limited search

- Complete? No (unless $l \geq d$)
- Time? $O(b^l)$
- Space? $O(l)$ (book wrong not $O(bl)$)
- Optimal? No (unless $l = d$ and constant step cost)

- DFS = DLS with $l = \infty$

Iterative deepening search (IDS)

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

  inputs: *problem*, a problem

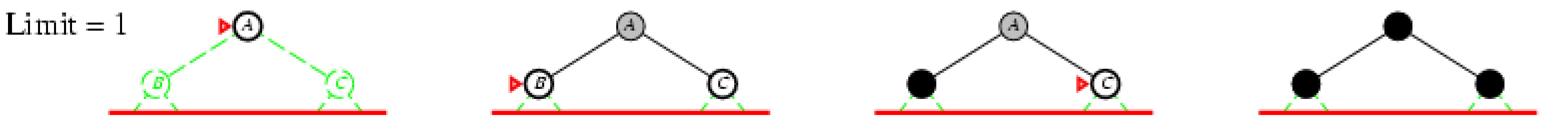
  for *depth* ← 0 **to** ∞ **do**

    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)

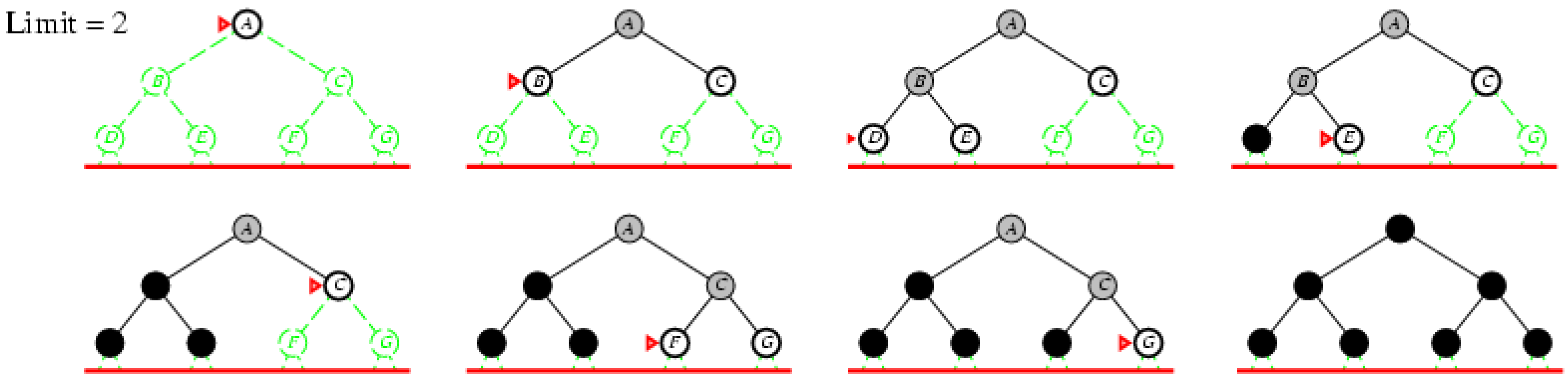
    if *result* ≠ cutoff **then return** *result*

Iterative deepening search / =0

Iterative deepening search / =1

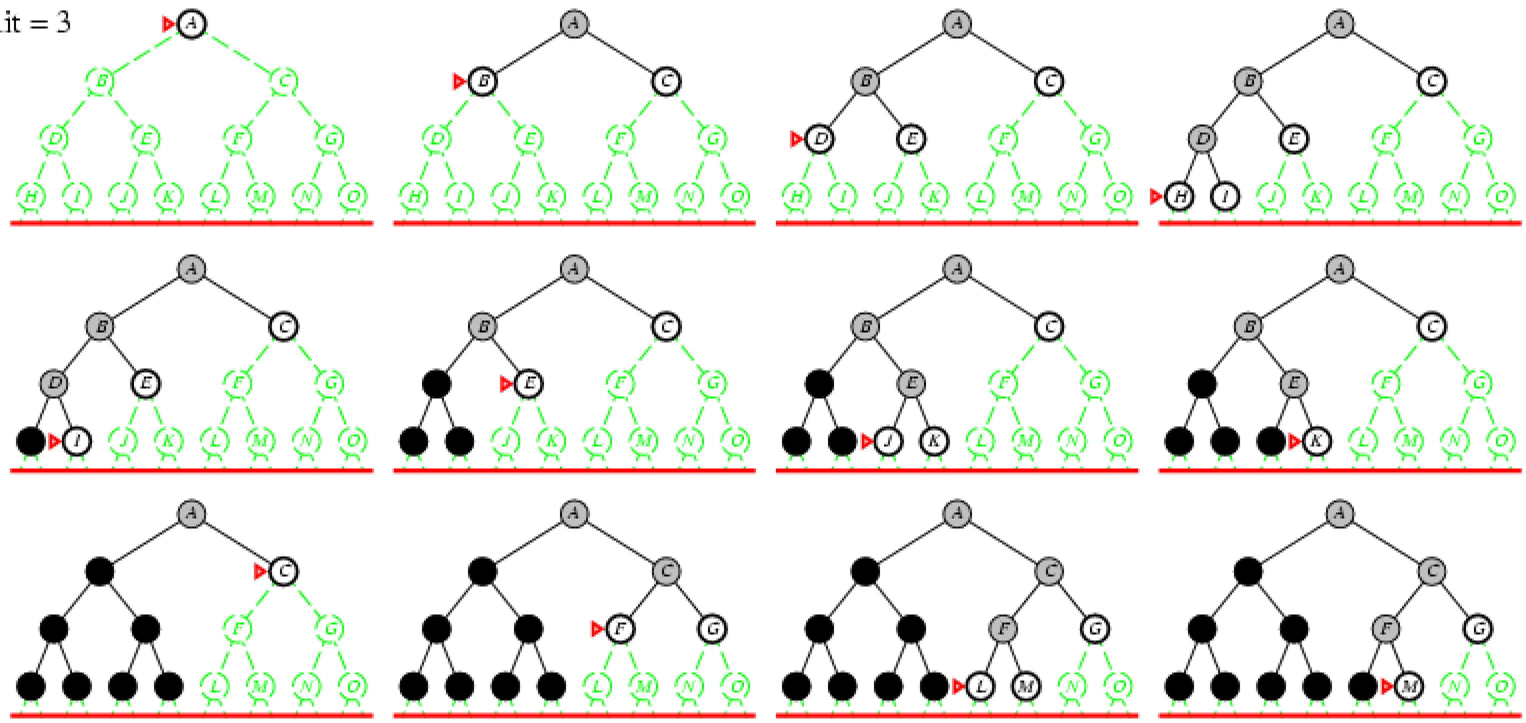


Iterative deepening search l =2



Iterative deepening search l =3

Limit = 3



Iterative Deepening Search

- Number of nodes generated in a depth-limited search with $l = d$ and branching factor b:

$$N_{DLS} = b + b^2 + \ldots + b^d$$

- Number of nodes generated in an iterative deepening search to depth d with branching factor b:

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

- For $b = 10$, $d = 5$,
  - $N_{IDS} = 50 + 400 + 3000 + 20000 + 100000 = \mathbf{123450}$
  - $N_{DLS} = 10 + 100 + 1000 + 10000 + 100000 = \mathbf{111110}$ (Book has a version of BFS that behaves same way)

- Overhead of IDS compared with DLS: $(123450 - 111110)/111110 \approx 11\%$

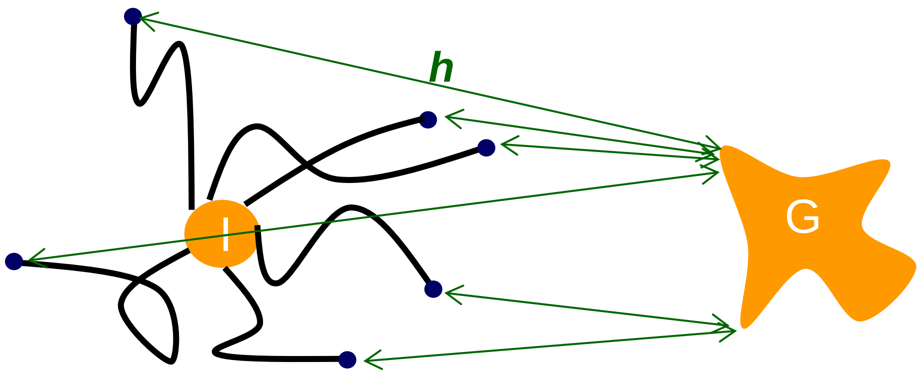
Properties of Iterative Deepening Search

