TDT4136 Introduction to Artificial Intelligence

Lecture 3 : Solving Problems by Searching

Chapter 3 in the textbook.

Pinar Öztürk

Norwegian University of Science and Technology
2023

Outline

- Problem solving as state space search
- Uninformed search
  - breadth first
  - uniform cost
  - depth first
  - iterative deepening
- Brief introduction to *informed search*

Problem solving and search

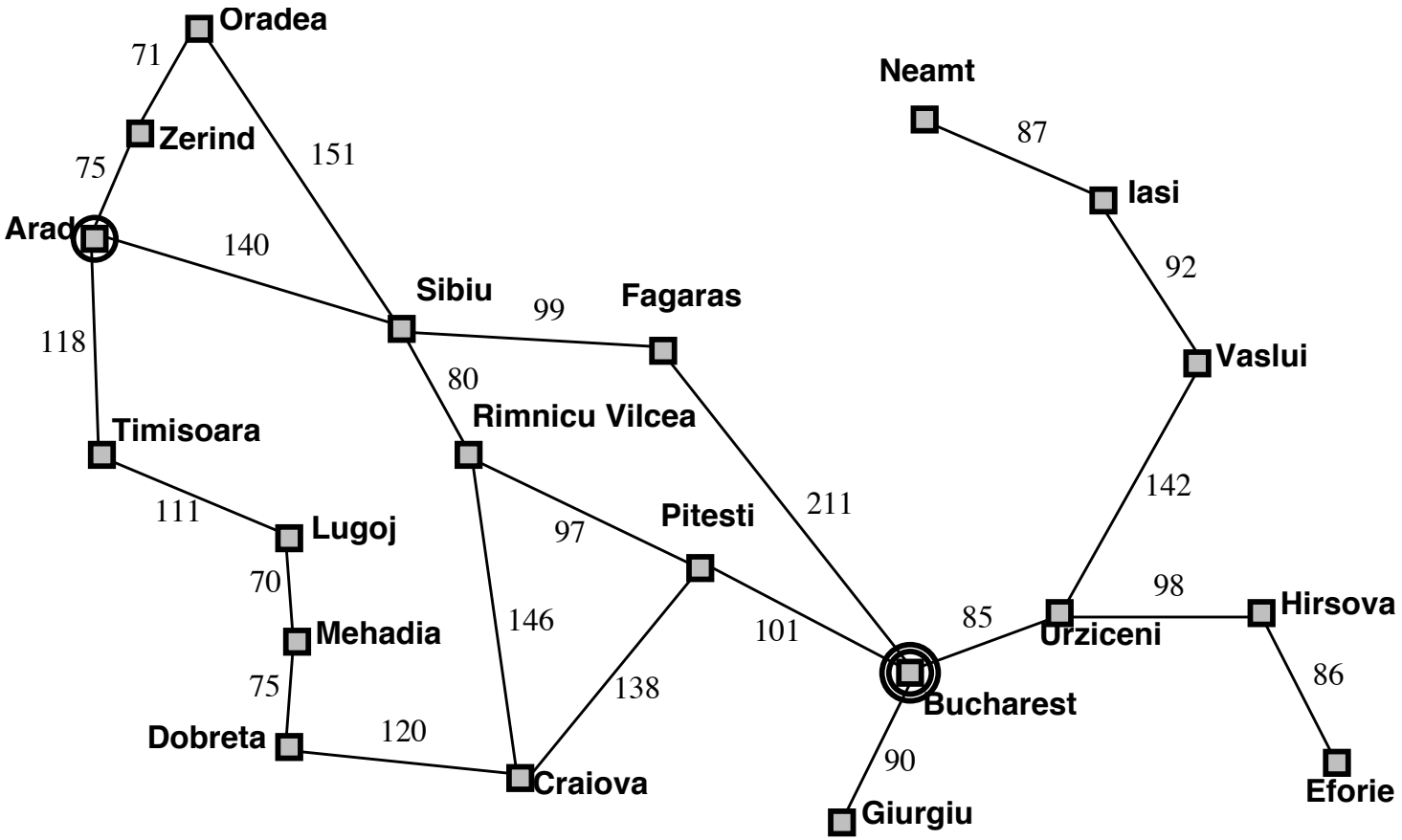
- Some problems have straightforward solutions

  Solved by applying a formula, or a well-known procedure
  Example: differential equations

- Other problems require search:

  no single standardized method
  alternatives need to be explored to solve the problem
  the number of alternatives to search among can be very large, even infinite.

Example: Romania



A simplified map of part of Romania, with road distances in miles.

Problem solving process

For fully observable, deterministic, discrete, static, known environments.

- Goal formulation
- Problem formulation - description of states and actions necessary to reach the goal
- Search - simulate sequences of actions in the world model to find a sequence that reaches the goal. This sequence of actions is a "solution".
- Execution - execute the actions in the solution, one at a time

Problem Formulation - Romania example

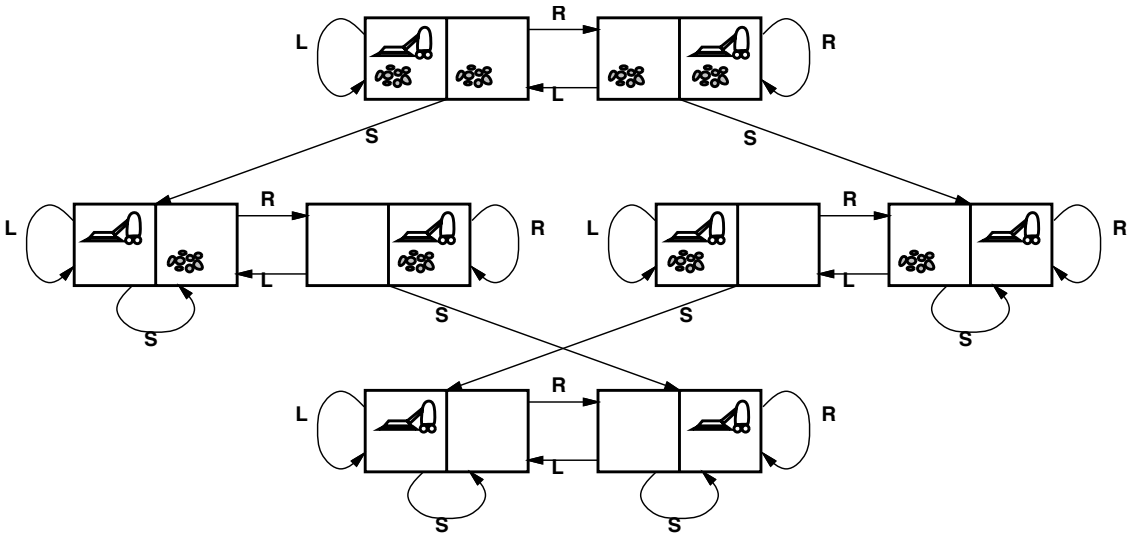
State Space: All the possible states of the environment.

Problem definition for Romania example:

- State Space, various locations in the map
- Initial state, e.g., Arad
- Goal state: Bucharest
- Actions: Actions(Arad)= ToSibiu, ToTimisoriara, ToZerind
- Transition model: Result(Arad,ToZerind)=Zerind
- Action cost function: ActionCost(s,a,s')

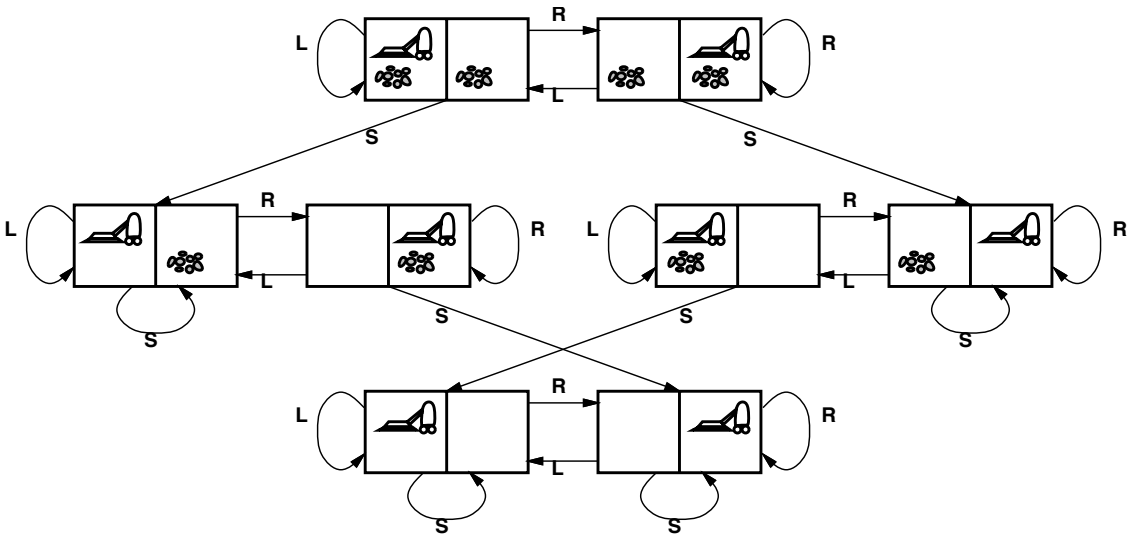
A good problem formulation has the right level of abstraction.

Problem formulation - Vacuum world example



- A state: the agent location and the dirt locations (ignore dirt amount etc.
- How many states in this state space?

Problem formulation - Vacuum world example



- A state: the agent location and the dirt locations (ignore dirt amount etc.
- How many states in this state space?
  The agent can be in one of the 2 locations, and each location may/not have dirt. $n \times 2^n$ states for an n-location environment

Problem formulation - Vacuum world example

- Initial state: any state can be defined as the initial state
- Goal states: no dirt
- Actions: Suck, moveLeft,moveRight - ... or Suck, Forward, Backward, TurnLeft, TurnRight, ..
- Transition model: Suck cleans the location, TurnLeft changes the direction of agent 90 degrees
- Action cost: 1 unit

Problem Formulation - 8-puzzle

4	8	
3	7	1
2	6	5

Start position

	1	2
3	4	5
6	7	8

Goal position

- States:
- Initial state:
- Actions:
- Transition model:
- Goal state:
- Action cost:

Problem Formulation - 8-puzzle

- States: different configurations of the tiles - integer locations of tiles
- Initial state: initial state (given)
- Actions: move blank left, right, up, down
- Transition model: the resultant configuration of the grid when an action is applied
- Goal state: given
- Action cost: 1 unit per action

