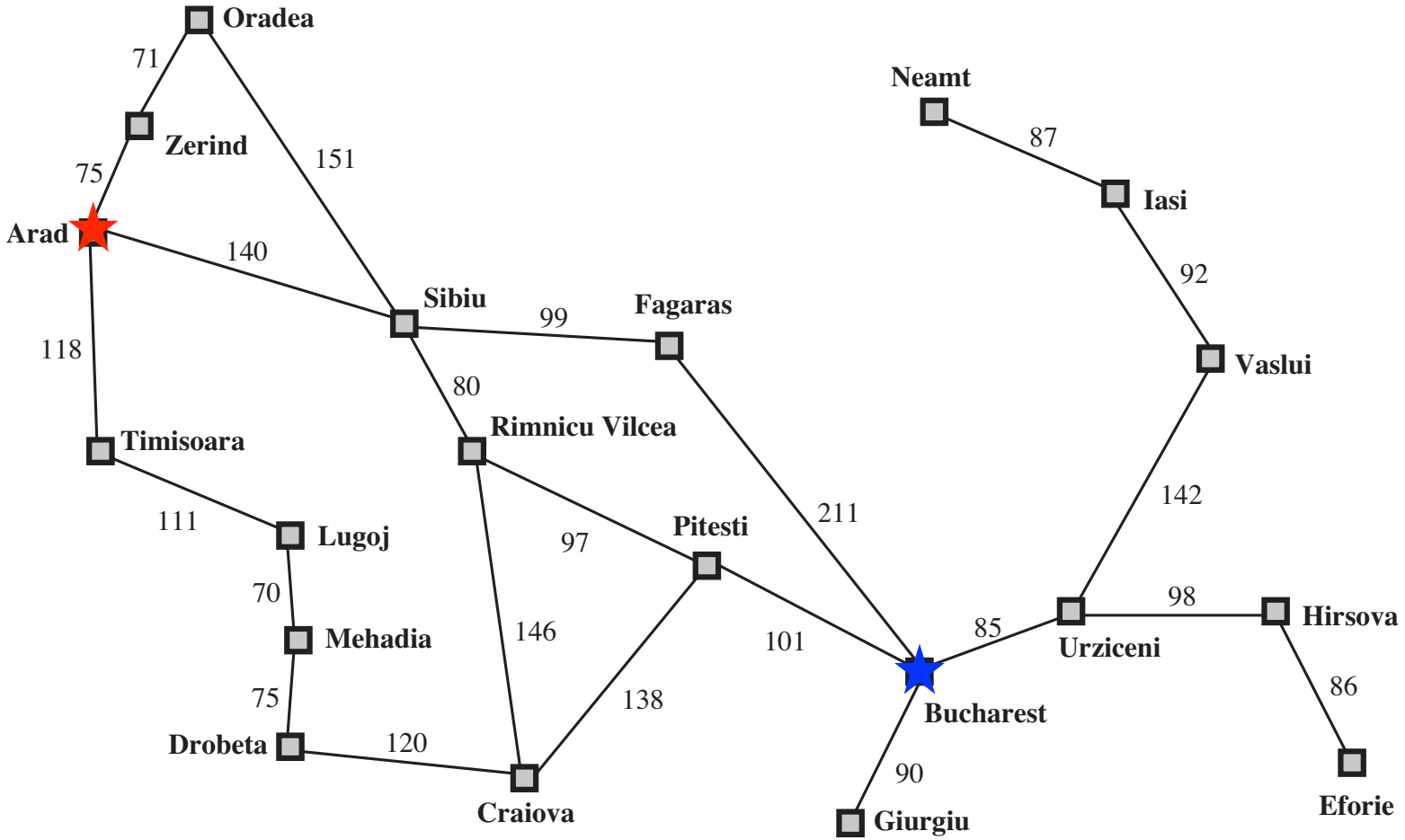
WPI

Artificial Intelligence
CS 534

Week 3

Travelling in Romania



Measuring problem-solving performance

- *Completeness*: Is the algorithm guaranteed to find a solution if there is one?
- *Optimality*: Does the strategy find the optimal solution (*i.e.* has the lowest path cost among all solutions) ?
- *Time complexity*: How long does it take to find a solution?
- *Space complexity*: How much memory is needed to perform the search?

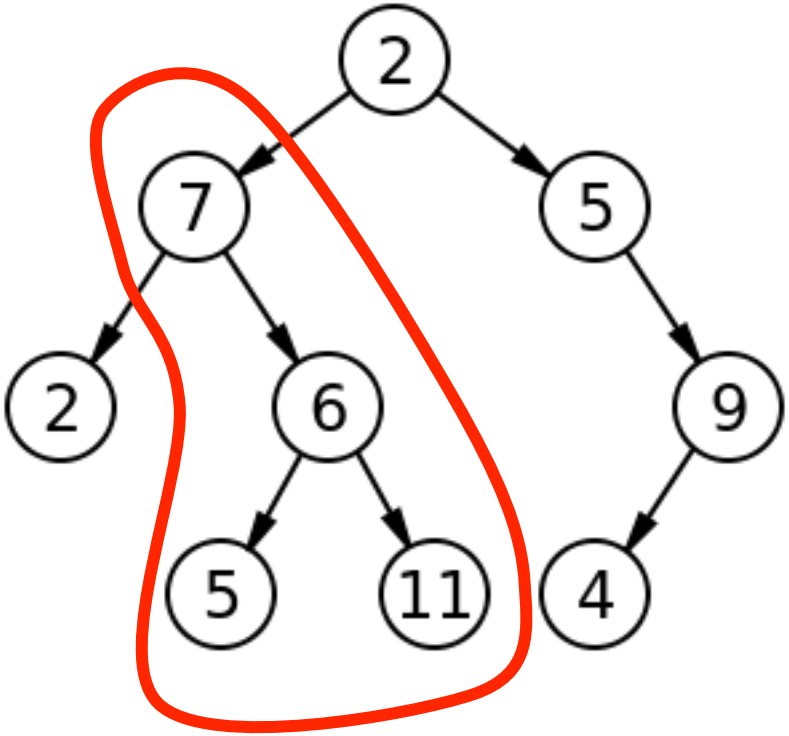
Uninformed search

- Aka the blind search: the strategy has no additional information about the states beyond that one provided in the problem definition
- *Breadth-first search*
- *Uniform-cost search*
- *Depth first search*
  - *Backtracking search*
- *Depth-limited search*
- *Iterative deepening depth-first search*
- *Bidirectional search*

All search strategies are distinguished by the order in which nodes are expanded

Tree terminology

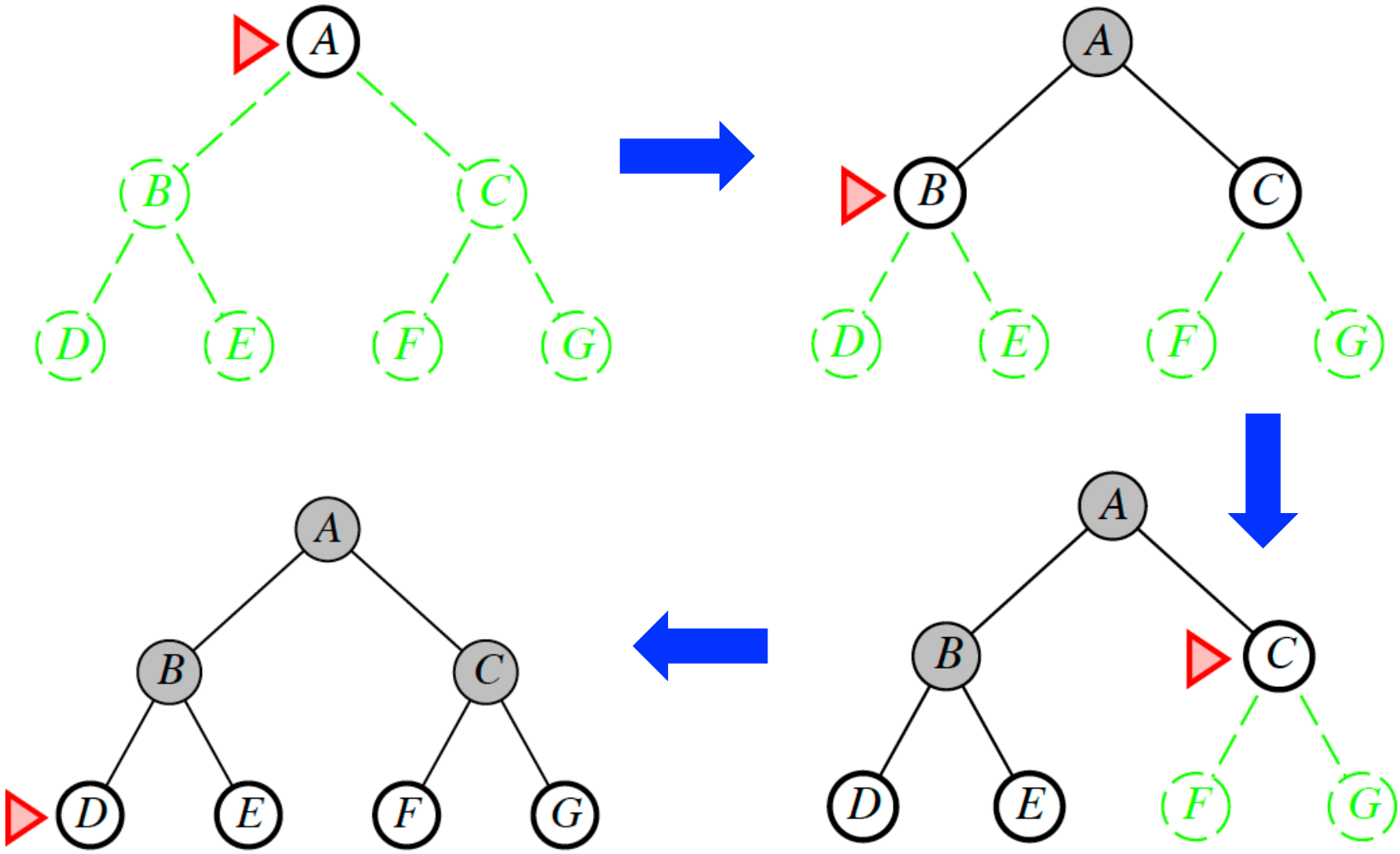
- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child
- *Level*: Depth
- *Internal nodes*: Parents and children
- *Branching factor*: N of children
- *Subtree*: A part of tree (a tree too)