- Complete? Yes
- Time? $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(d)$
- Optimal? Yes, if step cost is constant

Informed Search Algorithms

Informed Search

- Idea: use problem specific knowledge to pick which node to expand
- Typically involves a heuristic function $h(n)$ estimating the cheapest path from n.STATE to a goal state



Requirements

$h(s) \geq 0$

$h(goal) = 0$

Best-First Search

Classical Tree Search or Graph Search with:

1. PATH-COST(n) now called $g(n)$
2. Expansion of fringe node with lowest cost according to an evaluation function $f(n)$
3. CHILD-NODE() extended to update $f(n)$

Best-First Tree Search

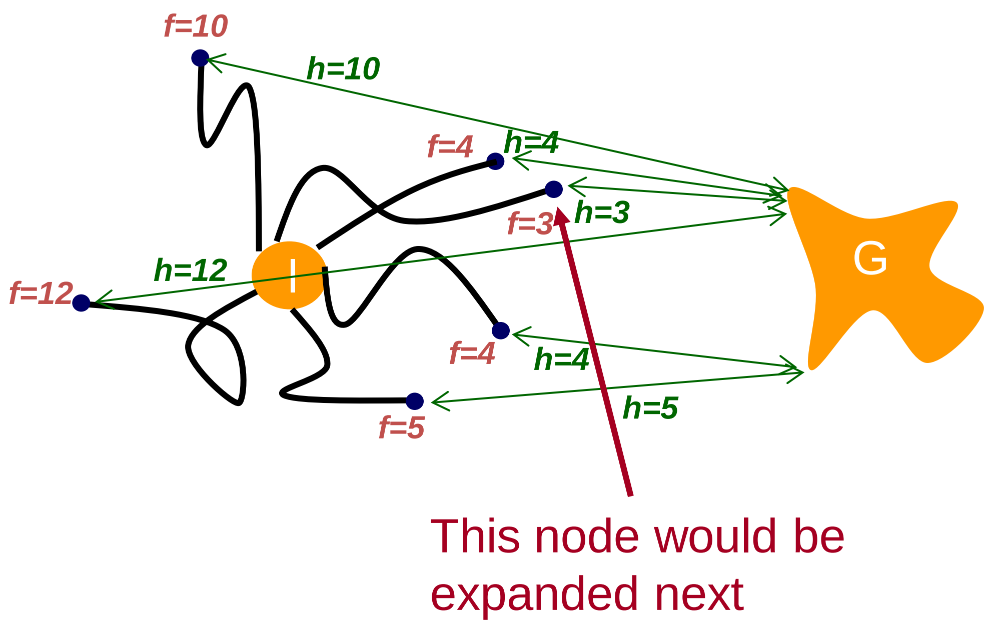
```
Function BEST-FIRST-TREE-SEARCH(problem) returns a solution or failure
  node ← a node with STATE = problem.INITIAL-STATE, g(node) = 0
  frontier ← a queue ordered in ascending f-value, containing node

  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier);
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem,node,action)

        frontier ← INSERT(child,frontier)
```

Best-First Graph Search

Function BEST-FIRST-GRAPH-SEARCH(*problem*) **returns** a solution or failure
    node ← a node with STATE = *problem*.INITIAL-STATE, *g*(*node*) = 0
    frontier ← a queue ordered in ascending *f*-value, containing *node*
    explored ← an empty set
    loop do
        if EMPTY?(*frontier*) **then return** failure
        node ← POP(*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*

Greedy Best-First Search

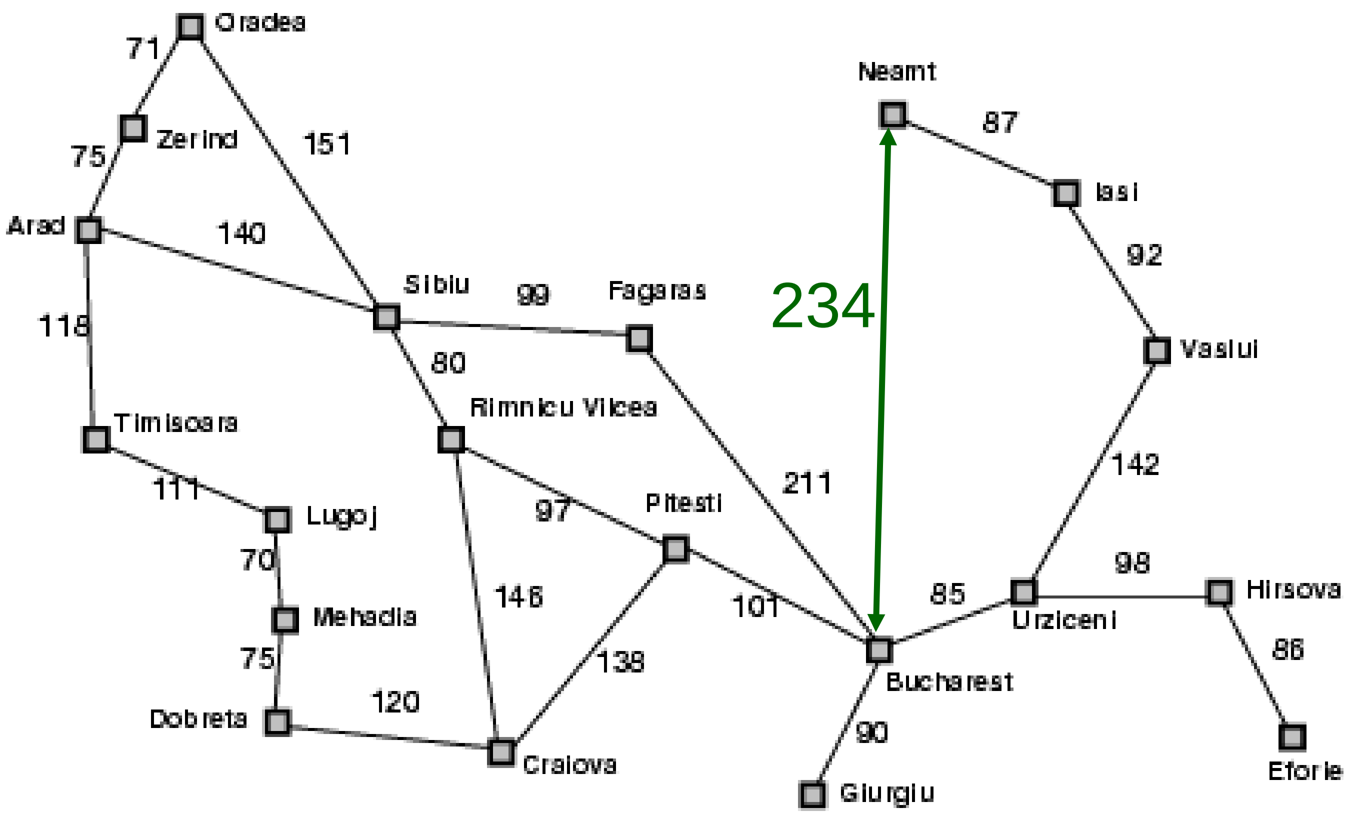
- $f(n) = h(n)$: Expand node that appears to be closest to the goal



Greedy Best-First Tree Search Example

- Go from Arad to Bucharest
- $f(n) = h_{SLD}(n) =$
  straight-line distance from n.STATE to Bucharest