Real-world problems - Touring problems

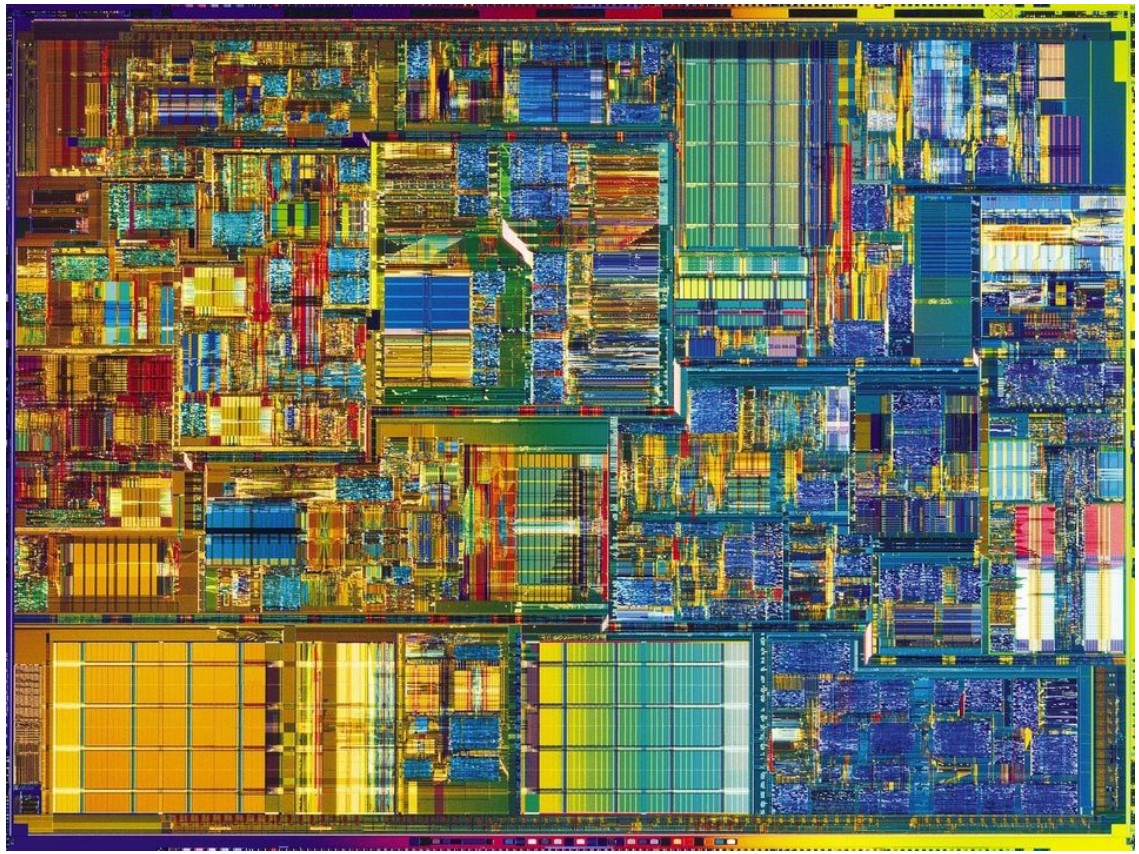
Traveling salesman problem - find shortest route that visits each location once and returns to initial location.
Example: Delivery services.

Real-world problems - Assembly problems

Find an order for assembling the parts of some object.
Example manufacturing.

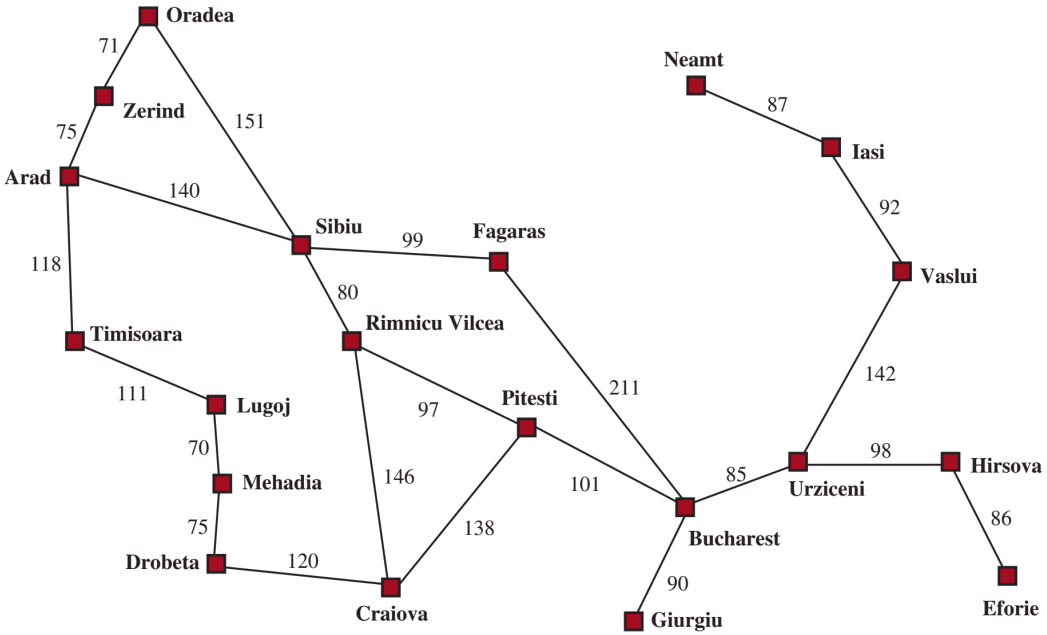
Real-world problems - Very large-scale integration

Very large-scale integration (VLSI) layout.

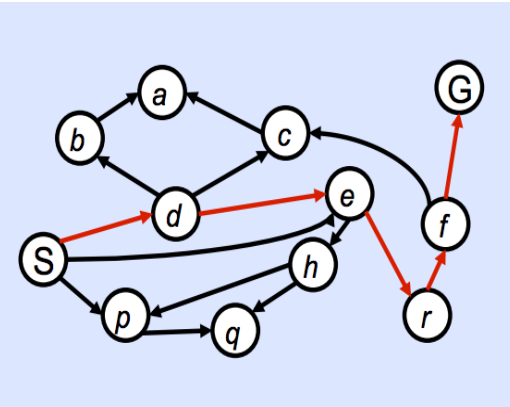


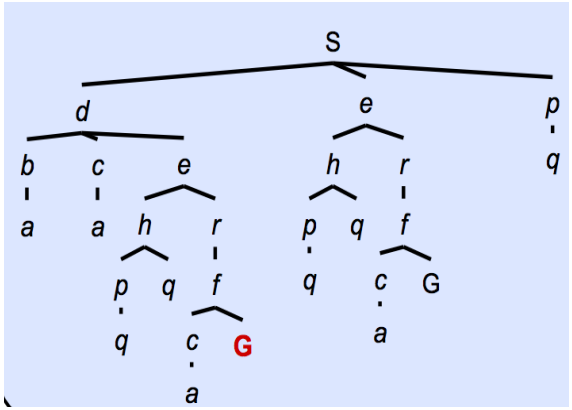
Search algorithm

Function *Search*(*problem*) returns a *solution* or *failure*



Search algorithms



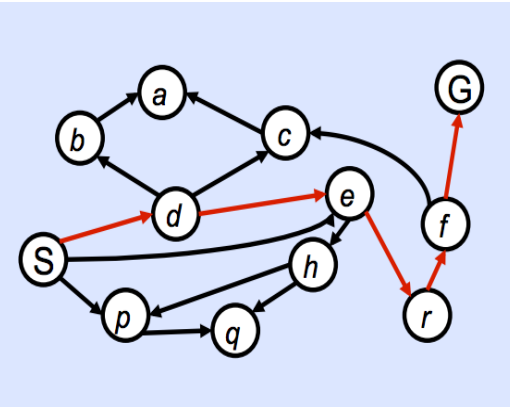


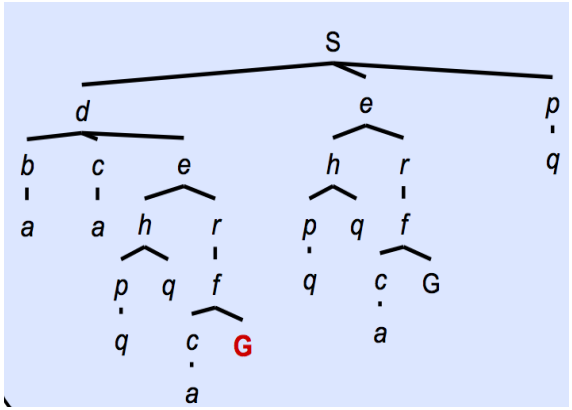
- A state is a (representation of) a physical configuration

- A state space graph represent all possible *states of the environment* and the transitions between the states.

- Search tree is superimposed on the state-space graph and shows how a particular search algorithm explores the state-space graph.

Search algorithms





- A state is a (representation of) a physical configuration

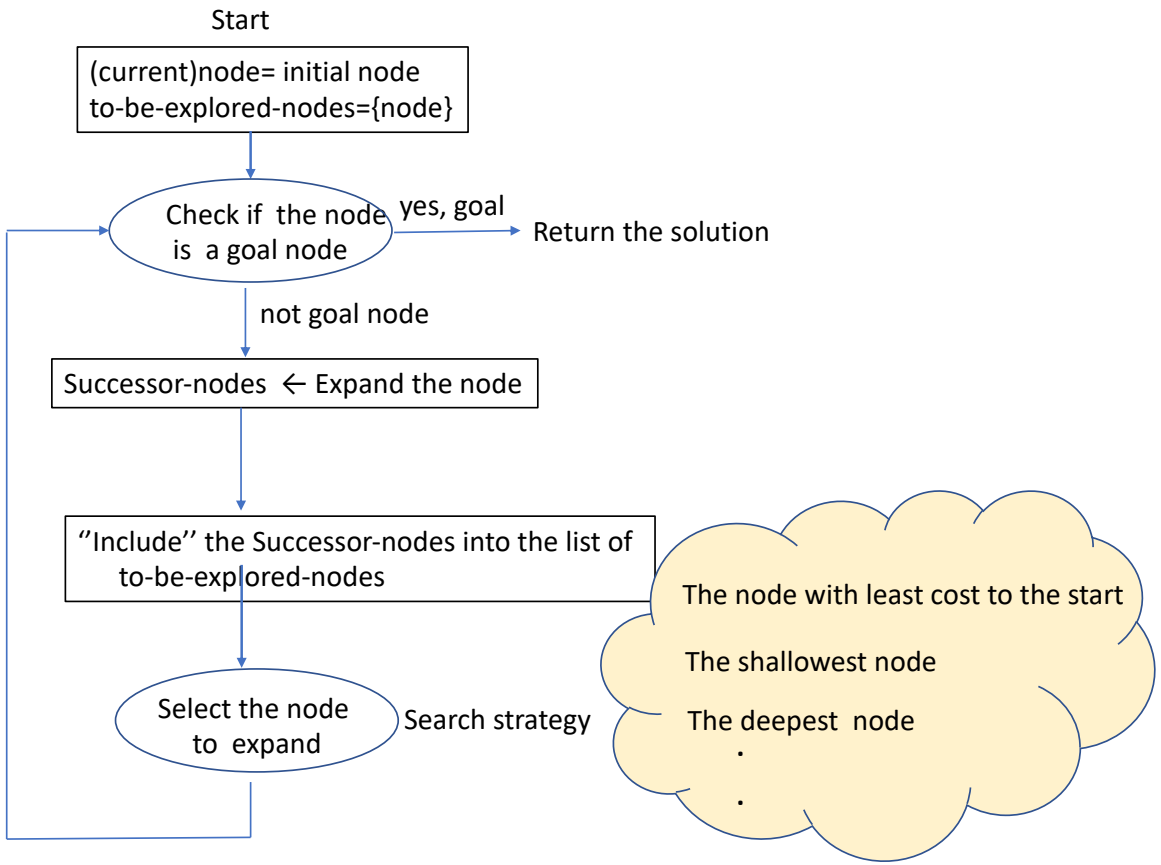
- A state space graph represent all possible *states of the environment* and the transitions between the states.