So the complexity of a search algorithm may depend on

- b: maximum branching factor of the search tree
- d: depth of the least-cost (shallowest) solution
- m: maximum depth of the state space (can be infinite)

Breadth-first search: Method



Measuring performance of breadth-first search

- *Completeness*: Complete if b *is finite*
- *Optimality*: Optimal if cost is universally c (*e.g.* 1) per each step
- *Time complexity*: (1) Goal test is applied to each node when it is selected for expansion

$$b+b^2+b^3+\ldots+b^d+b(b^d\text{-}1)=\mathrm{O}(b^{d+1})$$

(2) Goal test is applied to each node when it is generated rather than when it is selected for expansion

$$b+b^2+b^3+\ldots+b^d=\mathrm{O}(b^d)$$

- *Space complexity*: $\mathrm{O}(b^{d+1})$—keeps every node in memory
  - A much bigger problem than time

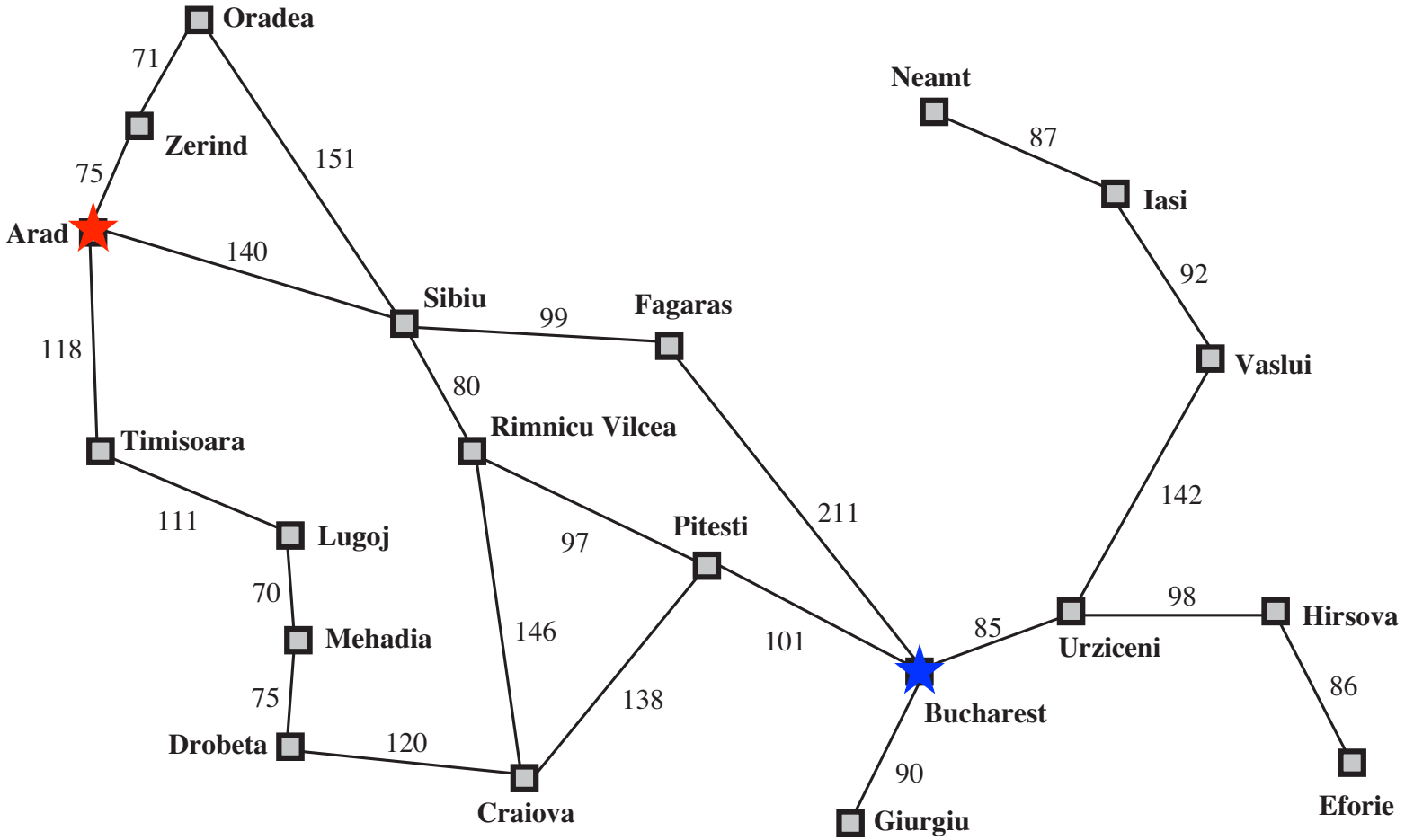
Time *vs.* memory requirements for BFS

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

Algorithm 2

BFS did not use the immediately available road length information



Uniform-cost search: Ideas

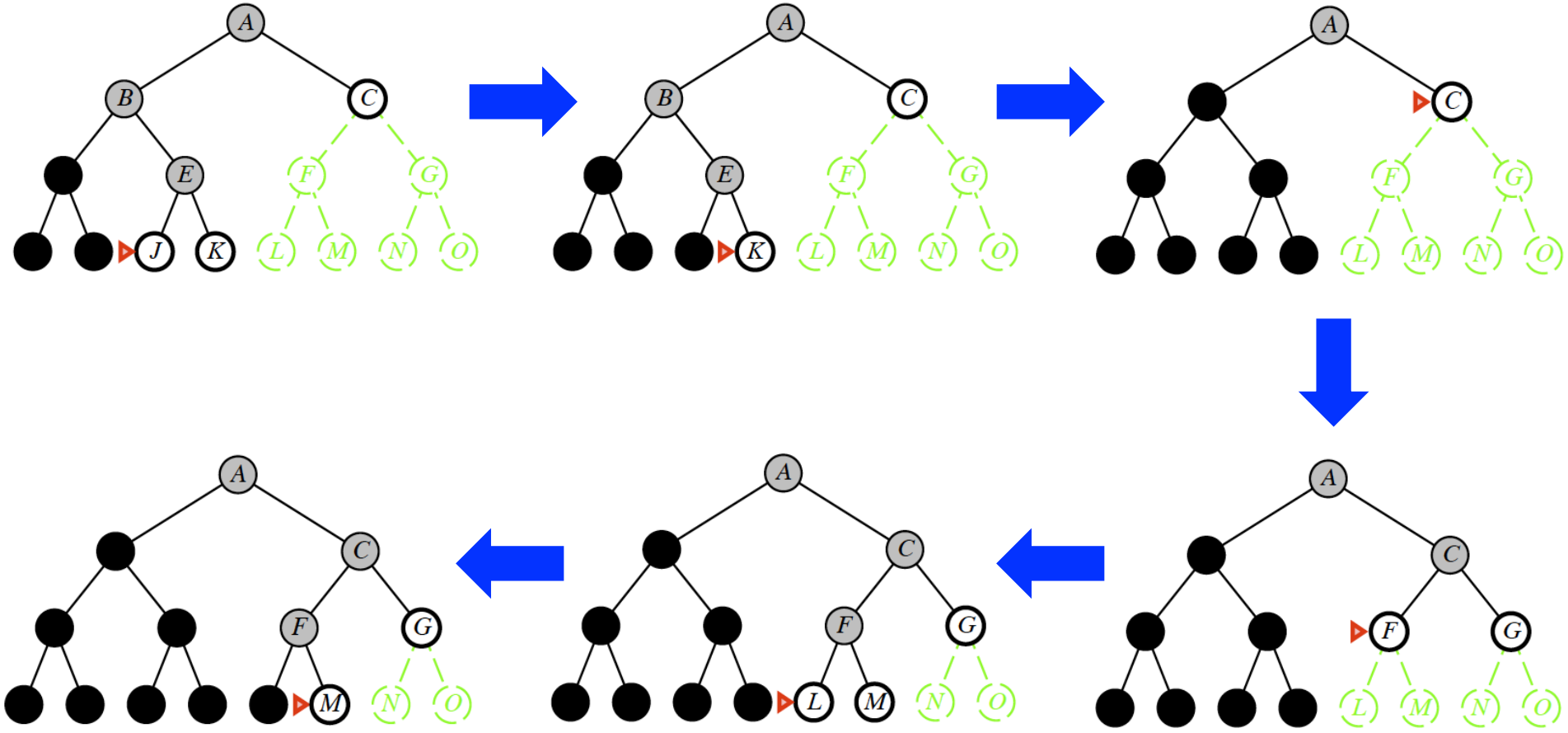
- Branches of the tree (steps) have different weights (step costs)
- Expand least-cost unexpanded node
- Equivalent to BFS if step costs all equal
- *Implementation*: queue ordered by path cost

Measuring performance of uniform-cost search

- *Completeness*: Complete if step cost is $\geq \varepsilon$
- *Optimality*: Yes: nodes are expanded in increasing order of $g(n)$
- *Time complexity*: Number of nodes with $g \leq \text{cost}$ of optimal solution, $O(b^{1+ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- *Space complexity*: Number of nodes with $g \leq \text{cost}$ of optimal solution, $O(b^{1+ceiling(C^*/\varepsilon)})$

Algorithm 3

Depth first search (DFS)



Measuring performance of DFS

- *Completeness*: No, fails in infinite-depth spaces (spaces with loops)
  - Needs to be modified to avoid repeated states along path
  - As a result: complete in finite spaces
- *Optimality*: No
- *Time complexity*: $O(b^m)$ only useful if m is much smaller than d
  - If solutions are dense, may be much faster than BFS
- *Space complexity*: $O(bm)$ — linear!!!