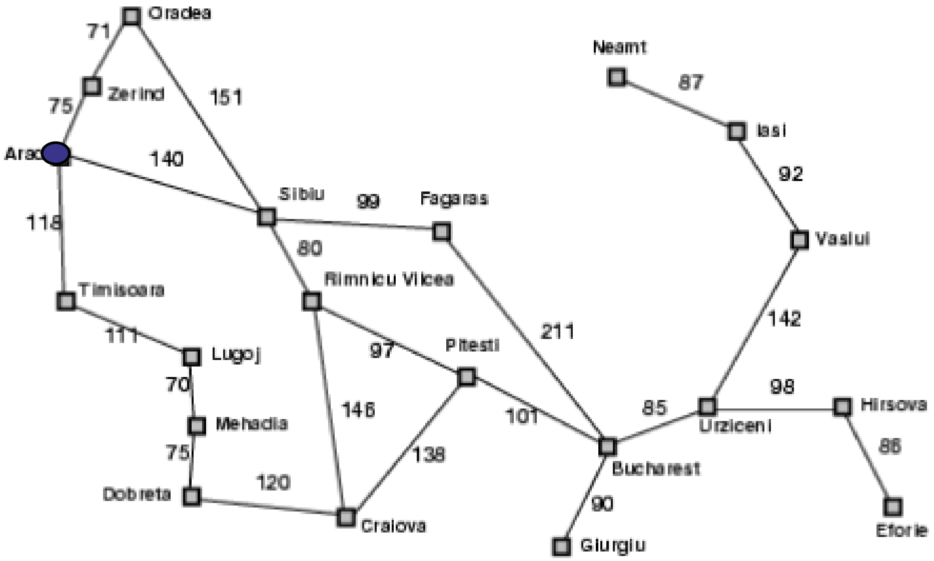
Romania with Step Costs in km



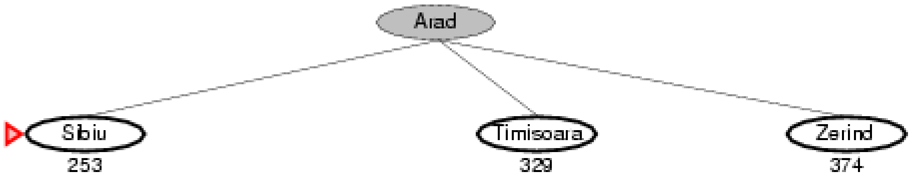
h(*n*)

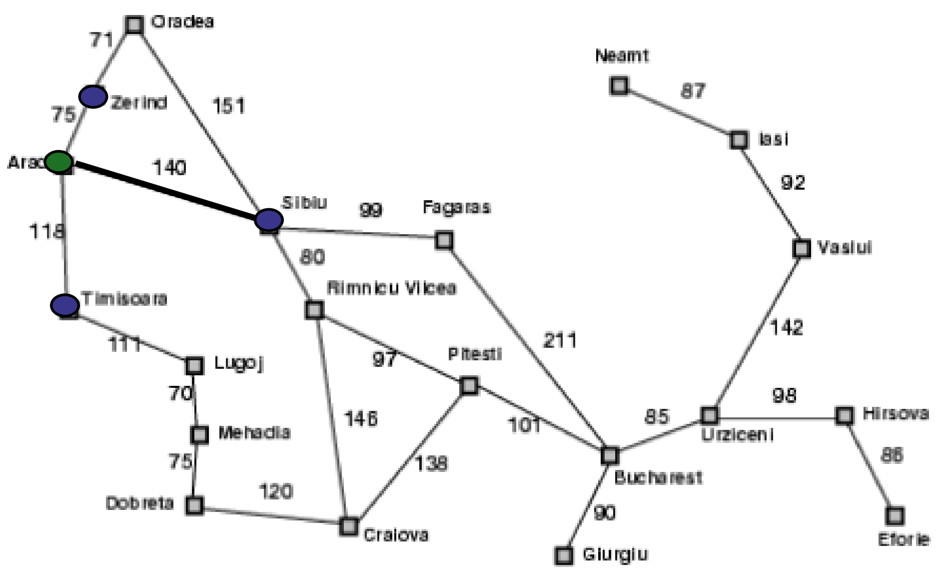
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Tree Search Example