- Search tree is superimposed on the state-space graph and shows how a particular search algorithm explores the state-space graph.

What does a *node*, the *root node*, and an *edge* in the search tree correspond to in a *problem formulation*?

State graph is incrementally explored

- Most of the time it is not feasible or too expensive to build and represent the entire state graph. The problem solver agent (i.e., AI) generates a solution by **incrementally exploring** a small portion of the graph
- **Basic idea:** simulated exploration of state space by generating successors of already-explored states



Best-first search

function BEST-FIRST-SEARCH(*problem*, f) **returns** a solution node or *failure*
  node ← NODE(STATE=*problem*.INITIAL)
  frontier ← a priority queue ordered by f, with *node* as an element
  reached ← a lookup table, with one entry with key *problem*.INITIAL and value *node*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if s is not in *reached* **or** *child*.PATH-COST $<$ *reached*$[s]$.PATH-COST **then**
        reached$[s]$ ← *child*
        add *child* to *frontier*
  return *failure*

function EXPAND(*problem*, *node*) **yields** nodes
  s ← *node*.STATE
  for each *action* **in** *problem*.ACTIONS(s) **do**
    s' ← *problem*.RESULT(s, *action*)
    cost ← *node*.PATH-COST + *problem*.ACTION-COST(s, *action*, s')
    yield NODE(STATE=s', PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

Node data structure

A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree

includes state, parent, children, path cost

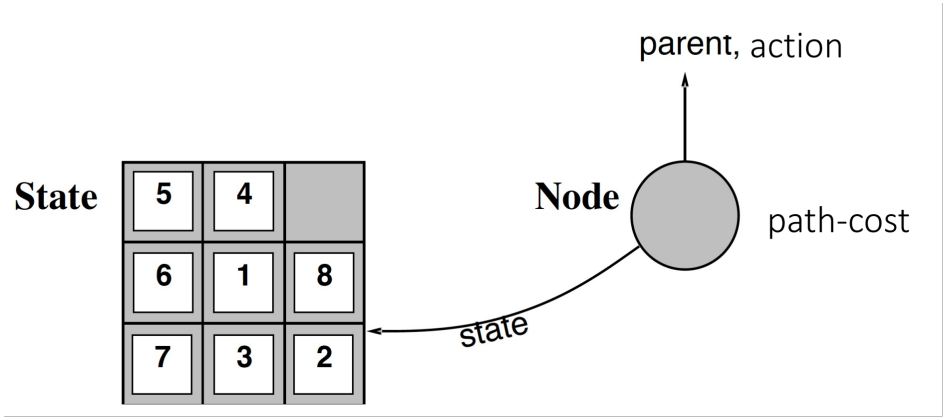
States do not have parents, children, depth, or path cost.

node.STATE

node.PARENT

node.ACTION

node.PATH-COST



Some basic concepts

We need the following concepts for the search process:

- **Frontier**: the list of the nodes in the search tree that the search process considers worth to expand. Some of these will be expanded, some other not based on the progress of the search process
- The expanded nodes will have a special status. Some books including the 3rd eds of this book called the expanded nodes as **Closed** nodes where the ones in frontier are called as **open**.
- 4th edition adopts "frontier" and **reached** instead where the latter includes both the expanded nodes and nodes in the frontier.
- Hence nodes in the "reached" but not in the "frontier" are already expanded.

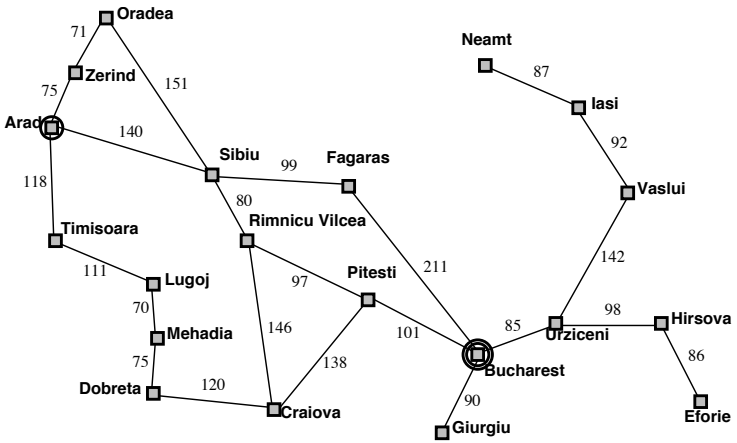
Frontier data structure

- Priority queue - best-first search
- FIFO queue - breadth-first search
- LIFO queue - depth-first search

Frontier Operations

- *IsEmpty*(*frontier*)- returns true only if there are no nodes in the frontier
- *Pop*(*frontier*) - removes the top node from the frontier and returns it.
- *Top*(*frontier*) - returns (but does not remove) the top node of the frontier.
- *Add*(*node*, *frontier*) - inserts node into its proper place in the queue.

Redundant paths



- Arad is a *repeated* state in the search tree.
- Arad→ Sibiu→Arad. is a *loopy/cyclic* path ⇒*infinite* search tree!
- There are also redundant but not loopy paths. Example: Two paths to Sibiu with difference path costs:
  1. Arad→Zerind→Oradea→Sibiu, and 2. Arad→Sibiu
- How to eliminate redundant paths?

How to Handle Redundant Paths

- Find all redundant paths - can be found through checking *reached nodes* and nodes in the frontier in order to remove the costly paths. The *Best-first search* algorithm in the book is a *graph search* (versus "tree-like search") algorithm doing this.
- Don't worry about redundant path problem, e.g., in assemby problem
- Check only cyclic loops - can be found by checking the chain of parent nodes

Measuring problem solving performance

Completeness - is algorithm guaranteed to find a solution (or failure)?

Cost optimality - does it find a solution with the lowest path cost?

Time complexity - how long it takes to find a solution (in seconds or states and actions considered)?

Space complexity - how much memory needed, e.g. *frontier*, *reached*?

How to measure complexity

For explicit graph:
$|V| + |E|$ - size of state-space graph

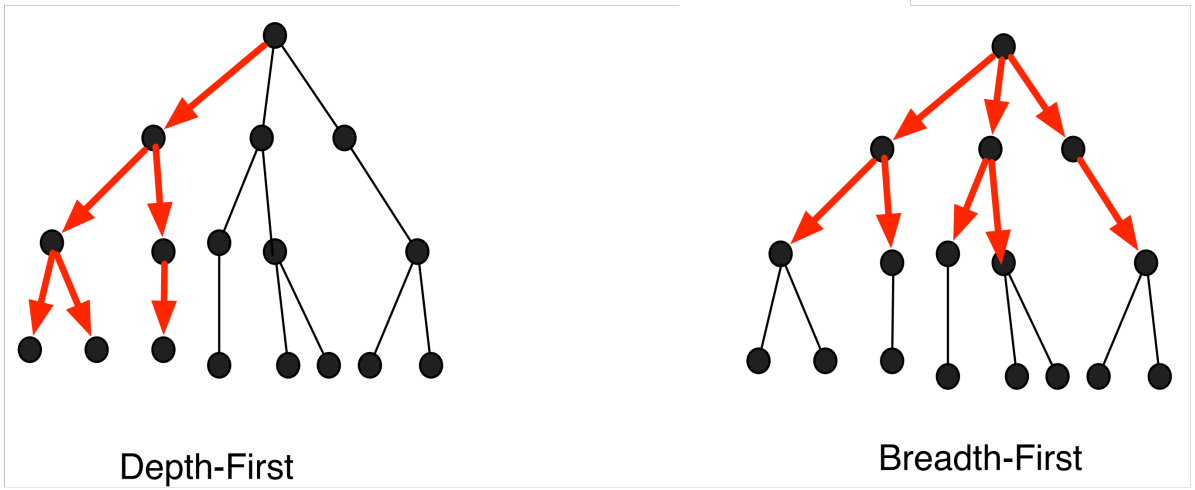
For implicit graph defined by initial state, actions and transition model:

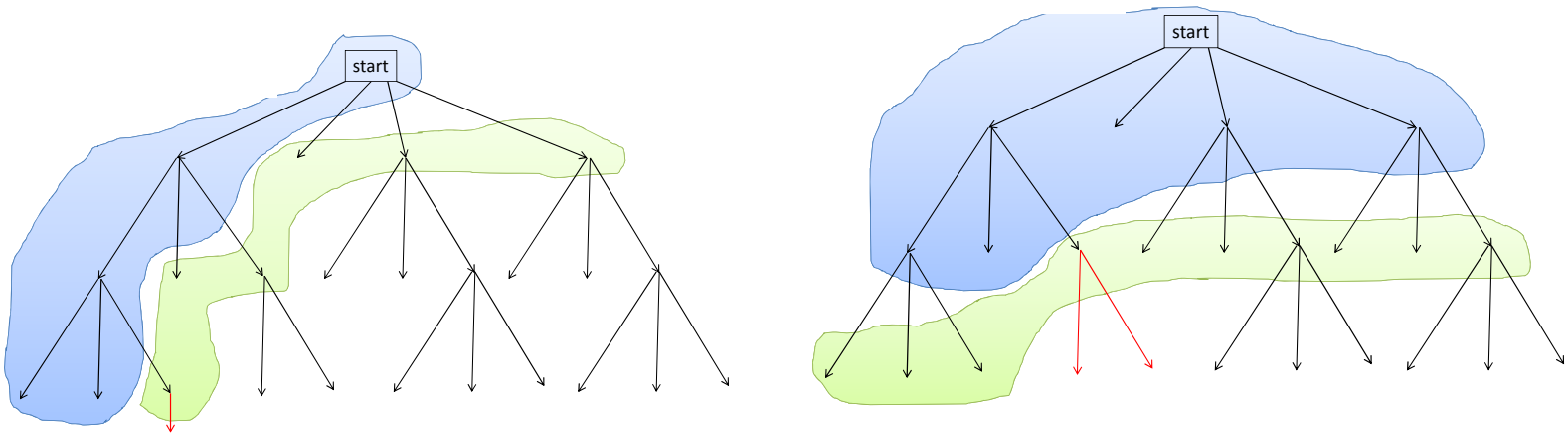
- **d** - depth - depth of the shallowest solution
- **b** - branching factor - number of successors of node to consider
- **m** - maximum depth of the state space (may be infinite).