Algorithm 4

Backtracking search: Ideas

- Only one successor is generated at a time rather than all successors
- Each partially expanded node remembers which successor to generate next
- Generate a successor by modifying the current state description directly rather than copying it first
- However, we must able to “undo” each modification when we go back to generate the next successor

Measuring performance of backtracking

- *Completeness*: No, fails in infinite-depth spaces (spaces with loops)
  - Needs to be modified to avoid repeated states along path
  - As a result: complete in finite spaces
- *Optimality*: No
- *Time complexity*: $O(b^m)$
- *Space complexity*: $O(m)$ — linear, independent of b !!!

Algorithm 5

Depth limited search: Ideas

- Depth-first search with depth limit l, i.e., nodes at depth l have no successors
- Allows to avoid the failure of DFS in infinite state spaces
- Sometimes the depth limit can be based on the knowledge od the problem

Measuring performance of depth-limited search

- *Completeness*: No—Introduces an additional source of incompleteness, when $l<d$: the shallowest goal is beyond the depth limit

- *Optimality*: No

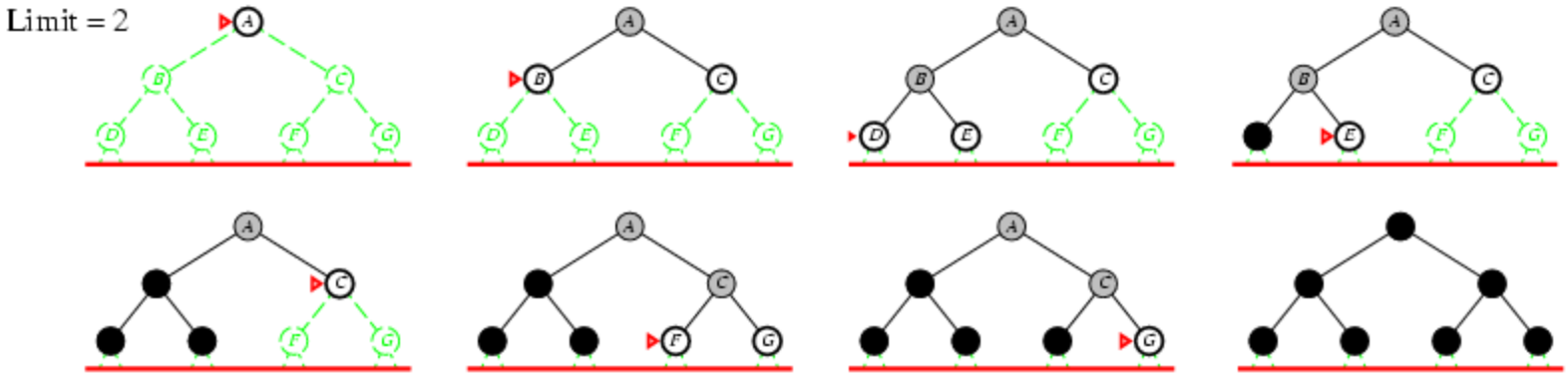
- *Time complexity*: $O(b^l)$ only useful if l is much smaller than d
  - If solutions are dense, may be much faster than BFS

- *Space complexity*: $O(bl)$ — linear!!!

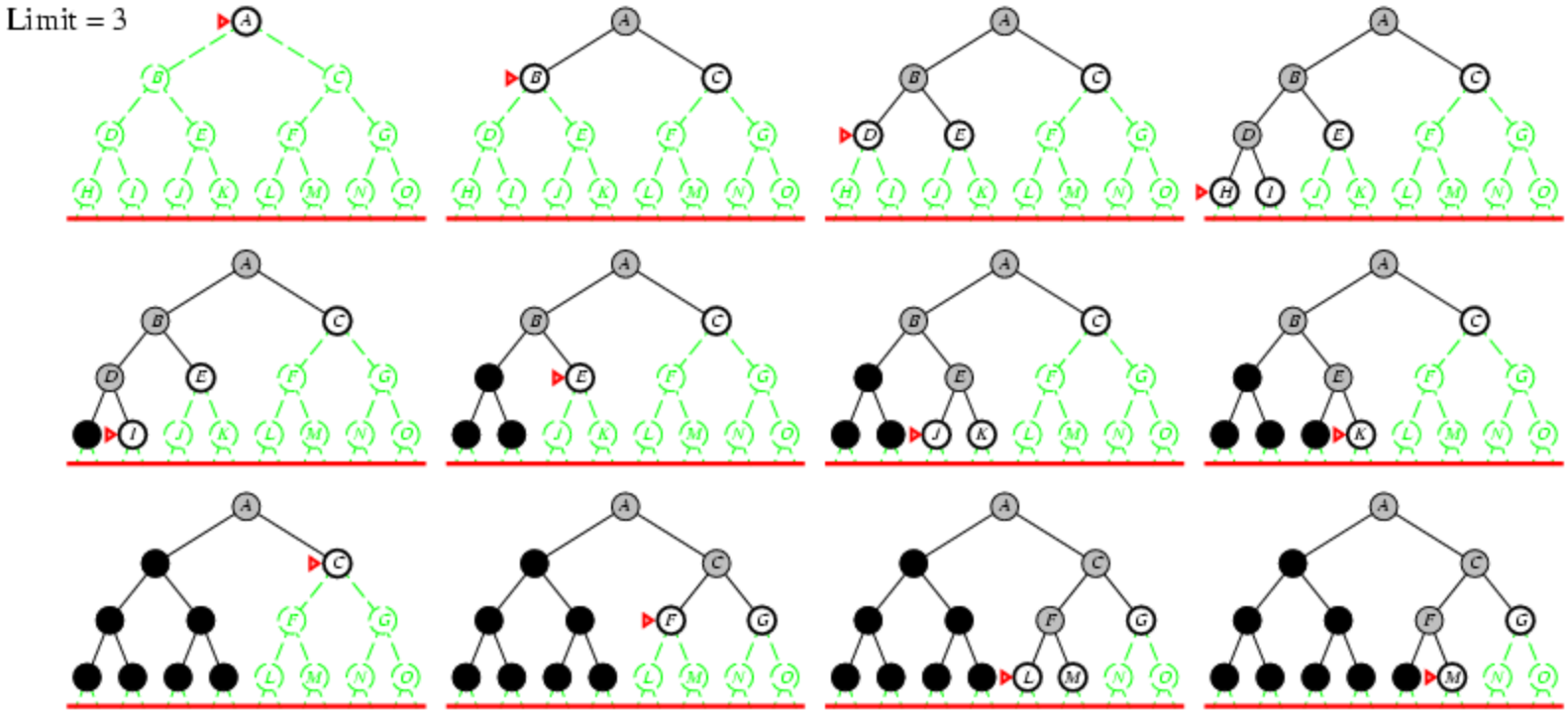
Algorithm 6

Iterative deepening DFS

Iterative deepening DFS



Iterative deepening DFS

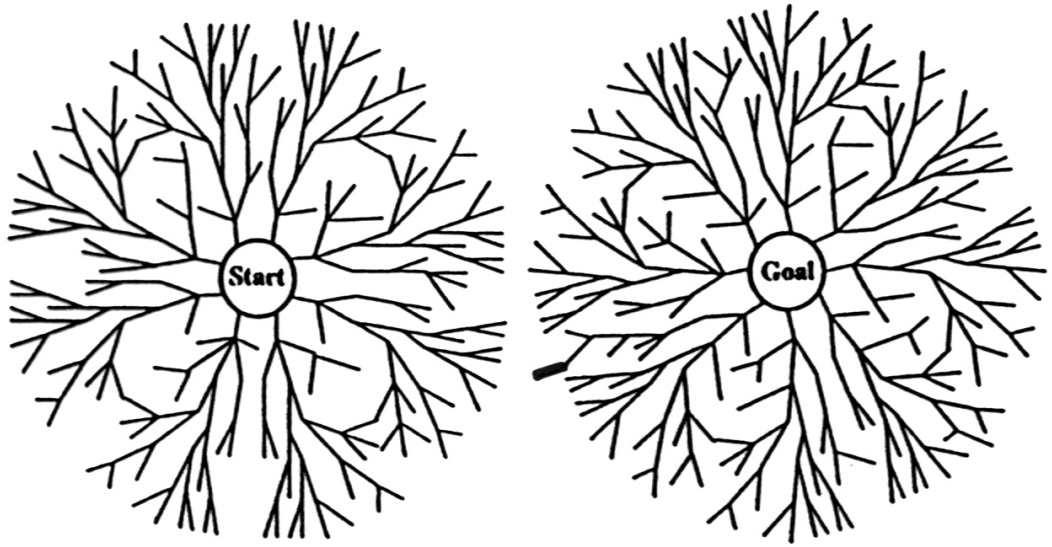


Measuring performance of iterative deepening DFS

- *Completeness*: Yes
- *Optimality*: *Yes if step cost = 1*
- *Time complexity*: $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- *Space complexity*: $O(bd)$ — linear!!!