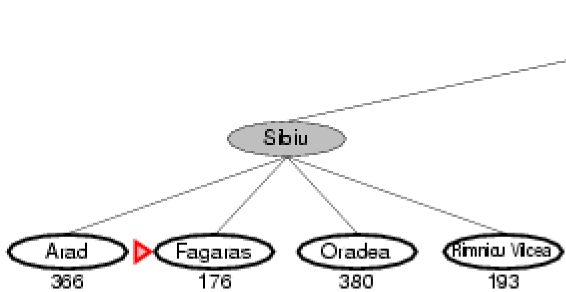


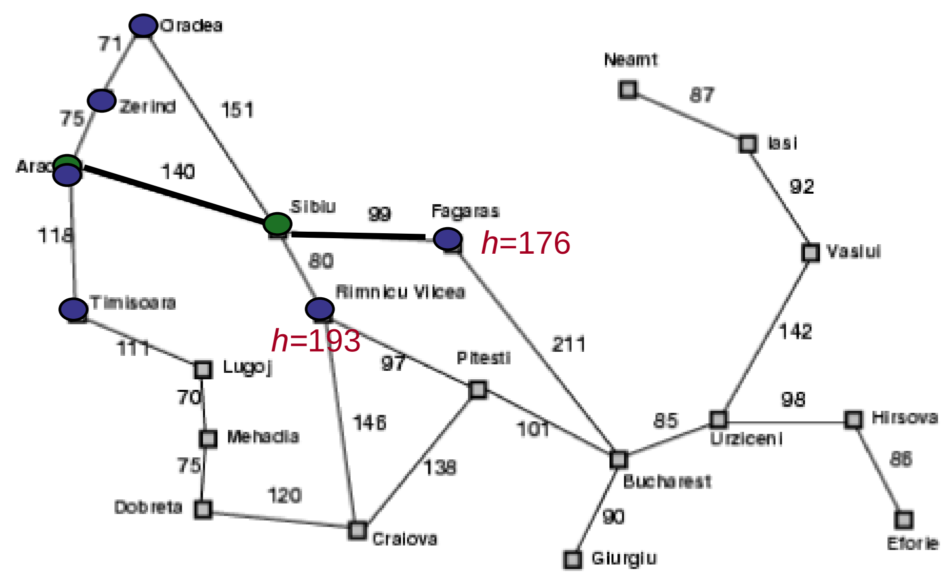
Greedy Best-First Tree Search Example



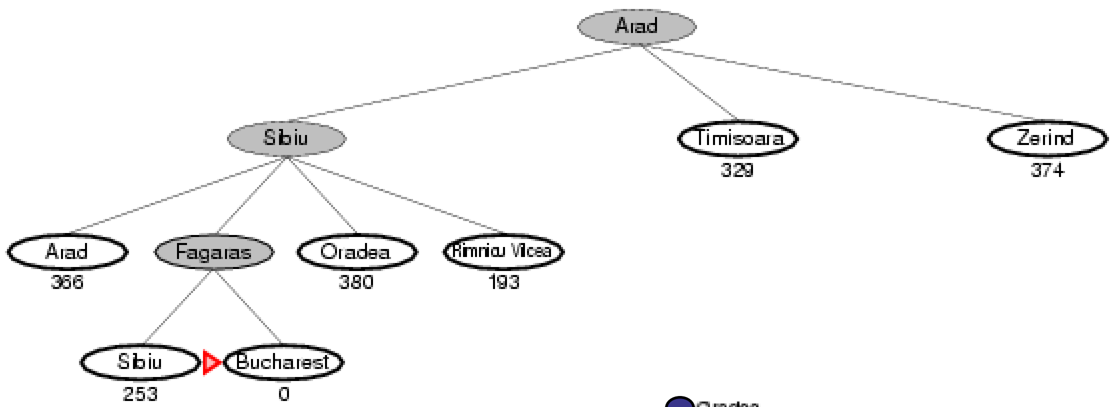


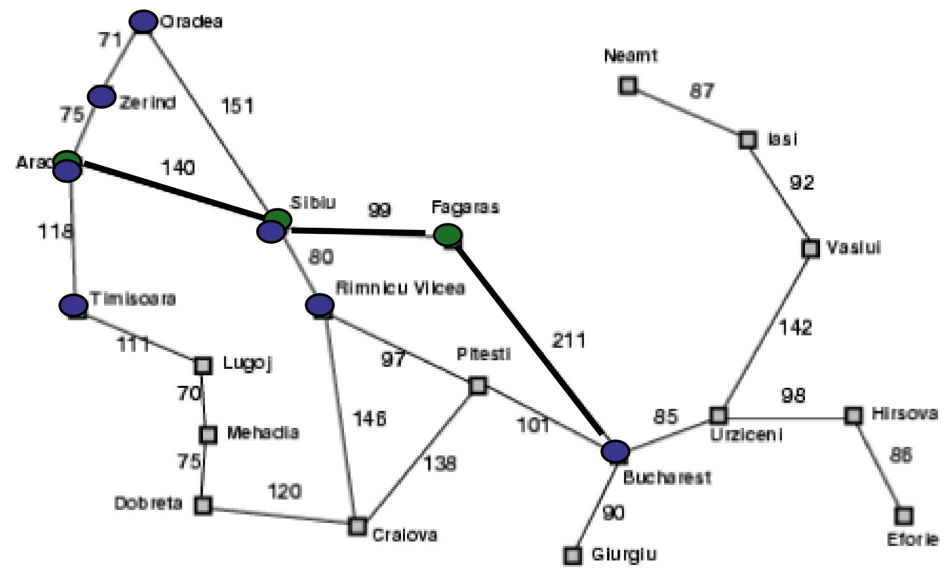
Greedy Best-First Tree Search Example



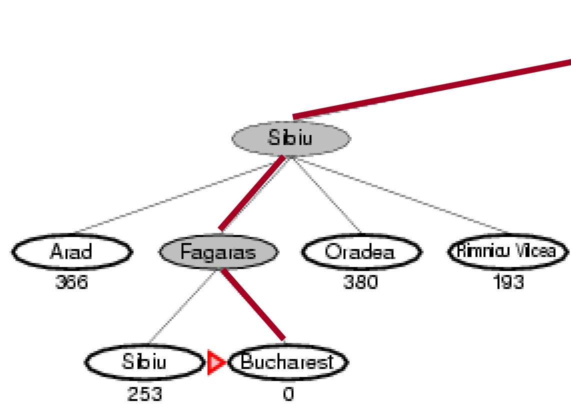


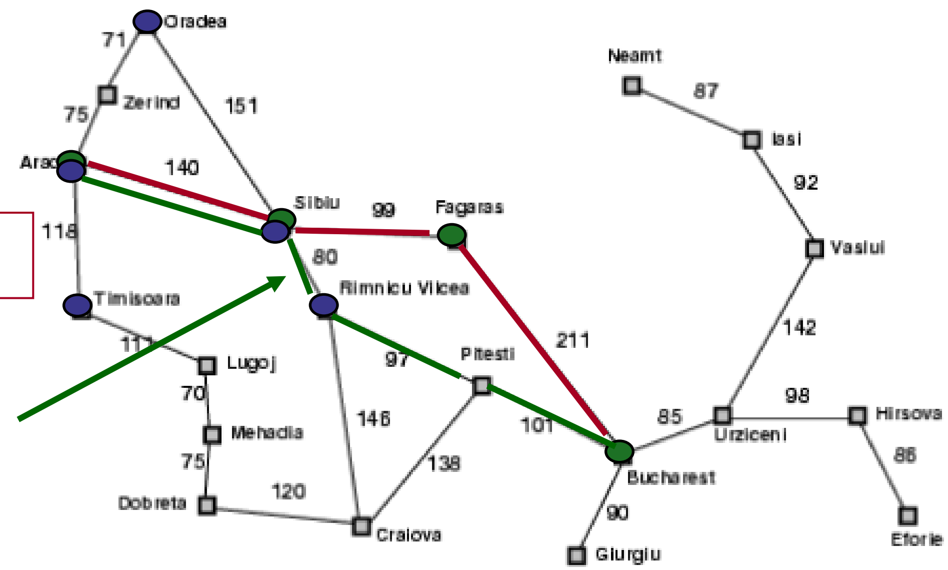
Greedy Best-First Tree Search Example





Greedy Best-First Tree Search Example



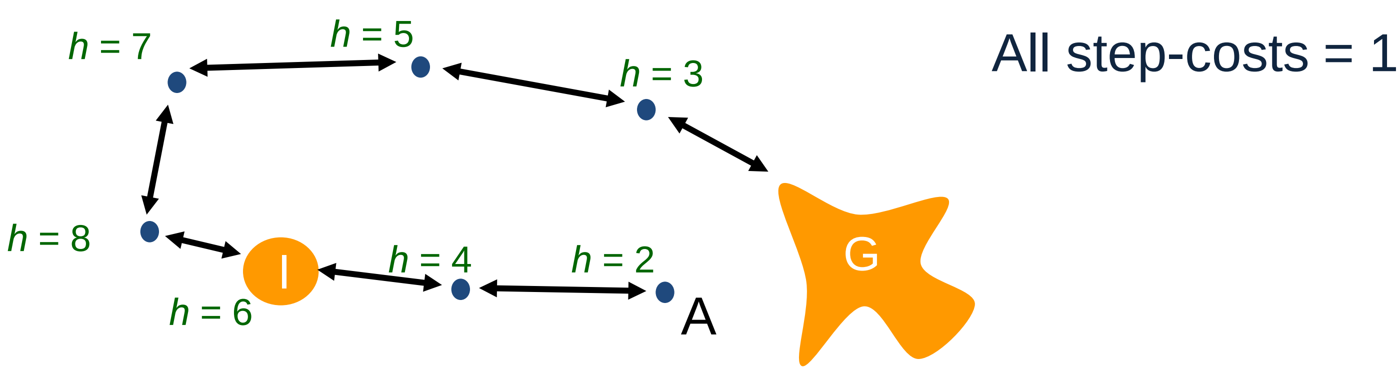


Solution turns out not to be optimal

Optimal solution

Completeness

- How does greedy best-first tree search behave on problems like the one below?



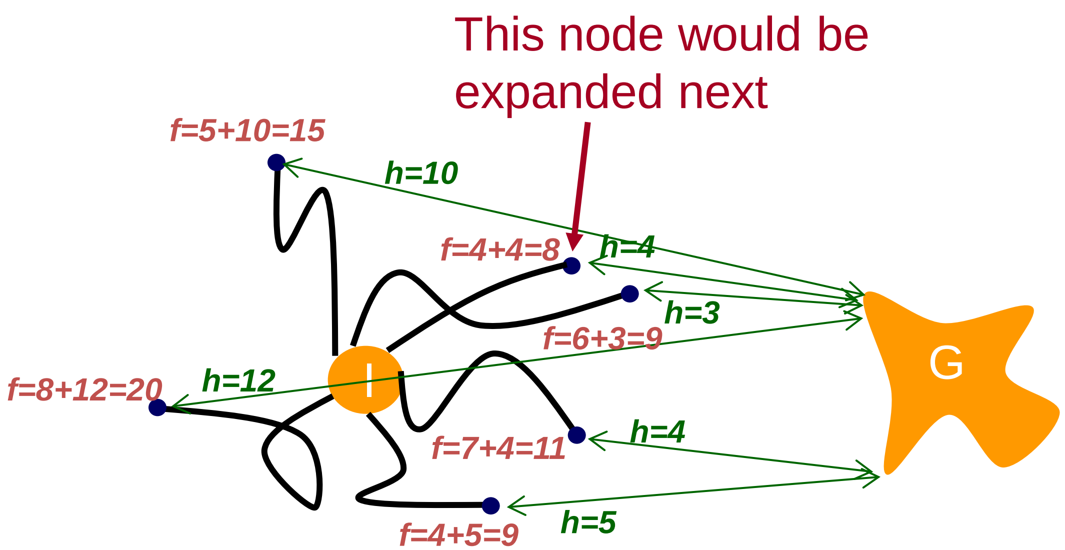
- What about greedy best-first graph search?