Uninformed search strategies

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

Difference in the order of checking nodes





Breadth-first search

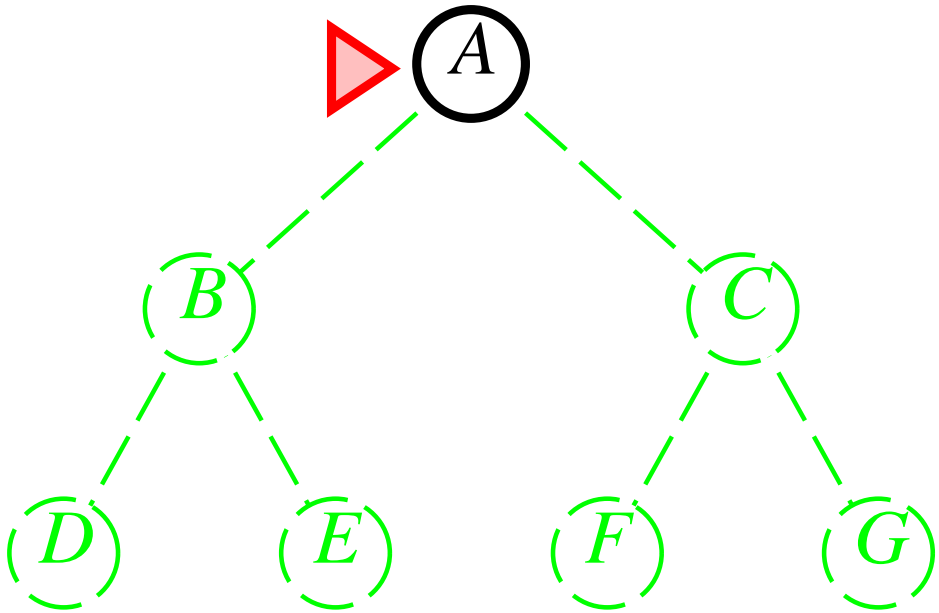
```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

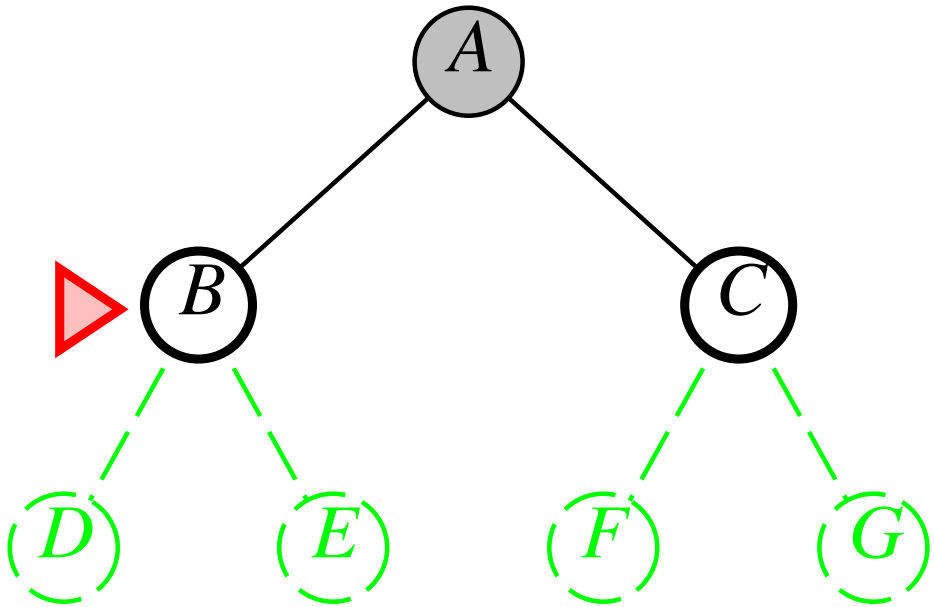


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

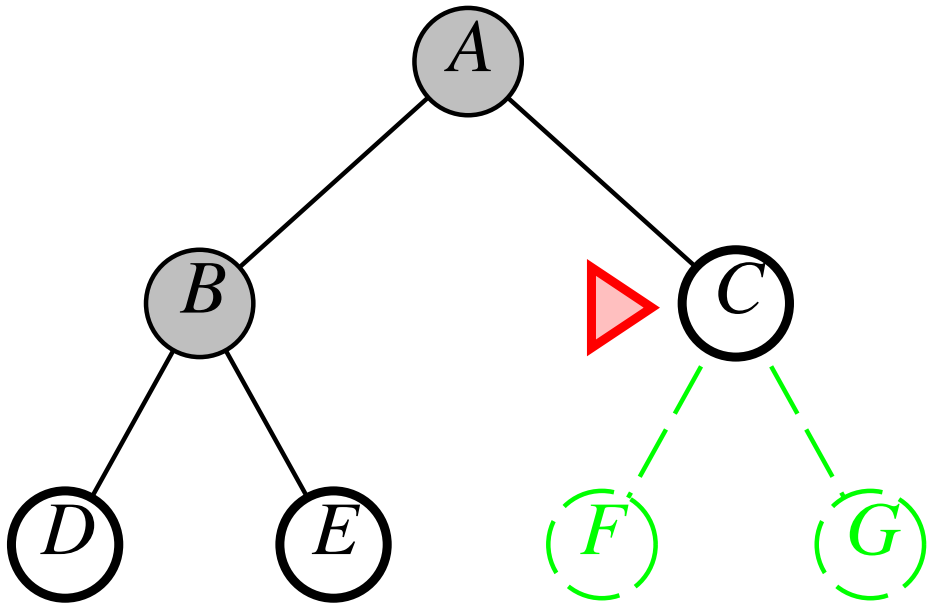


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

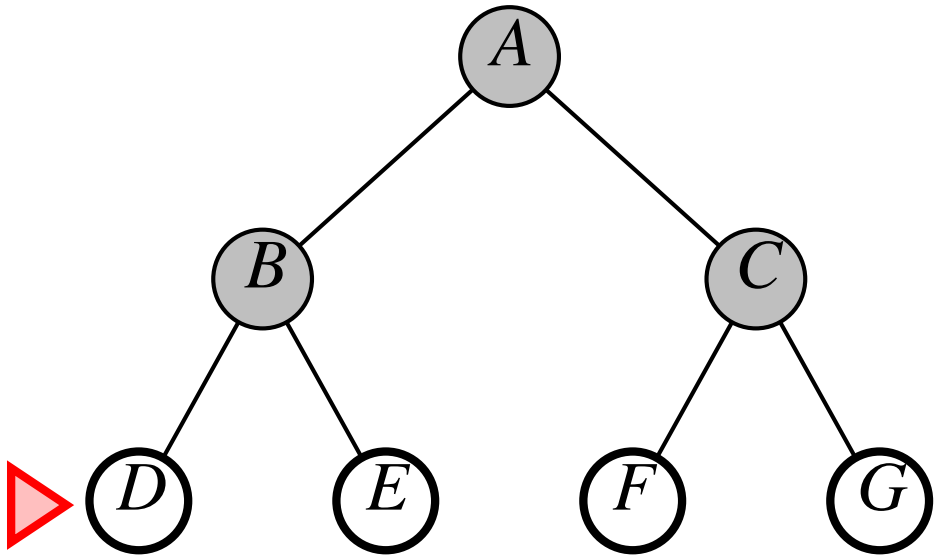


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

- Complete: Yes (if b is finite, and the state space either has a solution or is finite)
- Cost optimal: Yes - if actions have the same cost
- Time and space: $1 + b + b^2 + b^3 + ... + b^d = O(b^d)$

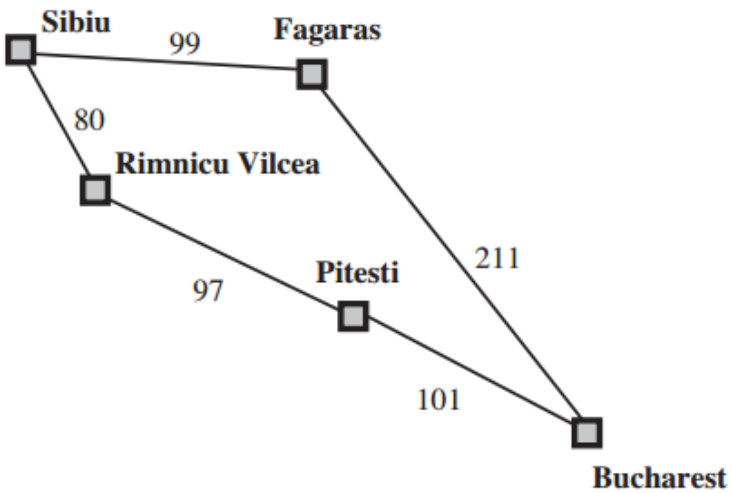
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

- Long time to find long solutions since we explore all shorter length possibilities first

Uniform-cost search

- Handles varying positive step cost
- Uses priority queue sorted by path cost
- Expands nodes with least path cost first.

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)



Properties of Uniform-cost search

- Complete: Yes -if b is finite, the state space either has a solution or is finite, and step cost $\geq \epsilon$
- Cost optimal : Yes
- Time and space : $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, i.e., $O(b^{d+1})$

  C^* - optimal solution cost

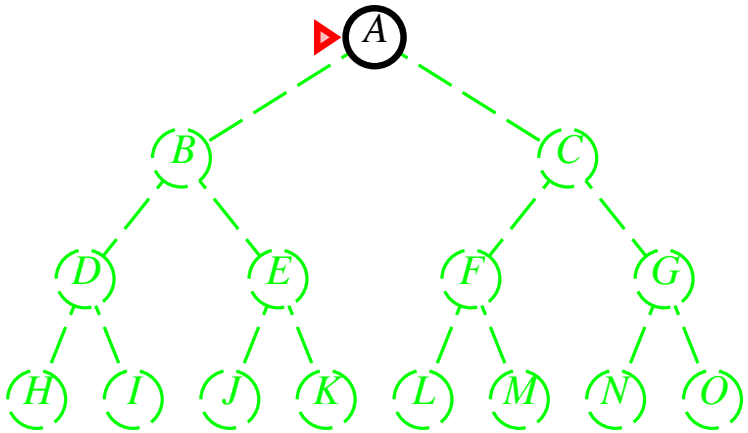
  ϵ - action cost lower bound

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Initial state = A