Let’s continue: Informed search

Bidirectional search



A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Motivation: $b^{d/2} + b^{d/2} << b^{d}$

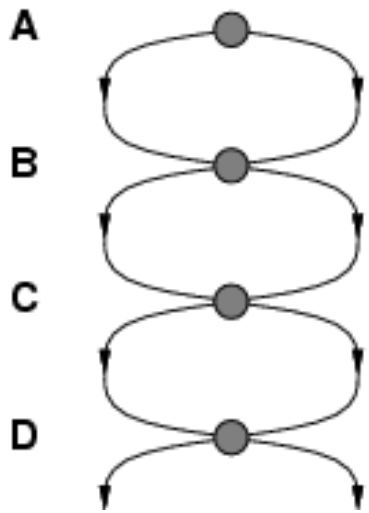
Summary of algorithms

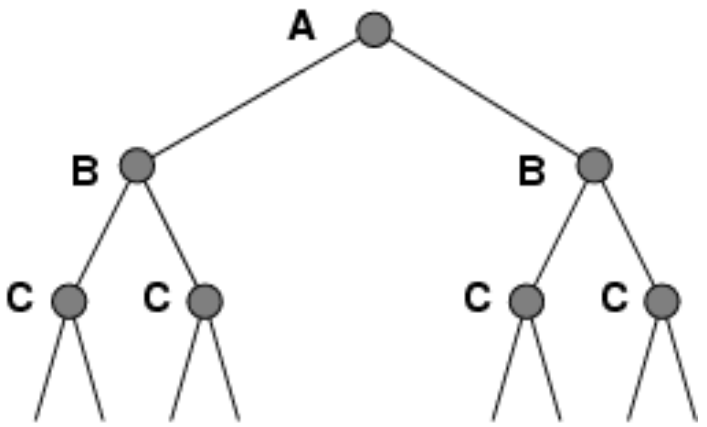
Criterion	BFS	Uniform-Cost	DFS	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+C^*/\varepsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+C^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Generally the preferred uninformed search strategy

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon$ for positive
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search

Repeated states





Need to be able to detect repeated states in order to avoid exponential time complexity:

- Do not return to the state you just came from. Have the expand function (or the operator set) refuse to generate any successor that is the same state as the node's parent.
- Do not create paths with cycles in them. Have the expand function (or the operator set) refuse to generate any successor of a node that is the same as any of the node's ancestors.
- Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory, resulting in a space complexity of $O(b^d)$, potentially. It is better to think of this as $O(s)$, where s is the number of states in the entire state space.

Material

- Reading Chapter 3 (Section 3.5), Chapter 4

Informed algorithms

- Best-first search
- Greedy best-first search
- A* search
- RBFS
- SMA
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms
- AND-OR Search trees
- Searching with no observations
- Online search

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - Estimate of "desirability"
  - Expand most desirable unexpanded node

- Implementation:

  Order the nodes in frontier in decreasing order of desirability

- Special cases:
  - Greedy best-first search
  - A* search

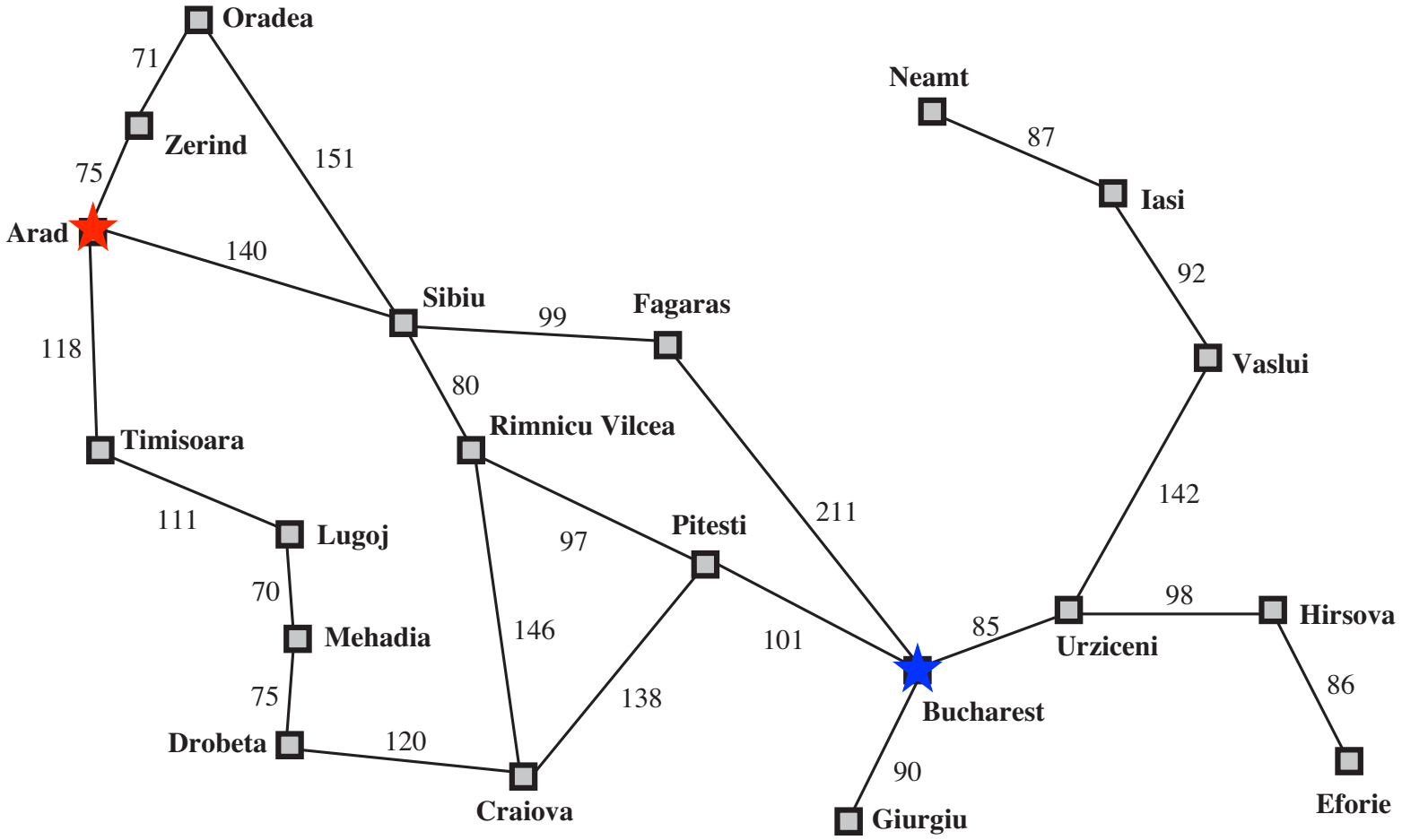
Greedy best-first search: Ideas

- Tries to expand the node that appears closest to the goal first, assuming that it leads to solution quickly

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from n to *goal*

- Example:
  $h_{SLD}(n)$ = straight-line distance from n to Bucharest

Traveling in Hungary



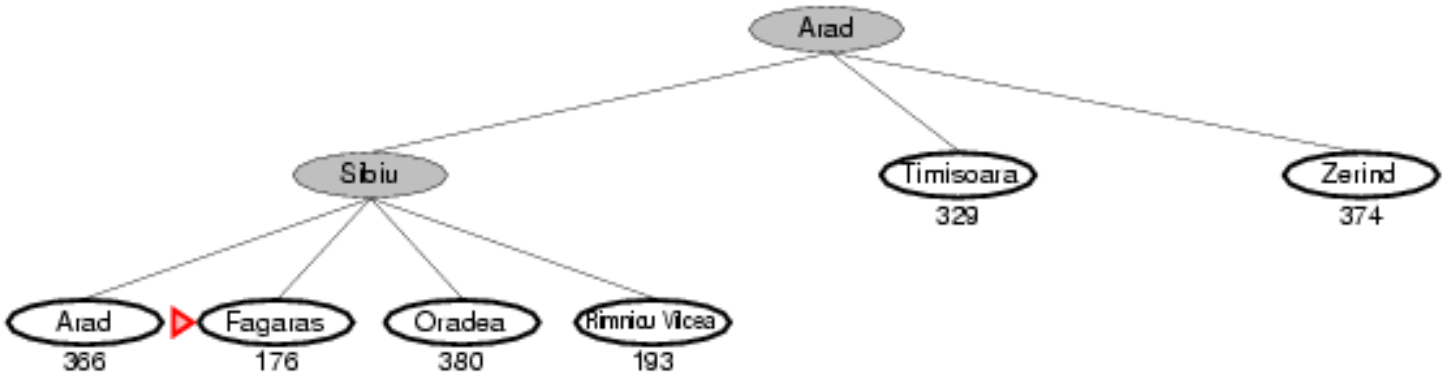
Romania with step costs in km: Straight line distance to Bucharest

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

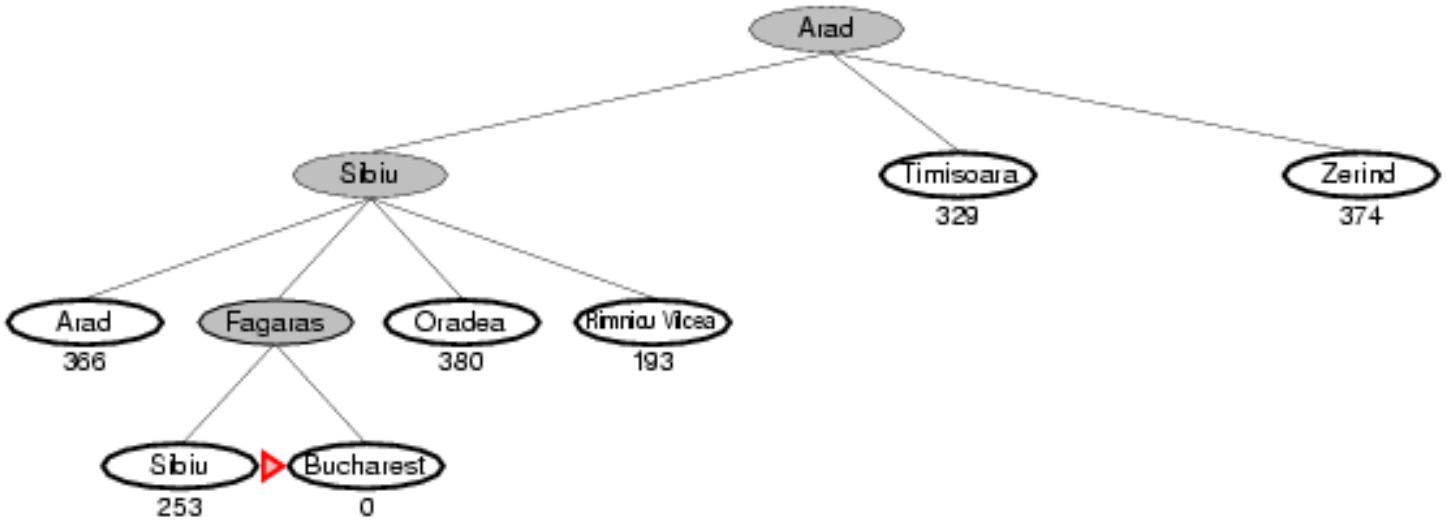
Greedy best-first search example: The initial state