A* Search

- Idea: include cost of reaching node
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost of reaching n
- $h(n)$ = estimated cost of reaching goal from state of n
- $f(n)$ = estimated cost of the cheapest path to a goal state that goes through path of n

A* Search

- $f(n) = g(n) + h(n)$: Expand node that appears to be on cheapest paths to the goal



Admissible Heuristics

- A heuristic $h(n)$ is admissible if for every node n,

  $h(n) \leq h^*(n)$, where $h^*(n)$ is the minimum cost to reach the goal state from n

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

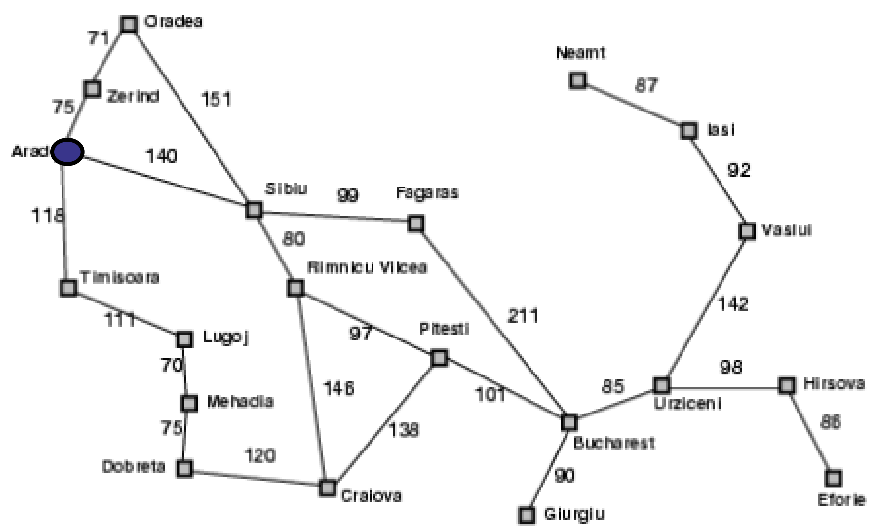
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

- Can you think of a heuristic function that is trivially admissible?

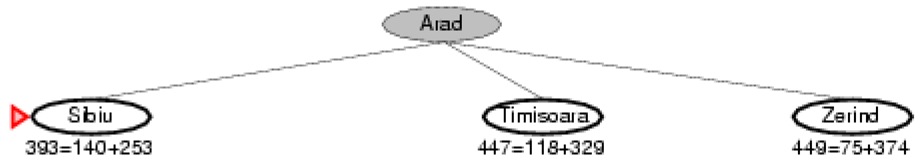
A* Tree Search Example

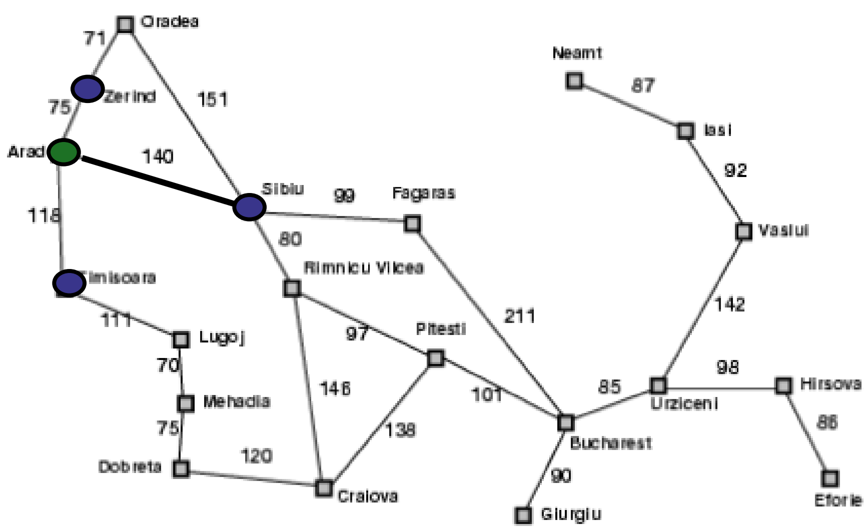
- Go from Arad to Bucharest
- $f(n) = g(n) + h_{SLD}(n)$