Is A a goal state?
Put A at front of queue.
frontier = $[A]$

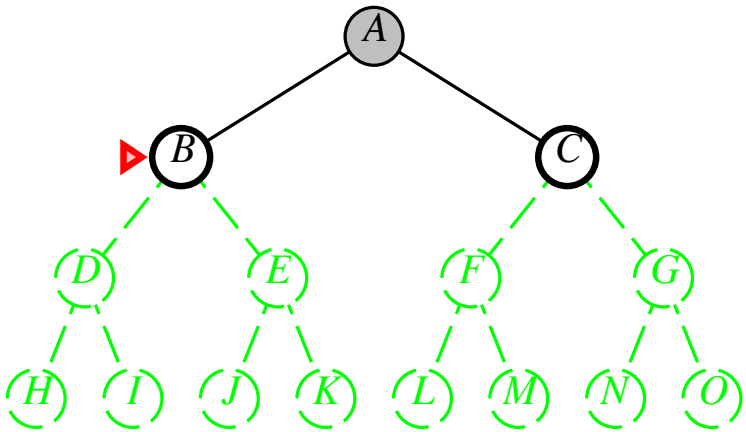
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand A to B, C.
Put B, C at front of queue.
frontier = $[B, C]$

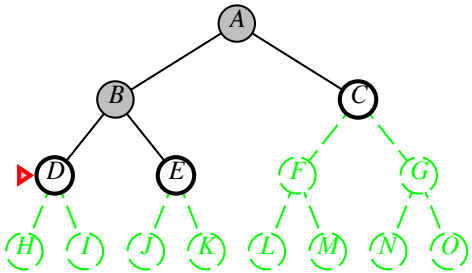
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand B to D, E. Put D, E at front of queue. frontier = $[D, E, C]$

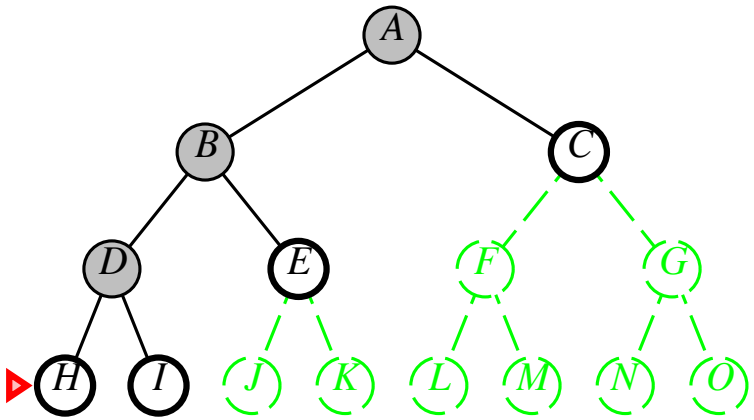
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand D to H, I.
IPut H, I at front of queue.
frontier $= [H, I, E, C]$

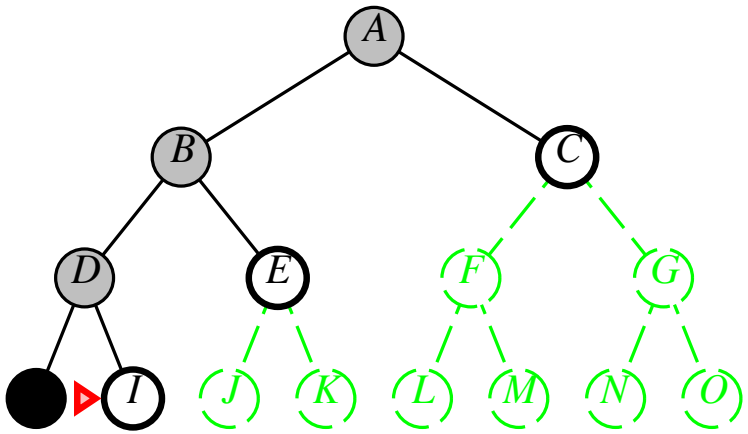
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand H to no children.
Forget H.
frontier = $[I, E, C]$

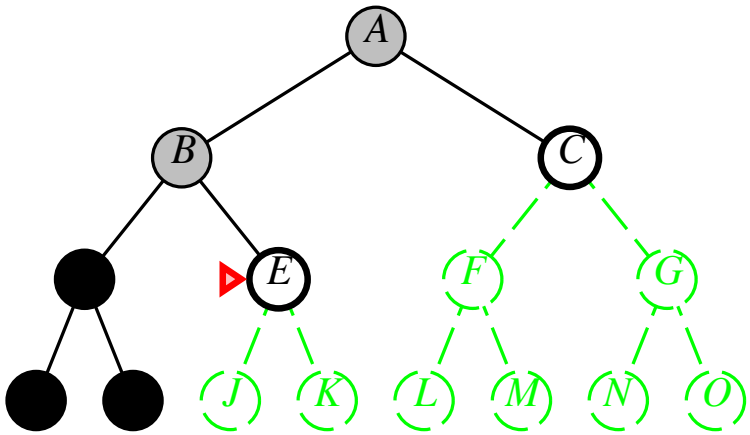
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand I to no children.
Forget D, I.
frontier = $[E, C]$

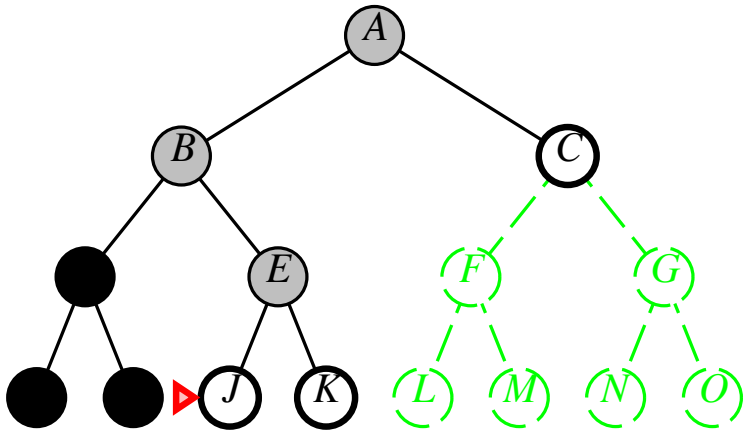
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand E to J, K.
Put J, K at front of queue.
frontier $= [J, K, C]$

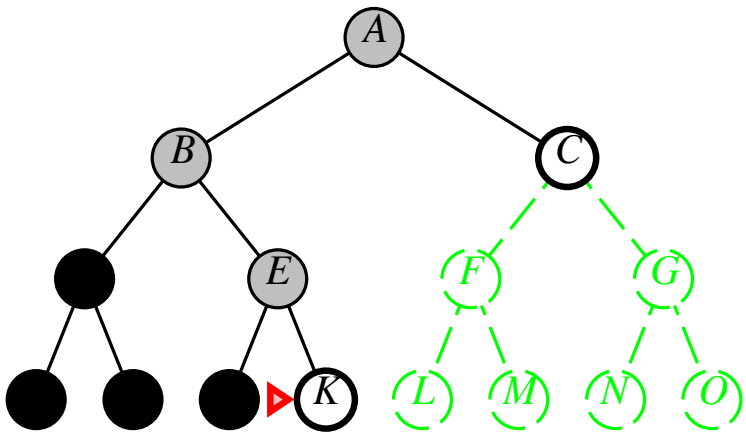
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand J to no children.
Forget J.
frontier = $[K, C]$

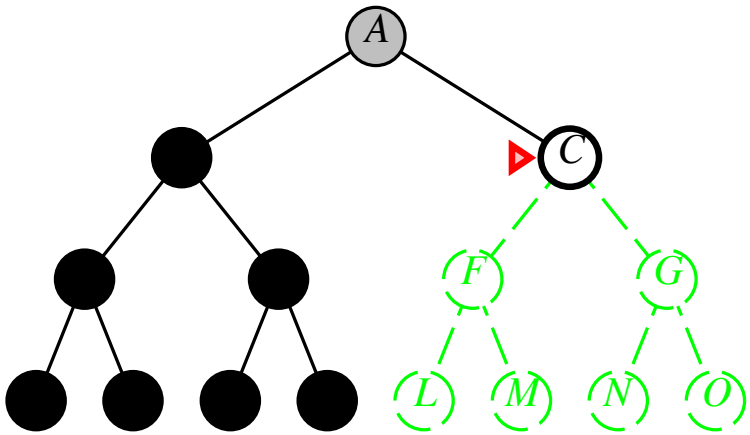
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand K to no children.
Forget B, E, K.
frontier = $[C]$

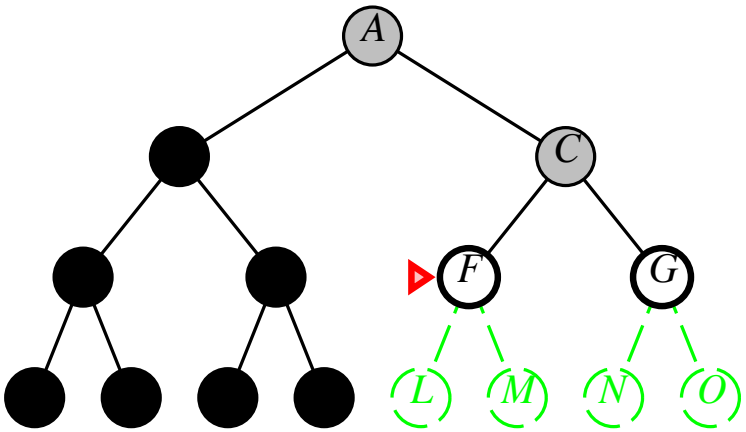
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Expand C to F, G.

Put F, G at front of queue.
frontier = $[F, G]$

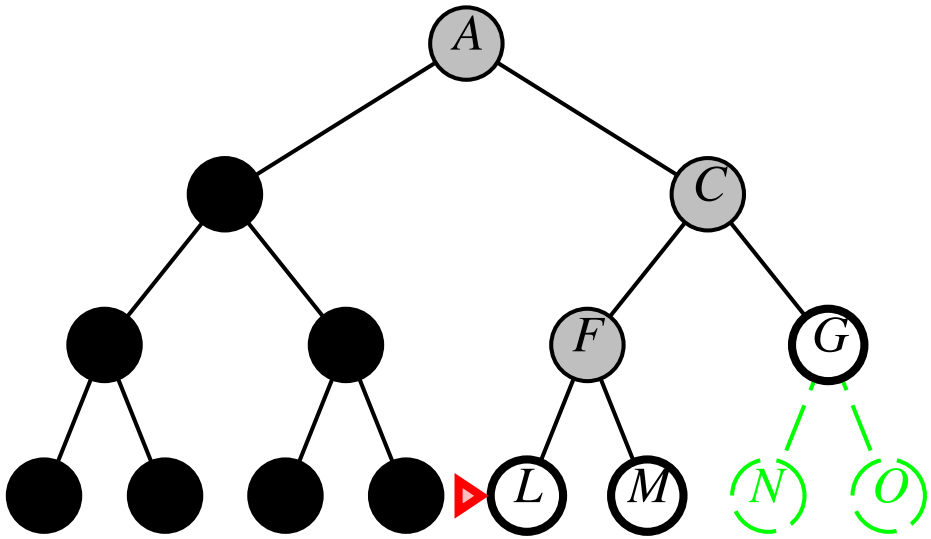
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