Greedy best-first search example: After expanding Arad

Greedy best-first search example: After expanding Sibiu



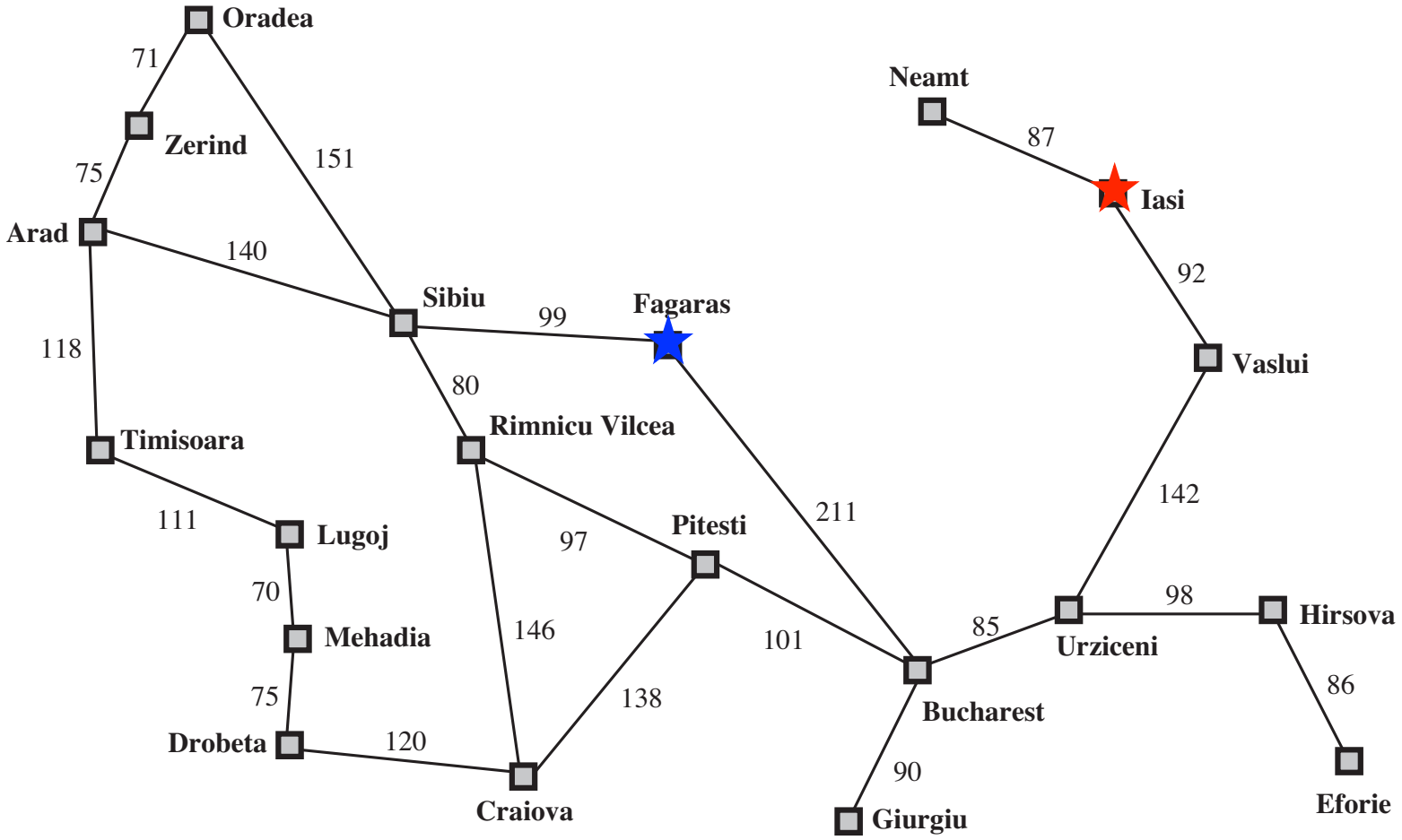
Greedy best-first search example: After expanding Fagaras



Measuring performance of greedy best-first search

- *Completeness*: No – can get stuck in loops, *e.g.*
  Iasi → Neamt → Iasi → Neamt →
- *Optimality*: No, since there might be a shorter path
- *Time complexity*:
- *Space complexity*:

Traveling in Hungary



Measuring performance of greedy best-first search

- *Completeness*: No – can get stuck in loops, *e.g.*
  Iasi → Neamt → Iasi → Neamt →
- *Optimality*: No, since there might be a shorter path
  Arad → Sibiu→ Rimnicu Vilscea→ Pitesti→ Bucharest is 32 km shorter
- *Time complexity*:
- *Space complexity*:

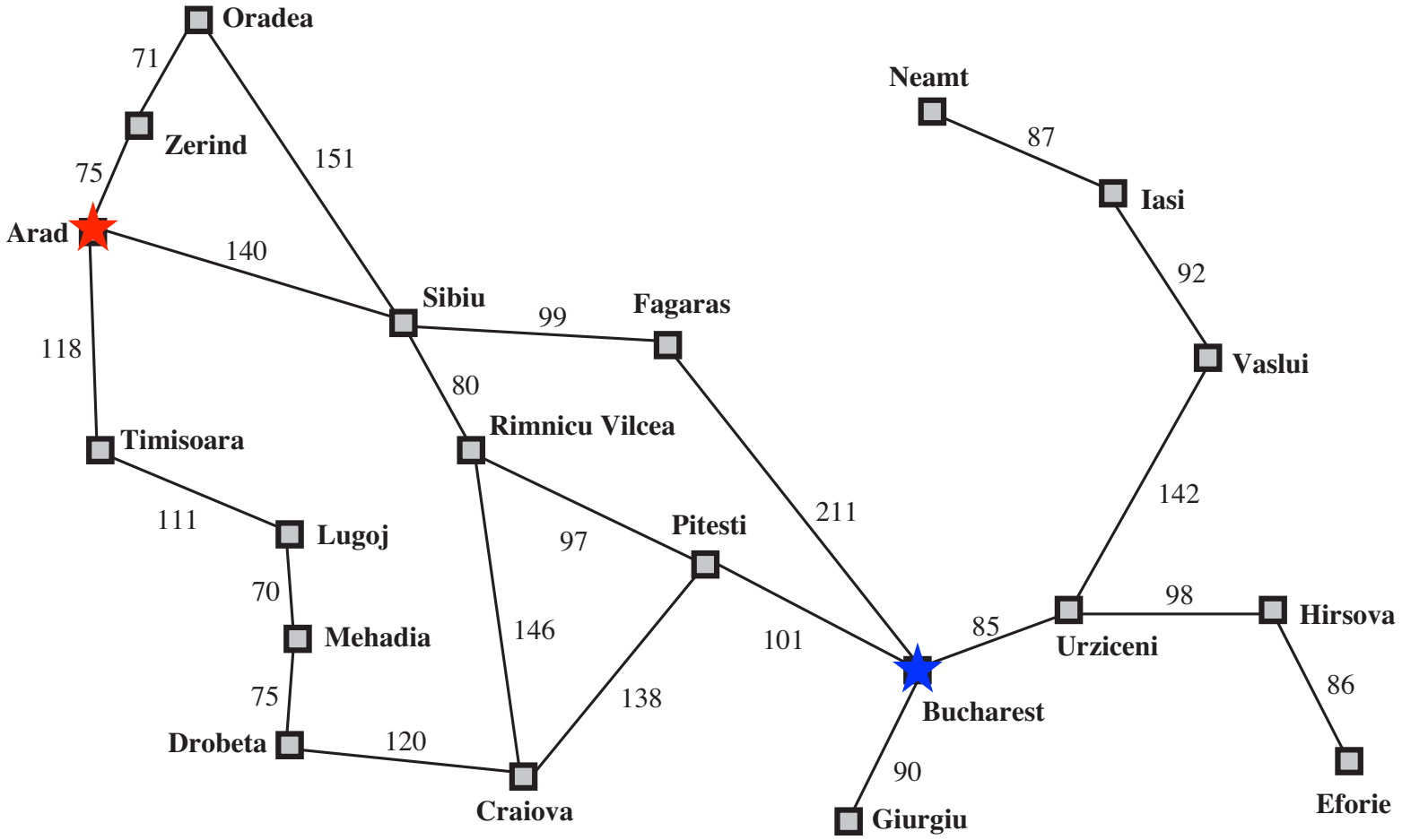
Measuring performance of greedy best-first search

- *Completeness*: No – can get stuck in loops, *e.g.*
  Iasi → Neamt → Iasi → Neamt →
- *Optimality*: No, since there might be a shorter path
  Arad → Sibiu→ Rimnicu Vilscea→ Pitesti→ Bucharest is 32 km shorter
- *Time complexity*: *$O(b^m)$ — might be big, but a good heuristic can give dramatic improvement*
- *Space complexity*: *$O(b^m)$ — keeps all nodes in memory*

A* search: Ideas

- Most widely known form of informed search
- A* tries to avoid expanding paths that are already too expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = path cost from the start node to node n
  - $h(n)$ = estimated cost of the cheapest path from n to goal
  - $f(n)$ = estimated total cost of path through n to goal
- Algorithm is identical to Uniform-Cost-Search, except using $g(n) + h(n)$ instead of $g(n)$