A* Tree Search Example

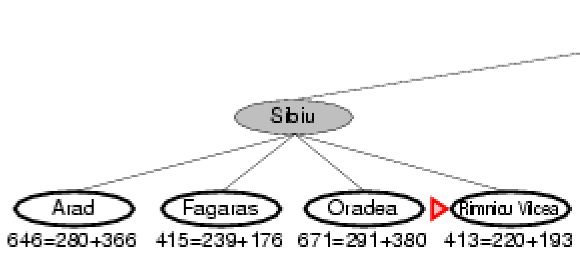


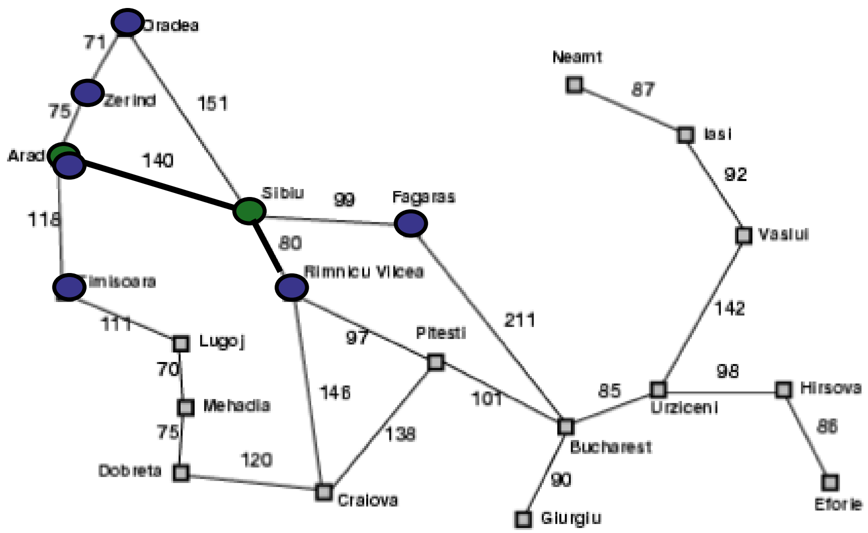
A* Tree Search Example



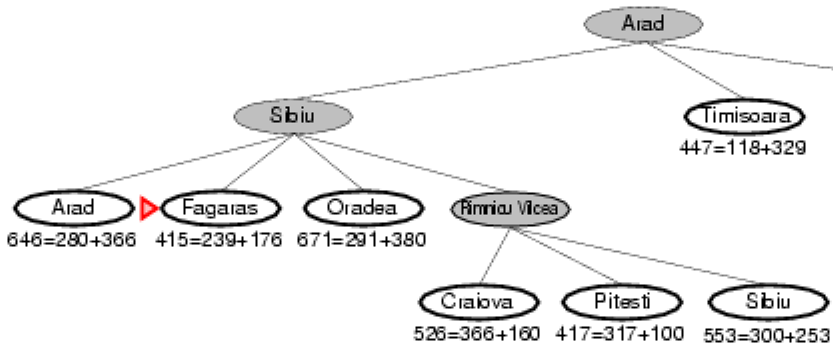


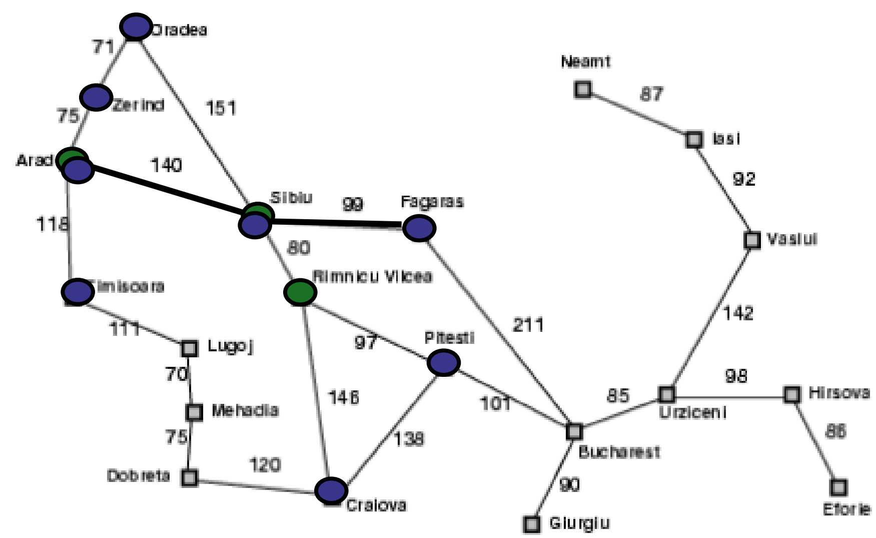
A* Tree Search Example



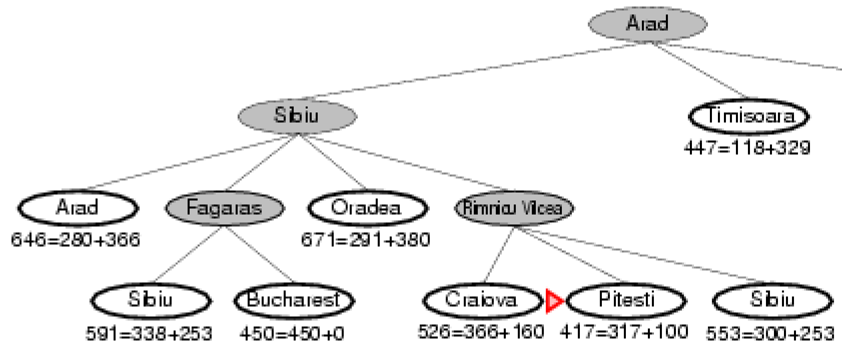


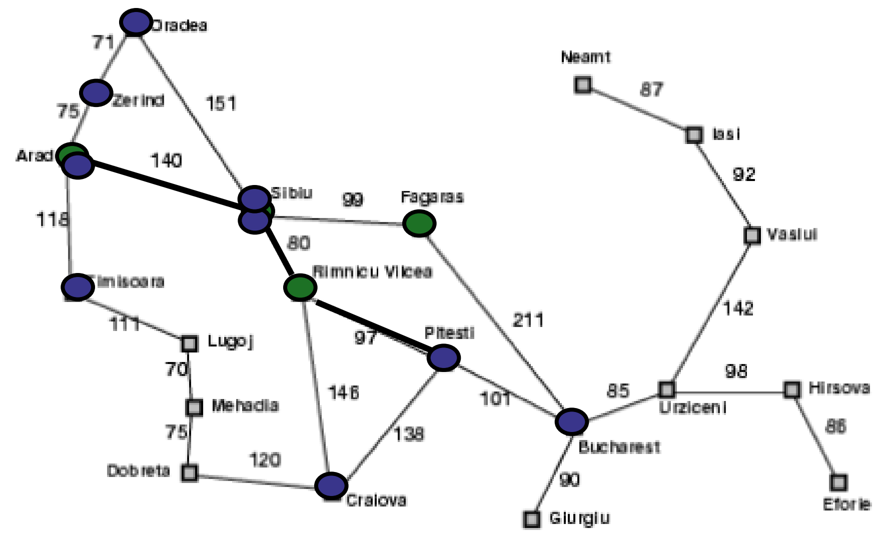
A* Tree Search Example



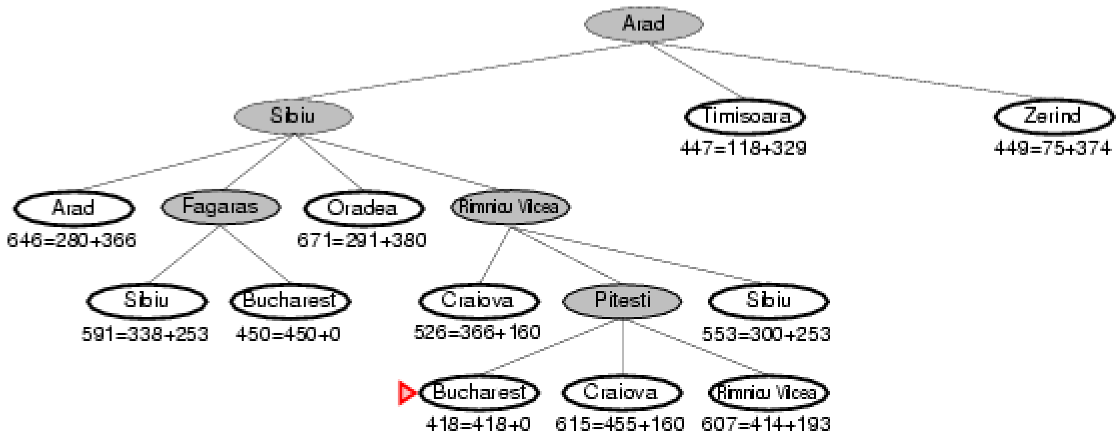


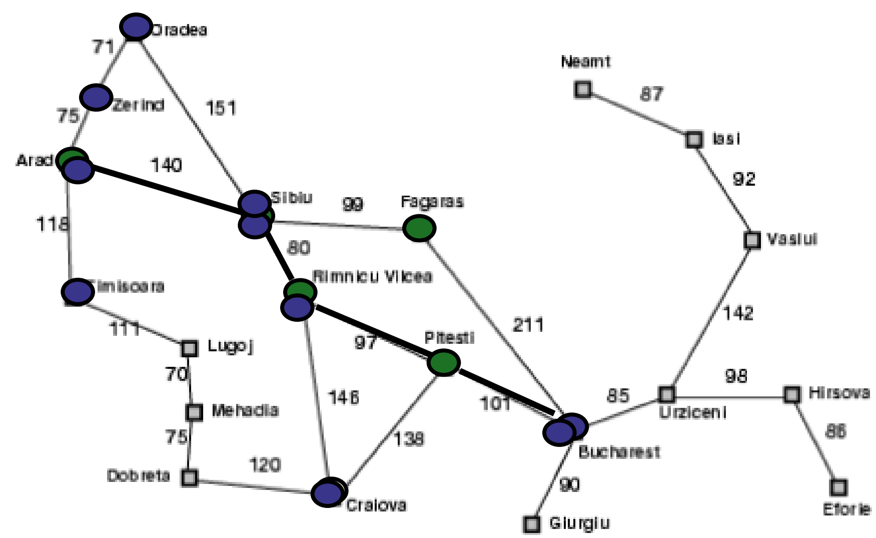
A* Tree Search Example



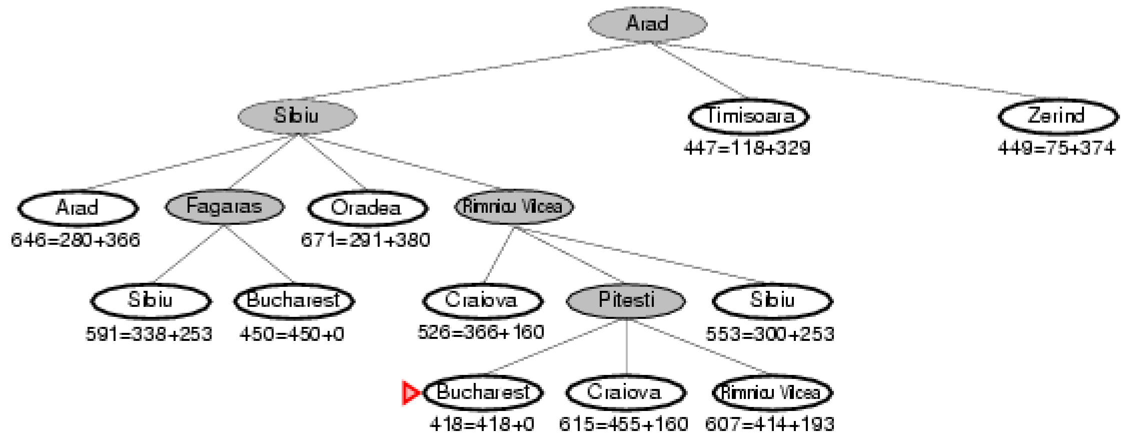


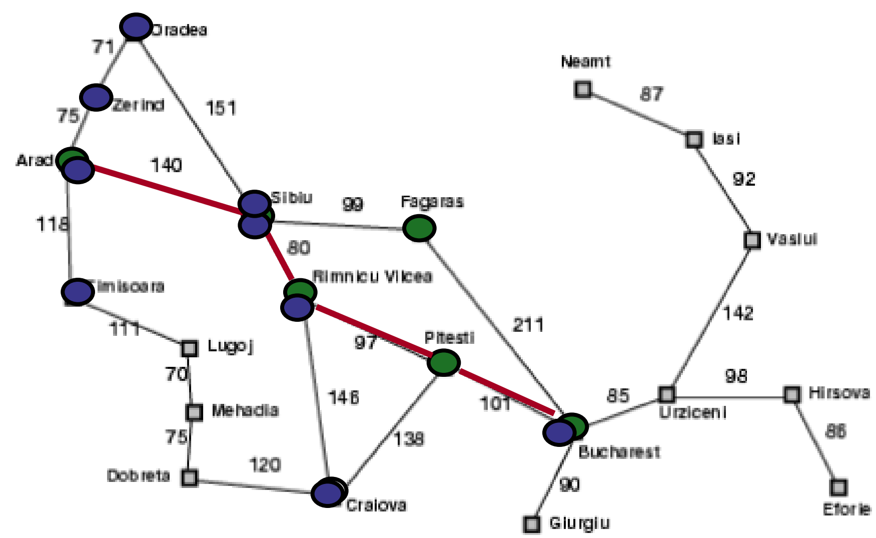
A* Tree Search Example





A* Tree Search Example

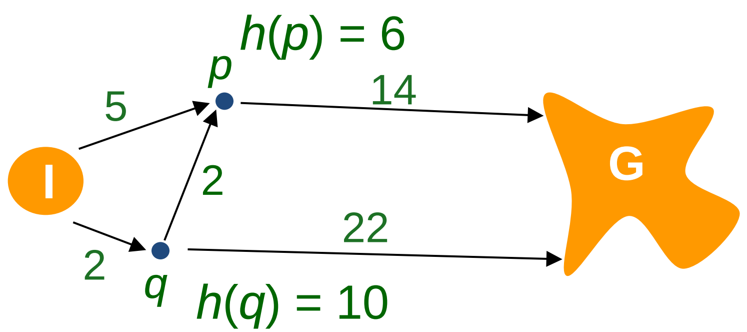




When *h*(*n*) underestimates the cost of reaching a goal (*h* is admissible), A* (tree search) is optimal

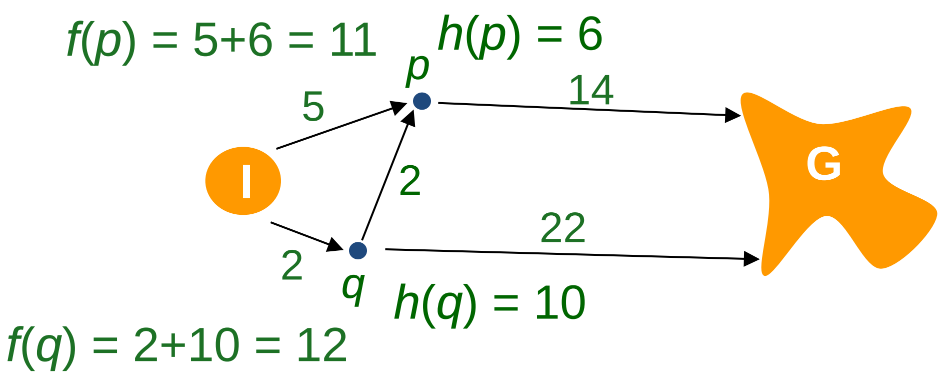
Duplicate Elimination

- A* graph search with admissible heuristic is not guaranteed to be optimal
- Example:



Duplicate Elimination

- A* graph search with admissible heuristic is not guaranteed to be optimal
- Example:

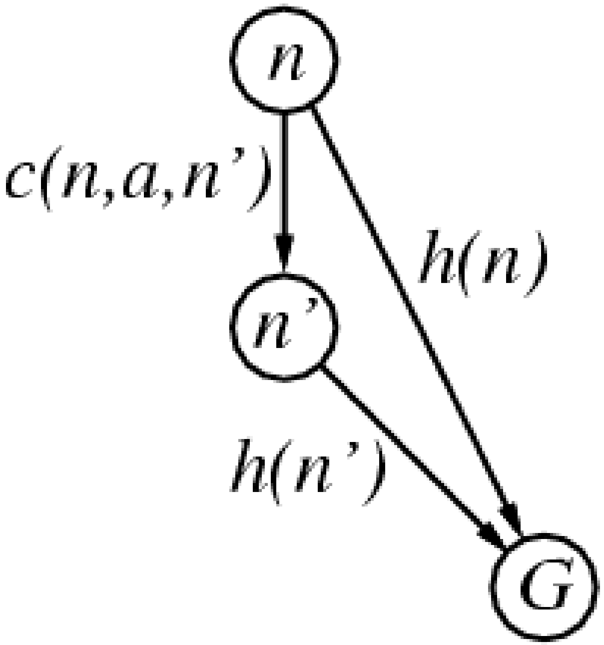


p is expanded before *q* and added to *explored*, but an optimal path with *g*=4 has not been found to *p* yet. Thus, A* overlooks the optimal path to *G*

Consistent Heuristics

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

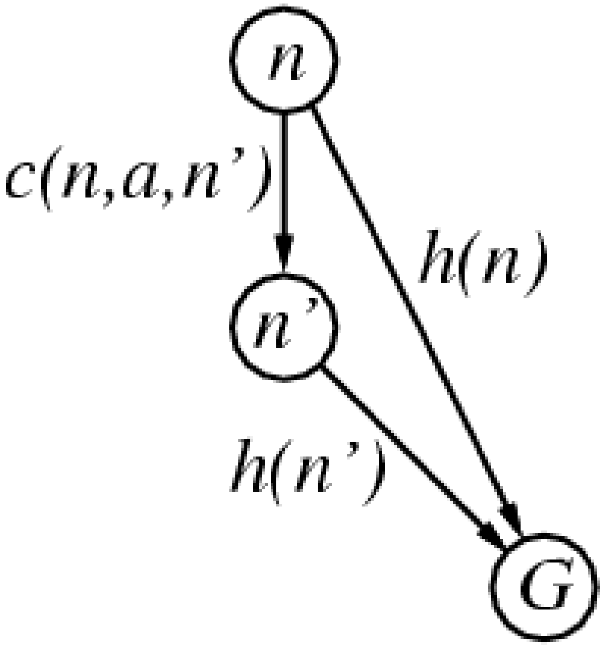
  $c(n,a,n') + h(n') \geq h(n)$



Consistent Heuristics

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

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



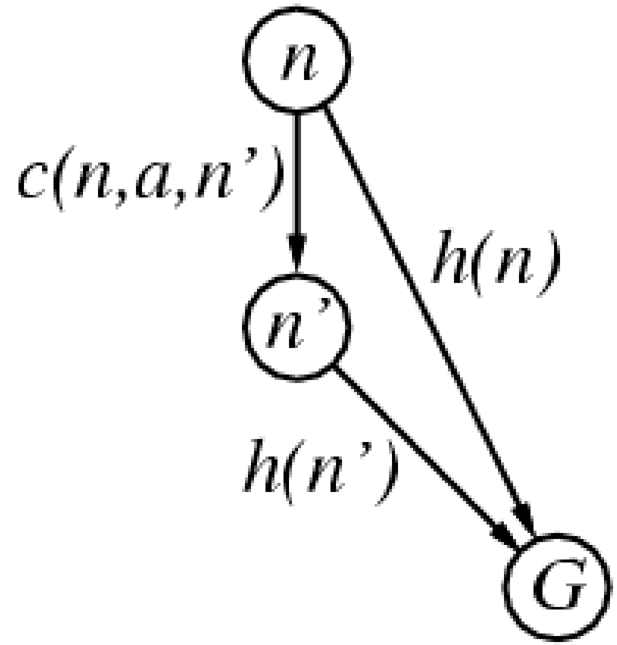
- If h is consistent, we have

  $$f(n') = g(n') + h(n')$$
  $$= g(n) + c(n,a,n') + h(n')$$
  $$\geq g(n) + h(n) = f(n)$$

Consistent Heuristics

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

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



- If h is consistent, we have

$$f(n') = g(n') + h(n')$$
$$= g(n) + c(n,a,n') + h(n')$$
$$\geq g(n) + h(n) = f(n)$$

- Thus, $f(n)$ is non-decreasing along any path

- A consistent heuristic is also admissible (exercise)