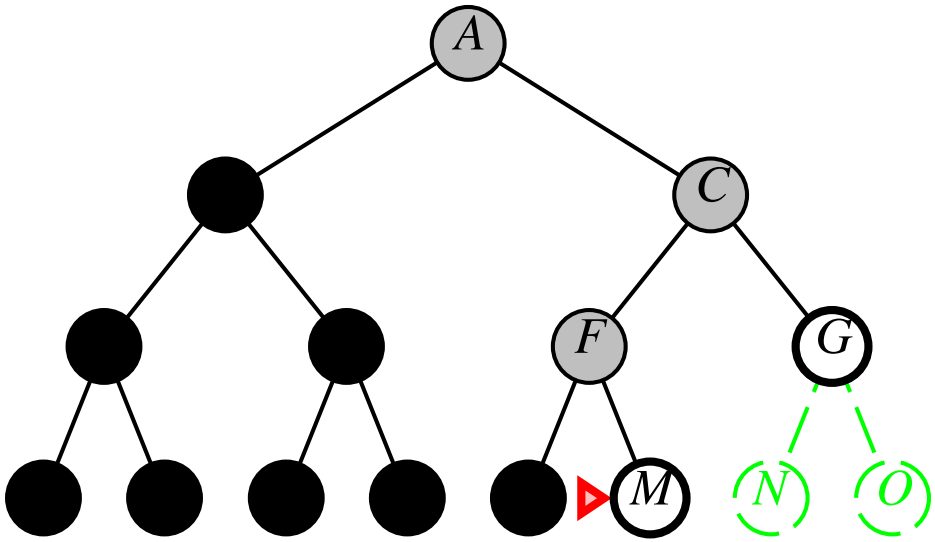


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Properties of depth-first search

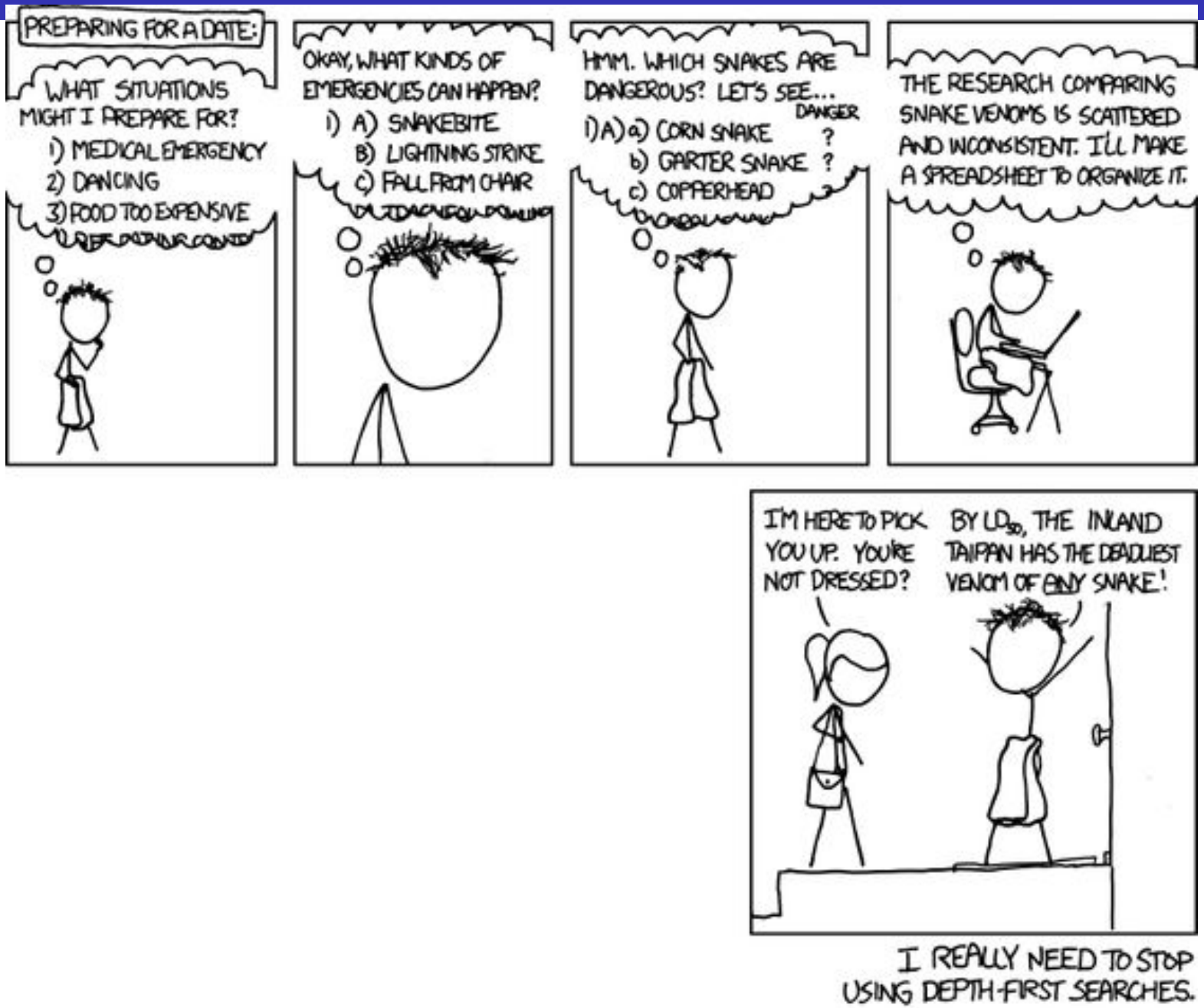
- Complete : No - fails in infinite-depth spaces and spaces with loop
- Cost optimal : No
- Time : $O(b^m)$
- Space : $O(bm)$ - linear, for tree-like search not checking

Properties of depth-first search

Complete?? No: tree search fails in infinite-depth spaces, e.g., spaces with loops

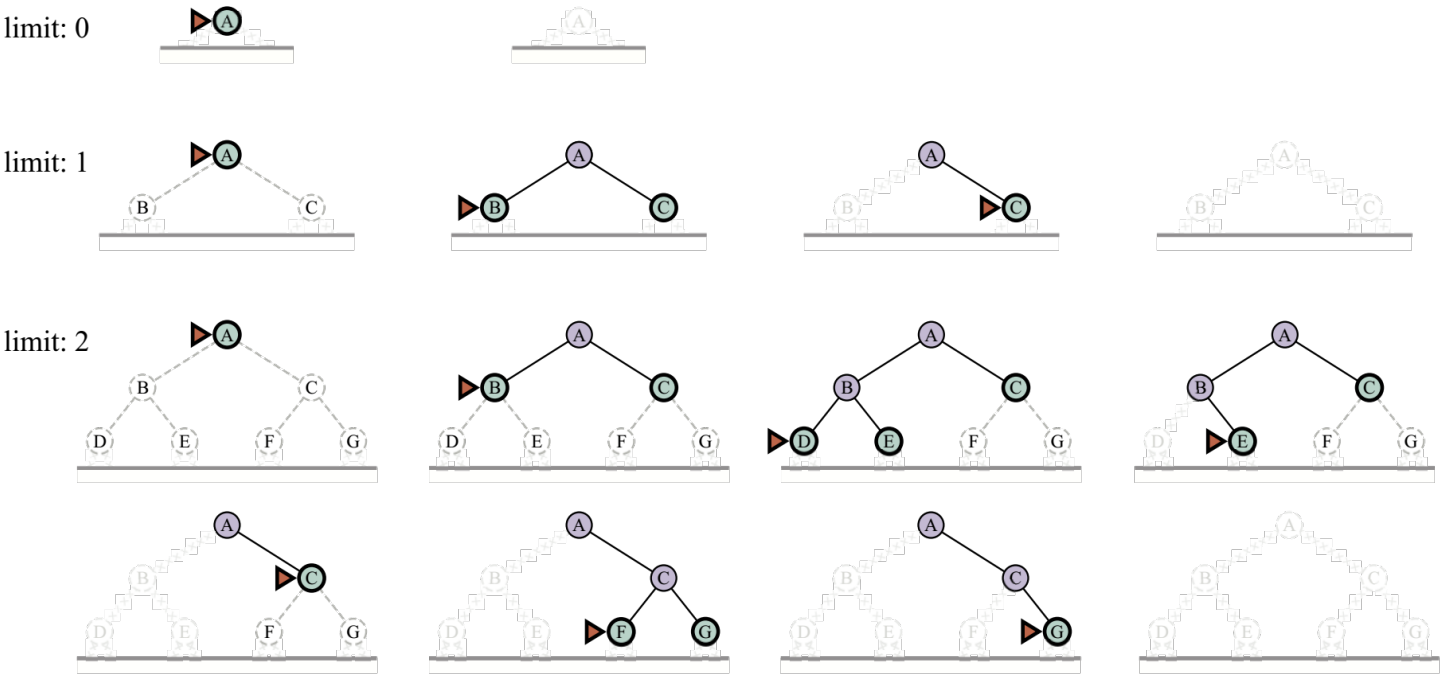
1. Modify to avoid repeated states along path
   Check if current nodes occurred before on path to root.
   No extra memory cost but it avoids only infiite loops, not redundant paths.
2. Can use graph search (remember all nodes ever seen).
   It is complete in finite spaces.
   Problem with graph search: space is exponential, not linear
   Still fails in infinite-depth spaces (may miss goal)

DFS

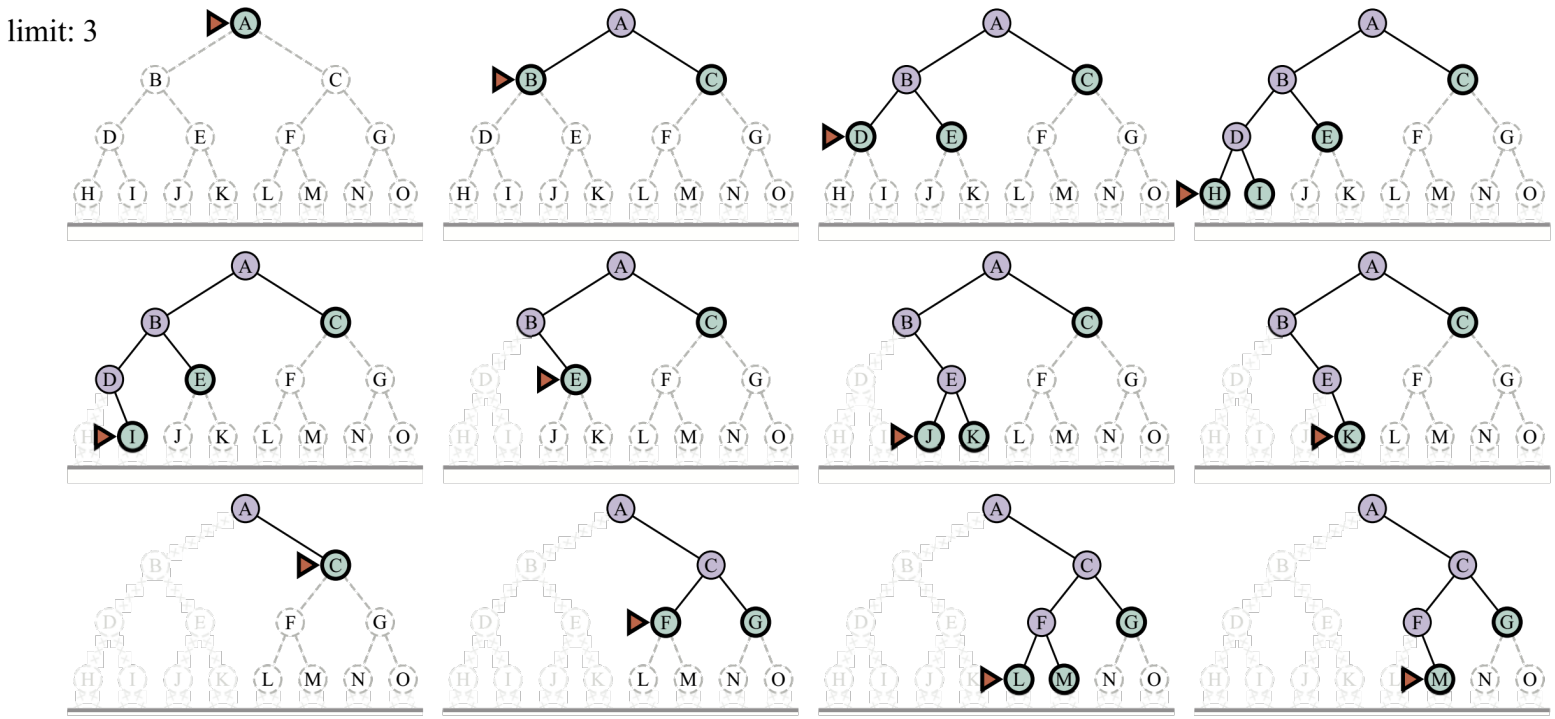

PREPARING FOR A DATE:
WHAT SITUATIONS MIGHT I PREPARE FOR?
1) MEDICAL EMERGENCY
2) DANCING
3) FOOD TOO EXPENSIVE
OKAY, WHAT KINDS OF EMERGENCIES CAN HAPPEN?
1) A) SNAKEBITE
B) LIGHTNING STRIKE
C) FALL FROM CHAIR
HMM. WHICH SNAKES ARE DANGEROUS? LET'S SEE...
1) A) a) CORN SNAKE ?
b) GARTER SNAKE ?
c) COPPERHEAD
DANGER
THE RESEARCH COMPARING SNAKE VENOMS IS SCATTERED AND INCONSISTENT. I'LL MAKE A SPREADSHEET TO ORGANIZE IT.
I'M HERE TO PICK YOU UP. YOU'RE NOT DRESSED?
BY LD_{50}, THE INLAND TAIPAN HAS THE DEADLIEST VENOM OF ANY SNAKE!
I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.