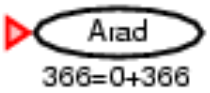
Traveling in Hungary



Romania with step costs in km: Straight line distance to Bucharest

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

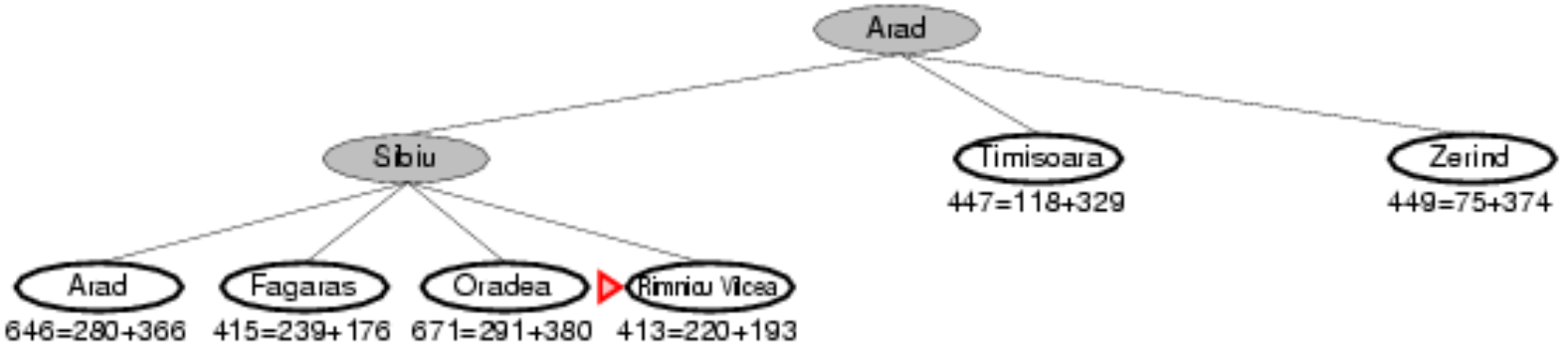
A* search example: Traveling in Hungary



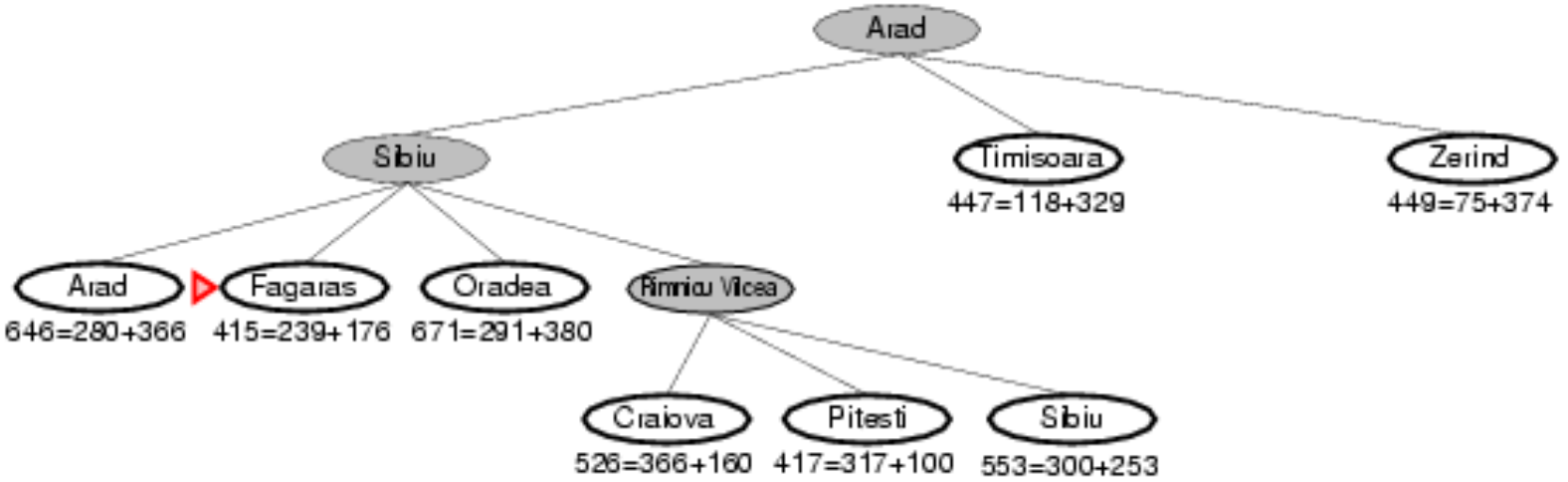
A* search example: Traveling in Hungary

A* search example: Expanding Arand

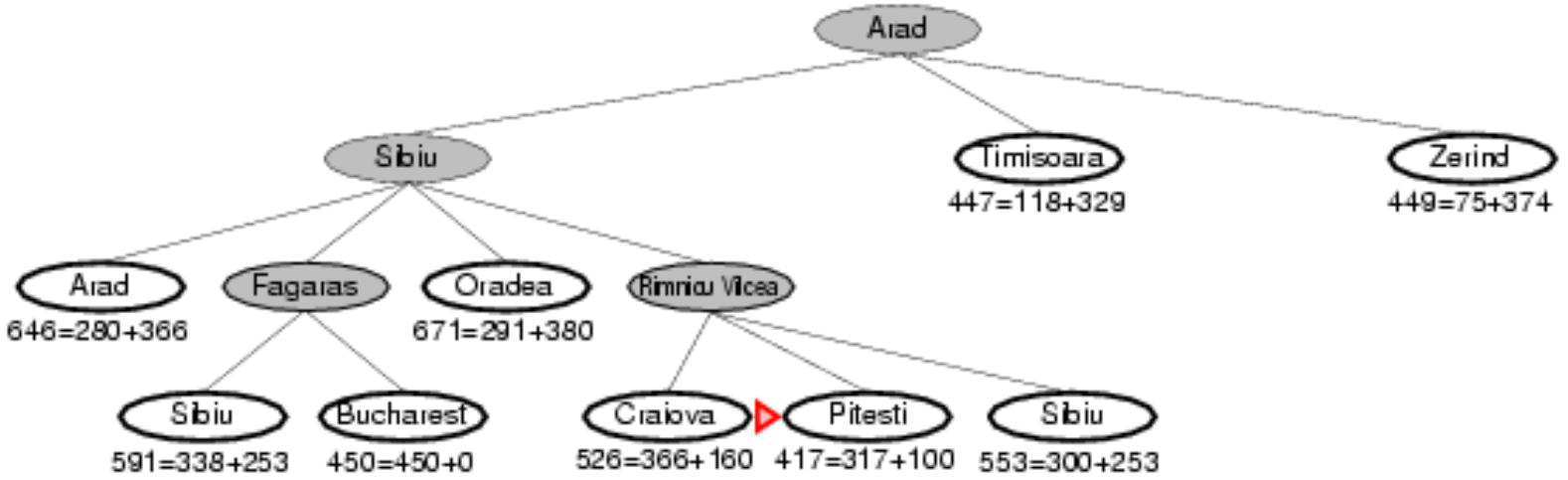
A* search example: Expanding Sibiu



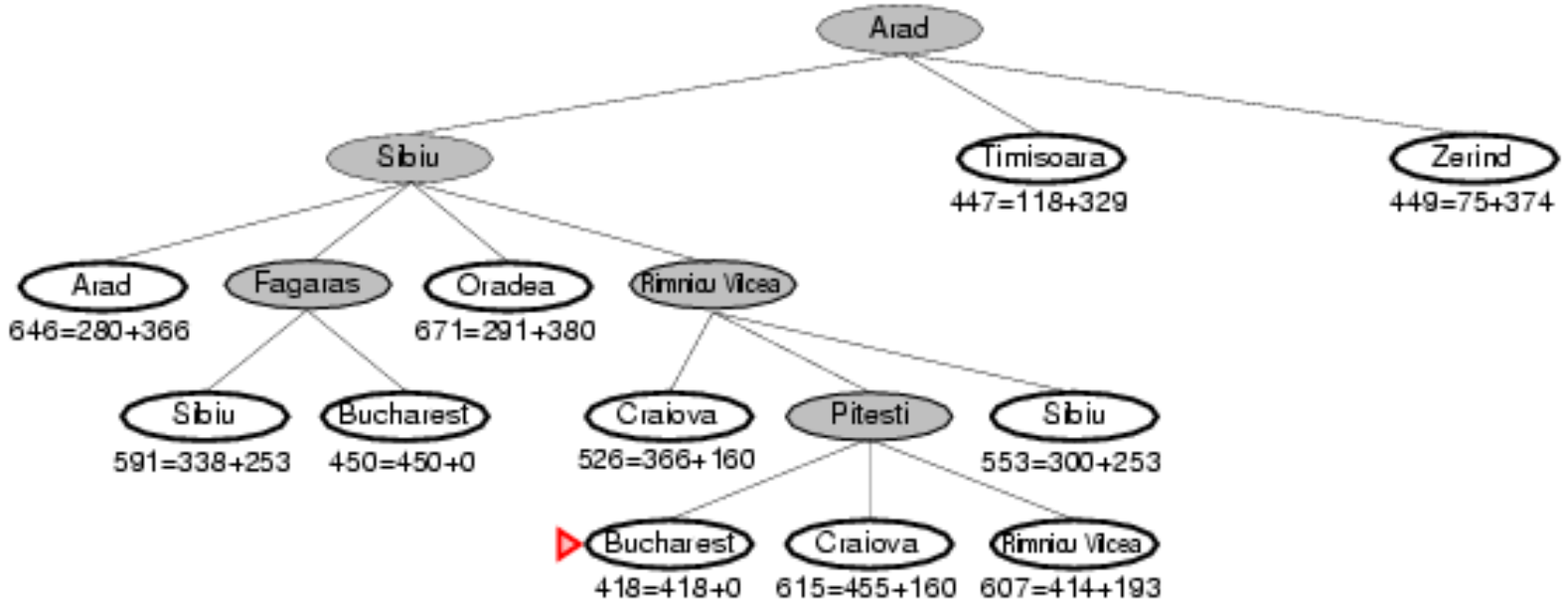
A* search example: Expanding Rimnicu Vilcea