If h is consistent, A* graph search is optimal

Claim: When a node n is expanded, an optimal path to its state is found

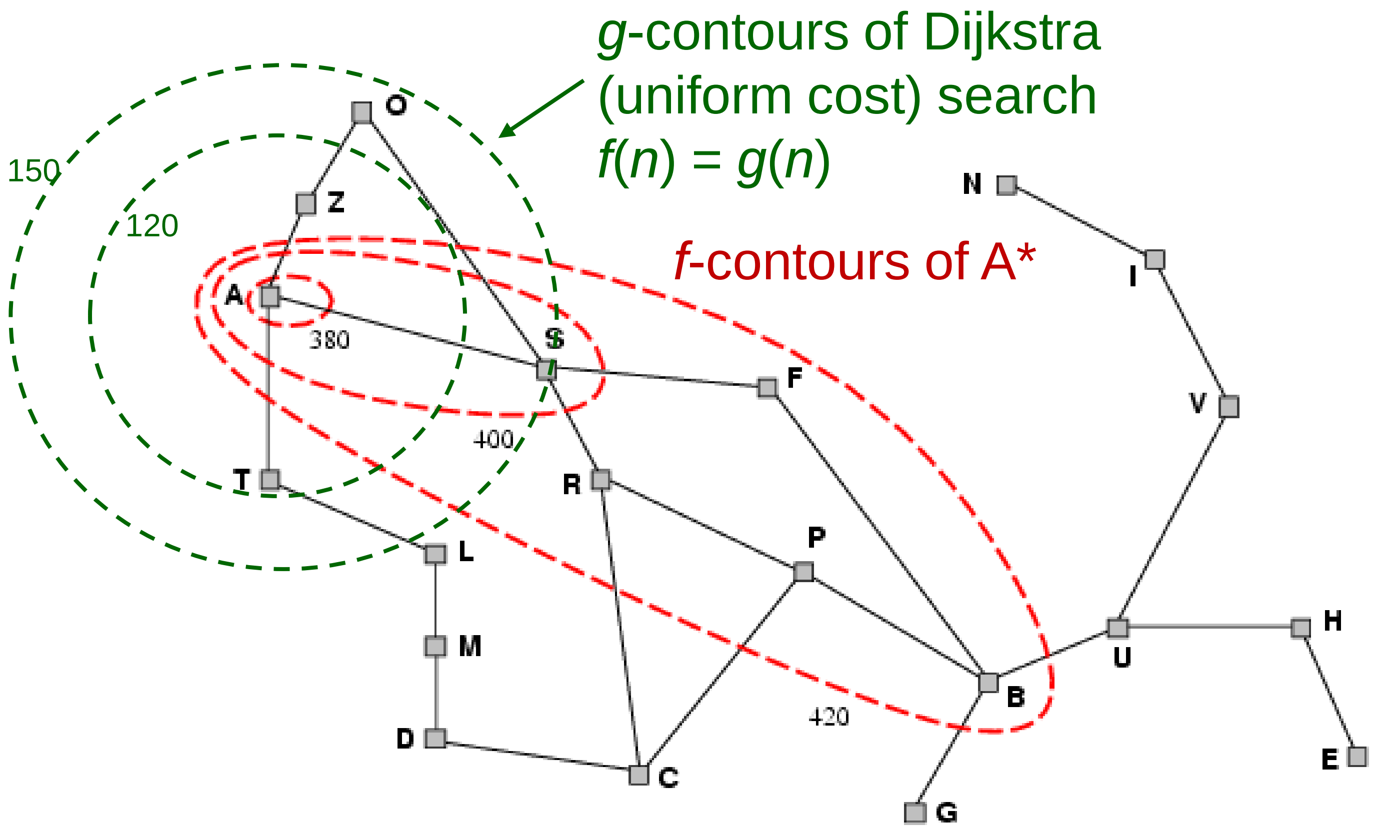
Proof:

Assume by contradiction the claim to be false then:

1. due to the separation property, a node n' on an optimal path to the state of n is on the frontier
2. We must have $f(n') > f(n)$ since n' otherwise would have been expanded before n

But then n' can never reach the state of n with lower cost, since f is increasing along the path

A* Graph Search with Consistent *h*



- A* graph search expands nodes of optimal paths in order of increasing *f*-value
- it expands all nodes with $f(n) < C^*$

Properties of A*

Consistent heuristic, graph-search version

- Complete? Yes, unless there are infinitely many nodes with $f(n) \leq C^*$
- Time? Exponential in solution length, unless h is very accurate $|h(n) - h^*(n)| \leq O(\log h^*(n))$
- Space? Keeps all nodes in memory
- Optimal? Yes, if h is admissible

Heuristics

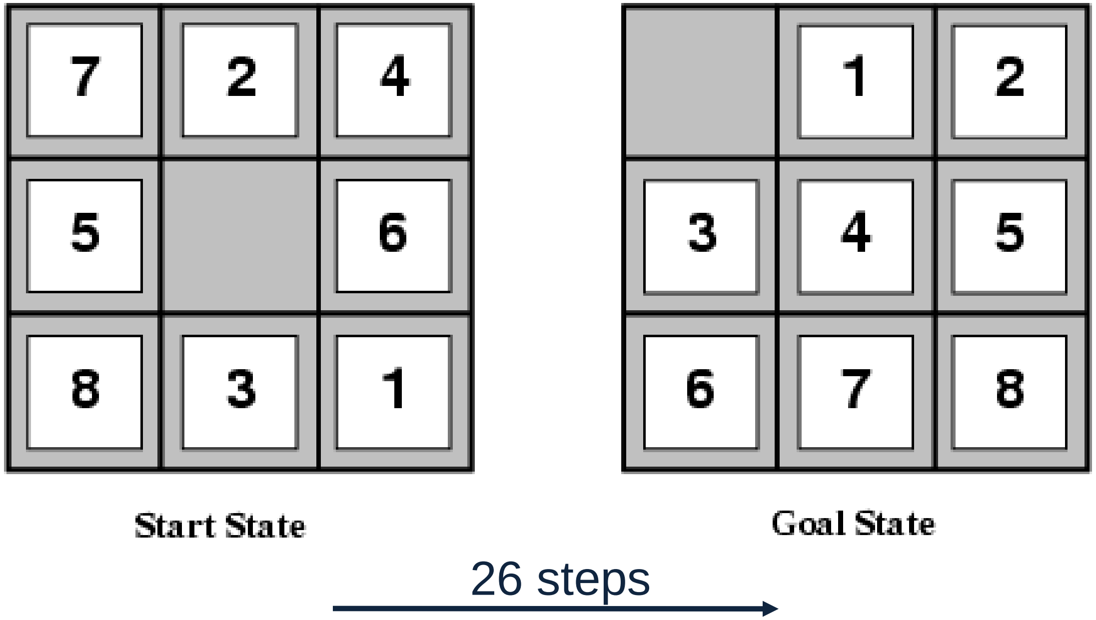
Admissible Heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = sum of Manhattan distances

$h_1(\text{Start}) = ?$

$h_2(\text{Start}) = ?$



Measuring Efficiency of Heuristics

- Efficiency measure: effective branching factor

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

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

Dominance

- If $h_2(n) \geq h_1(n)$ for all n then h_2 dominates h_1
- For A* graph search with a consistent heuristic, a reachable state s will be expanded if:

$$f(s) < C^*$$
$$\Leftrightarrow \quad g^*(s) + h(s) < C^*$$
$$\Leftrightarrow \quad h(s) < C^* - g^*(s)$$

- Thus, If h_2 dominates h_1 then A*(h_1) expands at least as many reachable states as A*(h_2)

Combining Heuristics

- Given k admissible heuristics $h_1(n), \ldots, h_k(n)$

- An admissible and dominating heuristic is

$$h_{max}(n) = max[h_1(n), \ldots, h_k(n)]$$

Relaxed Problems

Examples

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives an optimal solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives an optimal solution

Relaxed Problems

- A problem with fewer restrictions on the actions is called a relaxed problem (it adds edges to the state space)
- The cost of an optimal solution to a relaxed problem is:
  - an **admissible heuristic** of the original problem (more paths in state space $\Rightarrow$ optimal paths not longer)
  - a **consistent heuristic** to the original problem

Proof by contradiction:

If $c(n,a,n') + h(n') < h(n)$, then h is not associated with a optimal relaxed solution.