Depth-limited and iterative deepening search

- Do DFS to depth 0, then (if no solution) DFS to depth 1, etc.



Iterative deepening search



Iterative deepening search

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

function DEPTH-LIMITED-SEARCH(*problem*, ℓ) **returns** a node or *failure* or *cutoff*
  frontier $\leftarrow$ a LIFO queue (stack) with NODE(*problem*.INITIAL) as an element
  result $\leftarrow$ *failure*
  while not IS-EMPTY(*frontier*) **do**
    node $\leftarrow$ POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    if DEPTH(*node*) $>$ ℓ **then**
      result $\leftarrow$ *cutoff*
    else if not IS-CYCLE(*node*) **do**
      for each *child* **in** EXPAND(*problem*, *node*) **do**
        add *child* to *frontier*
  return *result*

PS! This is a tree-like search.

Properties of iterative deepening search

- Complete : Yes - if b is finite, and the state space either has a solution or finite.
- Cost optimal : Yes, if costs are equal
- Time : $(d)b^1 + (d-1)b^2 + (d-2)b^3 ..... + b^d = O(b^d)$
- Space : $O(bd)$, like DFS

- Has advantages of BFS (completeness) and DFS (i.e., limited space, finds longer paths quickly)

Uninformed strategies -Summary of the properties

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Uninformed Search disadvantageous

- systematically searching the search space blindly- not questioning where the goal may be in the space
- search space is often very large. Time/space problems with such exhaustive search - think of chess.