A* search example: Expanding Fagaras



A* search example: Expanding Pitesti



Conditions for optimality: Admissible heuristics

Definition: A heuristic $h(n)$ is admissible if for every node n: $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n

- Thus, an admissible heuristic is optimistic: it never overestimates the cost to reach the goal

Conditions for optimality: Admissible heuristics

Definition: A heuristic $h(n)$ is admissible if for every node n: $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n

- Thus, an admissible heuristic is optimistic: it never overestimates the cost to reach the goal
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

Conditions for optimality: Admissible heuristics

Definition: A heuristic $h(n)$ is admissible if for every node n: $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n

- Thus, an admissible heuristic is optimistic: it never overestimates the cost to reach the goal
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

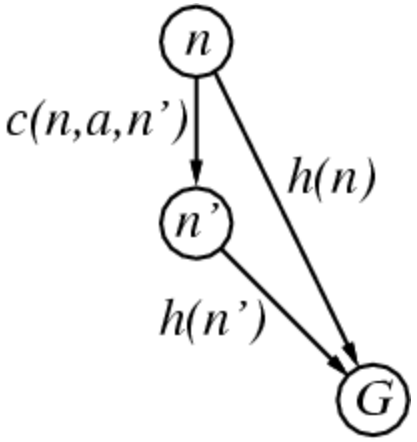
Theorem: If $h(n)$ is admissible, then A* using `TREE-SEARCH` algorithm is optimal

Consistent heuristics

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

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

- 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 non-decreasing along any path.

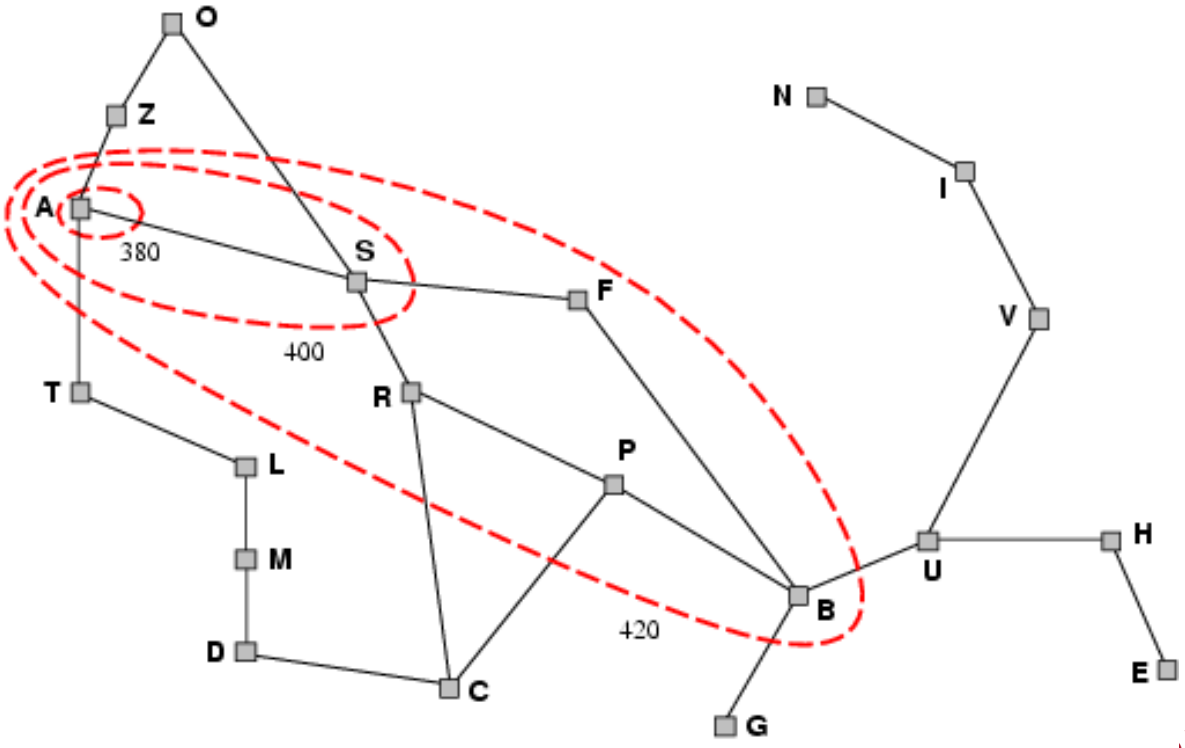
Theorem: If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

Consistent heuristics

- Consistency is required only for applications of A* to graph search
- It is a slightly stronger condition than admissibility: It is not difficult to show that every consistent heuristic is also admissible
- It is a form of general triangle inequality
- Example: H_{SLD} is a consistent heuristic

Optimality of A*

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



Measuring performance of greedy best-first search

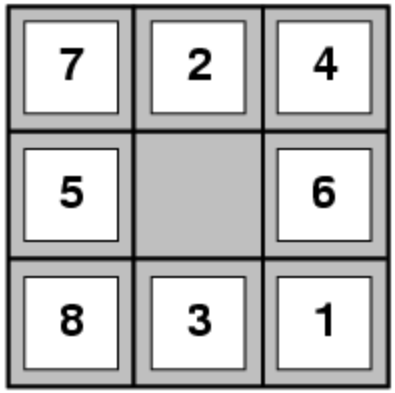
- *Completeness*: Yes
  (unless there are infinitely many nodes with f ≤ f(G))
- *Optimality*: Yes
- *Time complexity*: *Exponential*
- *Space complexity*: *keeps all nodes in memory*

Admissible heuristics

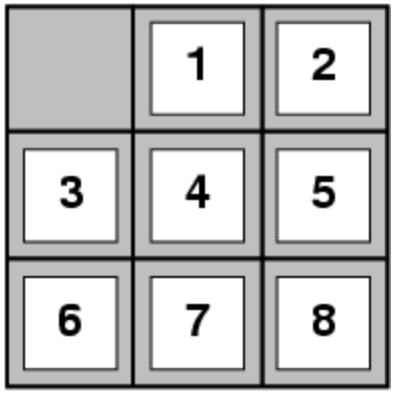
E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (sum of horizontal and vertical distance, or N of squares from desired location of each tile)

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



Start State

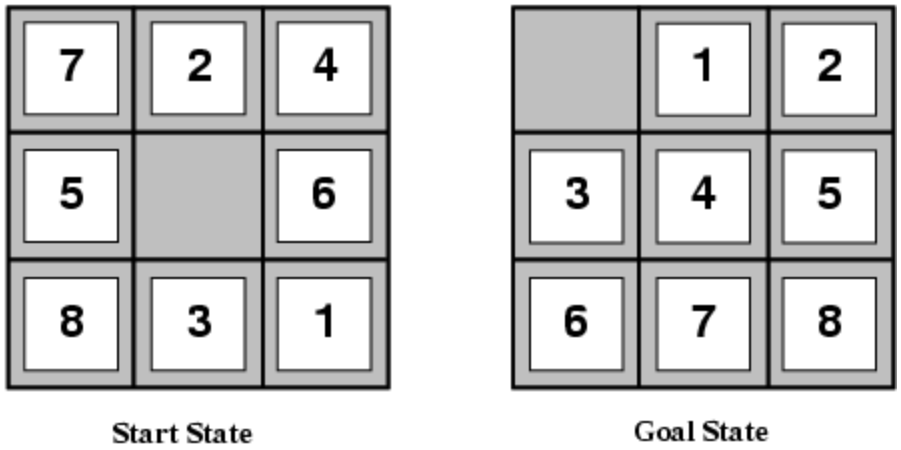


Goal State

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (sum of horizontal and vertical distance, or N of squares from desired location of each tile)



- $h_1(S) =$ 8
- $h_2(S) =$ 3+1+2+2+2+3+3+2 = 18

Admissible heuristics

- $h_1(n)$ is admissible heuristics because any tile that is out of place must be moved at least once
- $h_2(n)$ is admissible heuristics because all any move can do is move one step closer to the goal
-

Admissible heuristics

- $h_1(n)$ is admissible heuristics because any tile that is out of place must be moved at least once
- $h_2(n)$ is admissible heuristics because all any move can do is move one step closer to the goal

Question: How to characterize the quality of a heuristic?

- Effective branching factor b^*

Definition: Assume the total number of nodes generates by A* for a problem is N, and the solution depth is d, then b^* is defined to be a branching factor that a uniform tree of depth d would have in order to contain $N+1$ nodes

Therefore: $N+1 = 1+b^*+(b^*)^2+(b^*)^3+\ldots+(b^*)^d$

Admissible heuristics

	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

Dominance

Definition: If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1

- h_2 is better for search

- Typical search costs (average number of nodes expanded):
  - *d=12* IDS = 3,644,035 nodes
    $A^*(h_1)$ = 227 nodes
    $A^*(h_2)$ = 73 nodes
  - *d=24* IDS = too many nodes
    $A^*(h_1)$ = 39,135 nodes
    $A^*(h_2)$ = 1,641 nodes

Admissible heuristics from relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original 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

Admissible heuristics from relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original 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

Admissible heuristics from relaxed problems

Problem: What if we have heuristics $h_1(n)$, $h_2(n)$, …, $h_m(n)$, and no heuristic dominates any other? Which one do we select?

Answer: Select $h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$

- Composite heuristics
- Uses whichever function is most accurate on the node in question

Local search algorithms

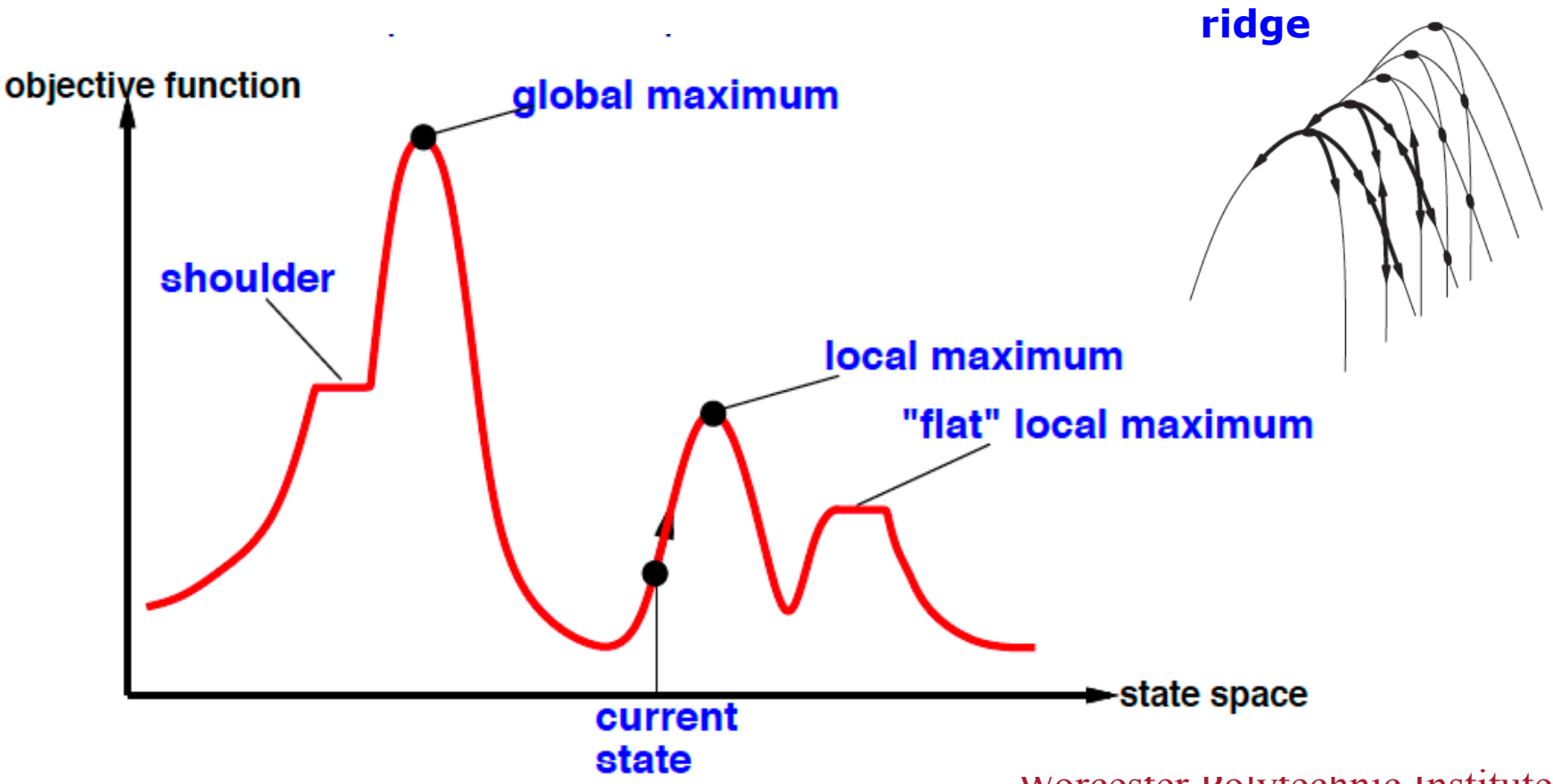
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
  - Find optimal configuration
  - Find configuration satisfying constraints
- In such cases, we can use local search algorithms
  - Iterative improvement algorithms
  - Keep a single "current" state, try to improve it
  - Use a single current node (instead of the path) and move to a neighbor of that node
  - Constant space, suitable for online as well as offline search

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima



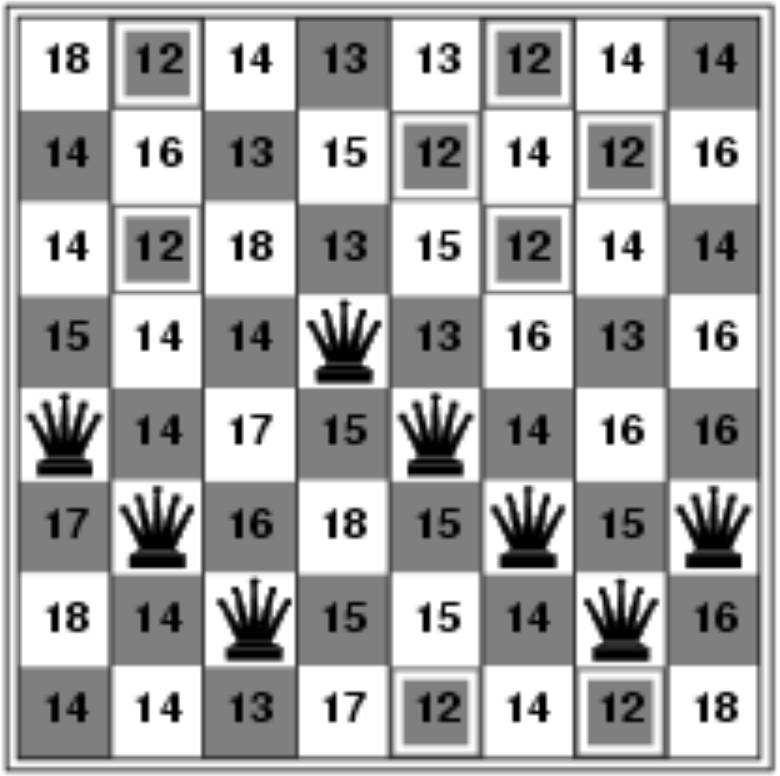
Hill-climbing search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

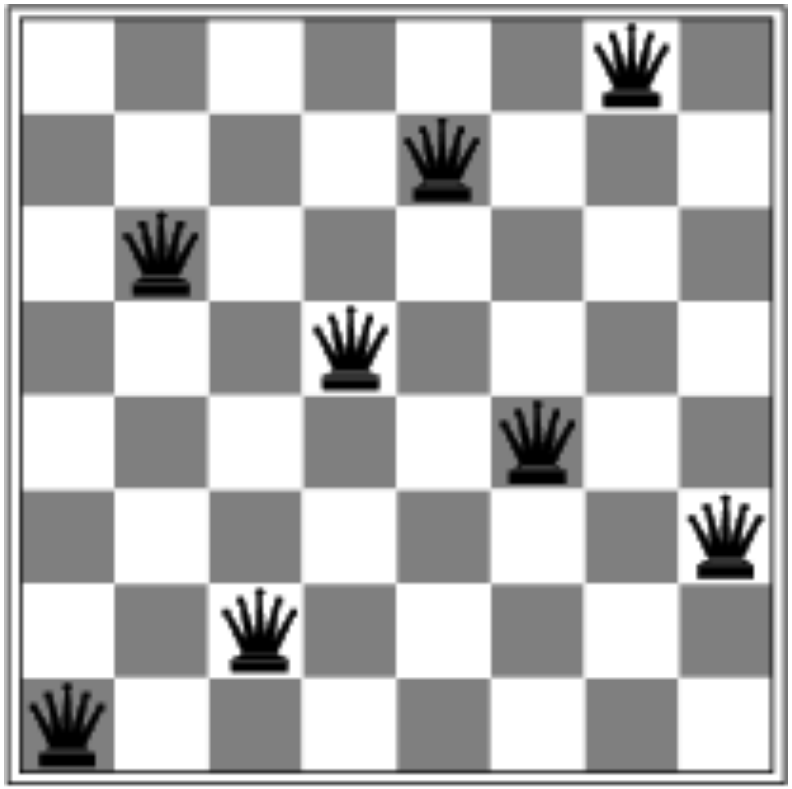
- A greedy local search
- Is simply a loop that continuously moves in the direction of increasing value (uphill)
- It terminates when it reaches a peak, where no neighbor has the higher value

Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- The global minimum is zero (at perfect solutions)
- *h* = *17* for the above state

Hill-climbing search: 8-queens problem



- A local minimum with $h = 1$

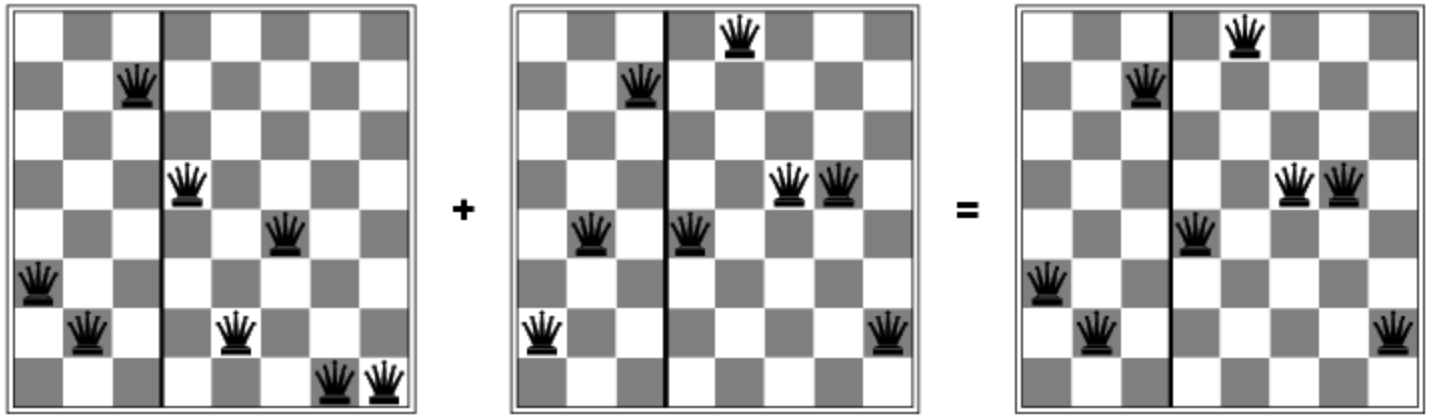
Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

Genetic algorithms

- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic algorithms



Genetic Algorithm

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```