- in such situations, better to use algorithms which do a more informed search

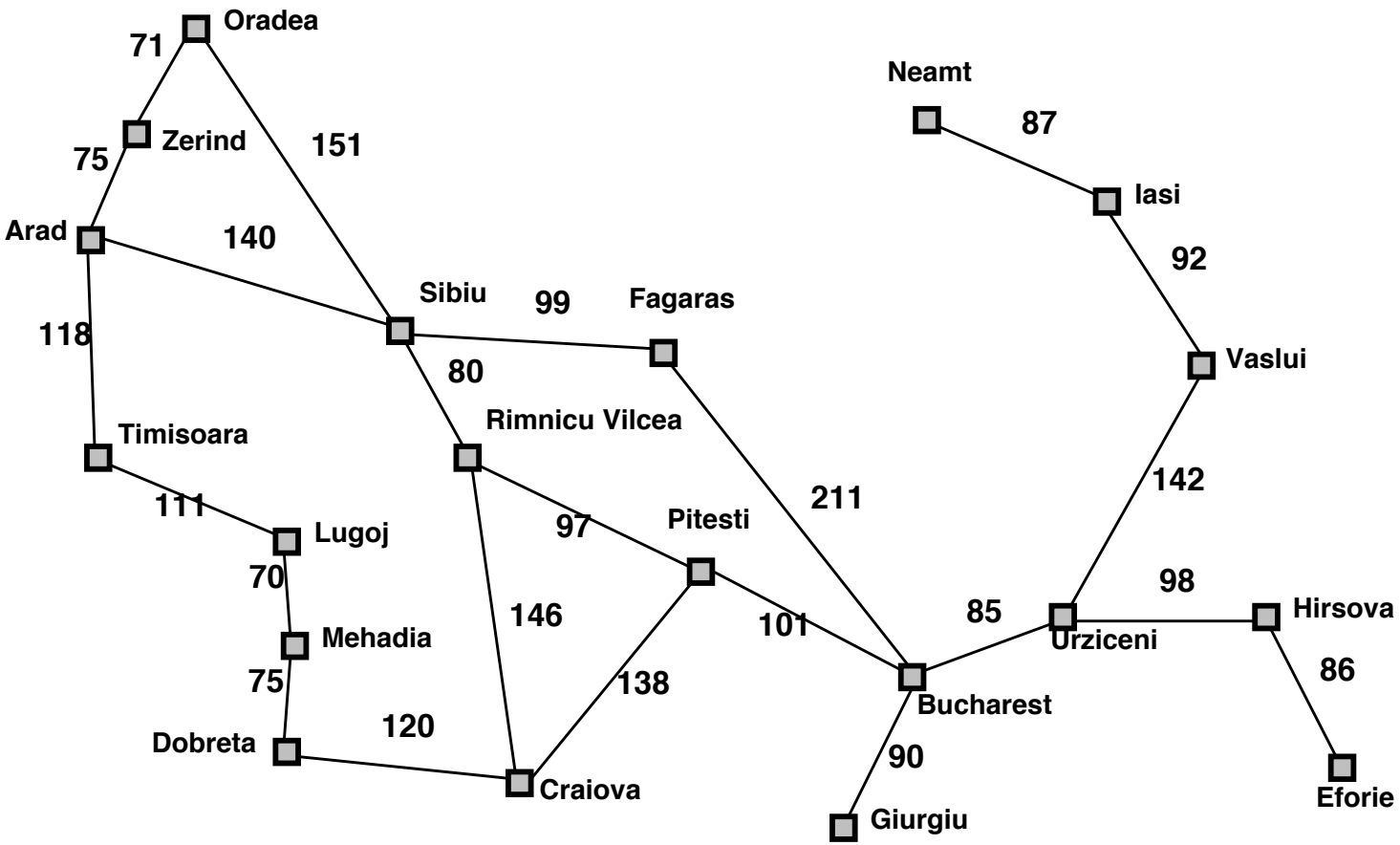
Informed Search

Use domain-specific hints about the "desirability" of nodes.

Heuristic function: $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

Heuristic - example

h_{SLD}: straight line distance to Bucharest.



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

Greedy Best search

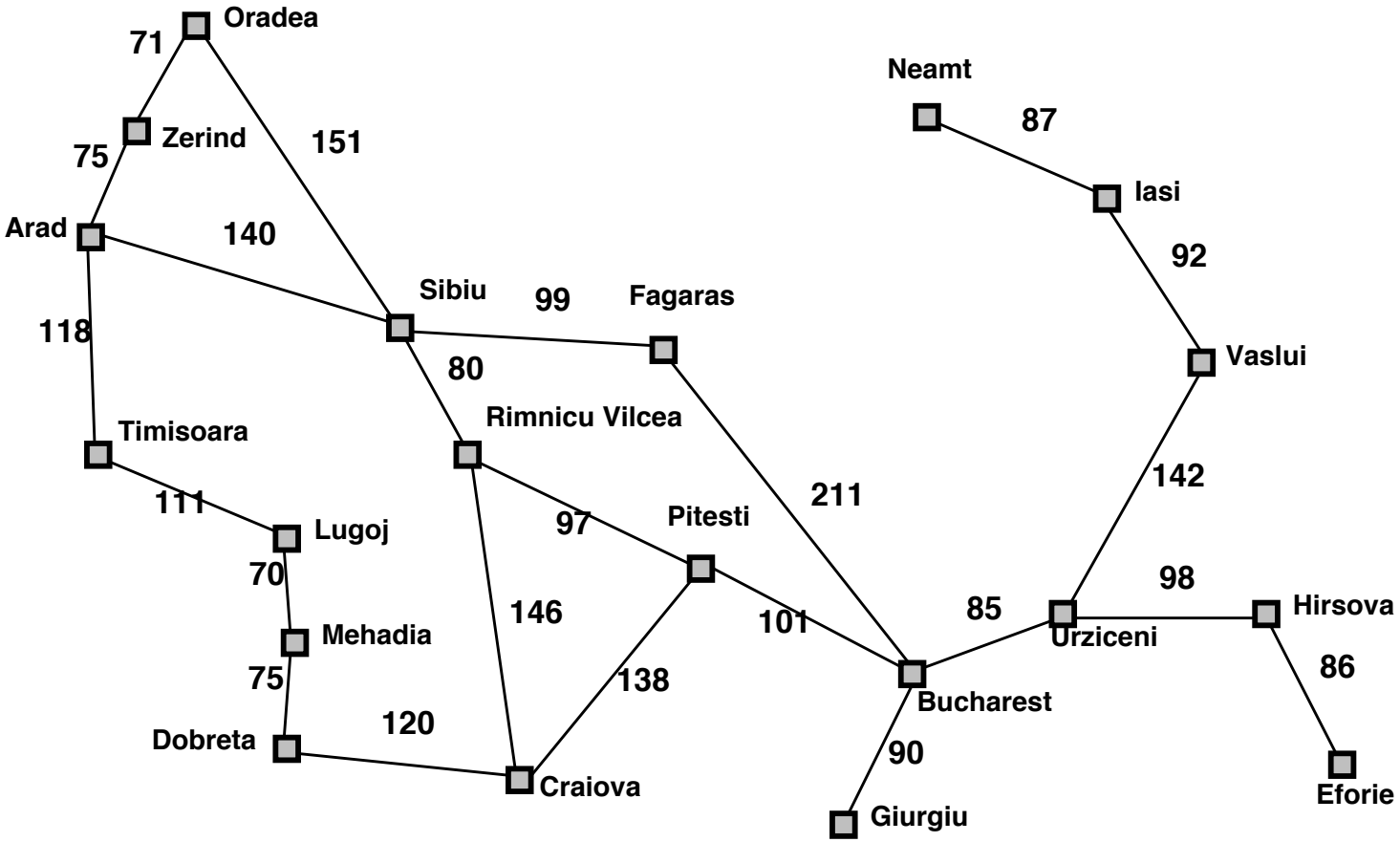
Evaluation function $= h(n)$ (heuristic)

estimate of cost from n to the closest goal

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

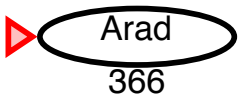
Greedy search expands the node that appears to be closest to goal

Greedy Best First Search, Romania Example

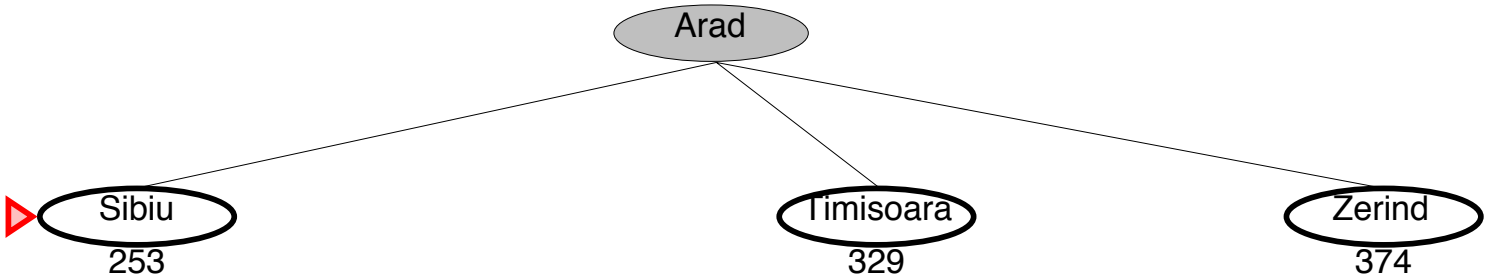


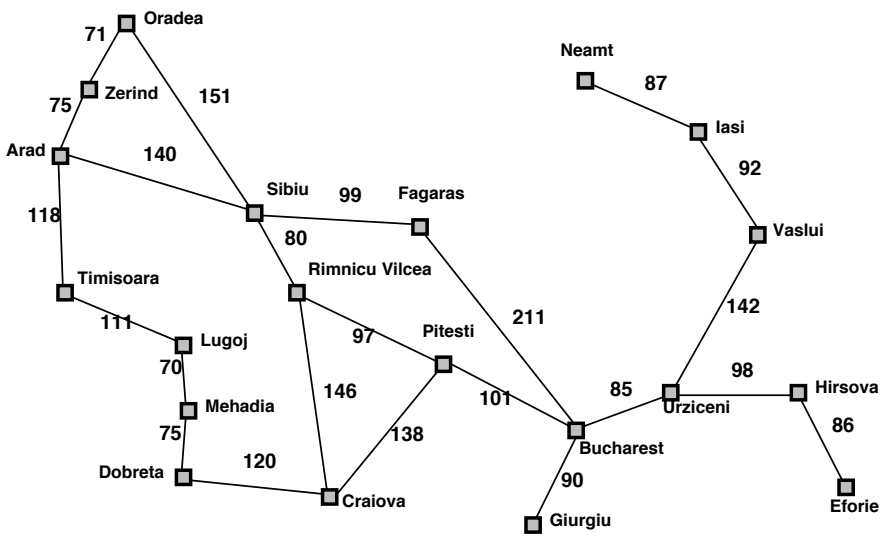
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best first search, Romania example



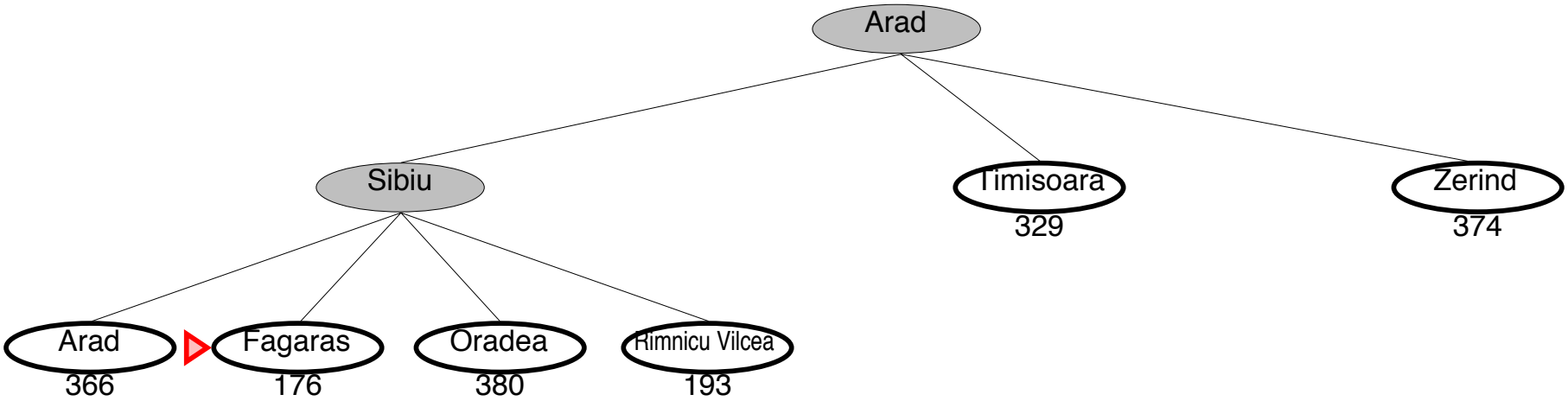
Greedy best first search, Romania example

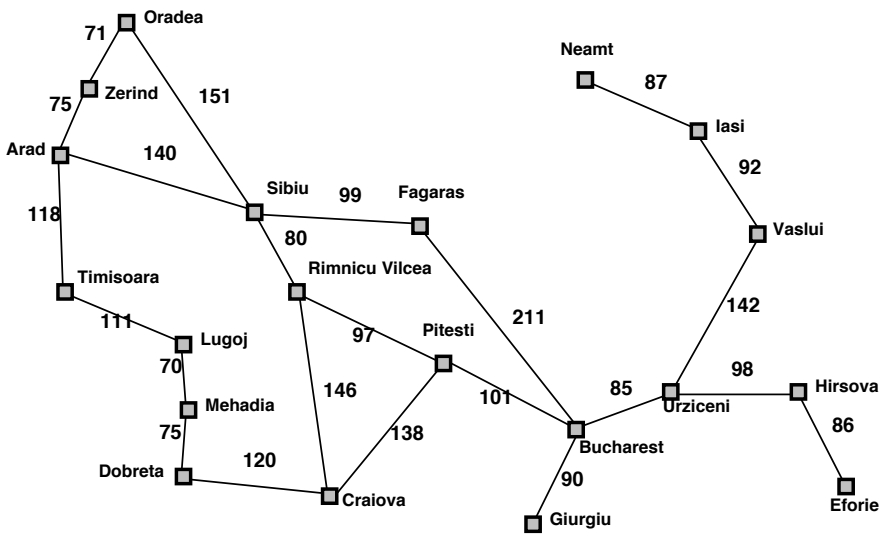




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best first search, Romania example

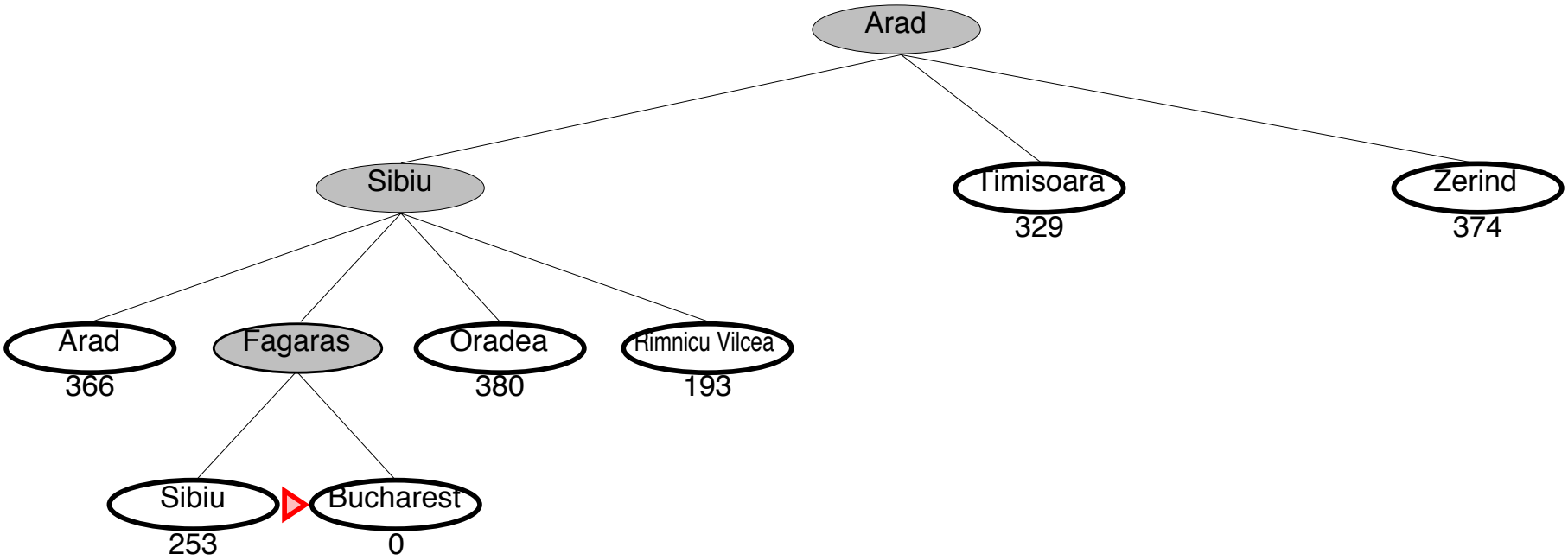


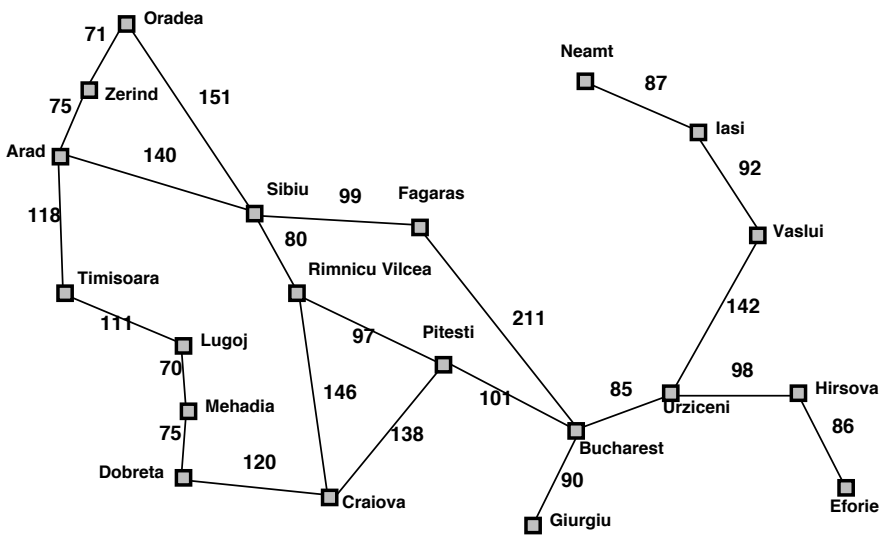


Straight–line distance to Bucharest

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

Greedy best first search, Romania example





Straight–line distance to Bucharest

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

Properties of greedy best first search

- Complete:

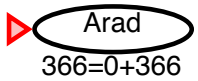
  Yes in finite state space
- Optimal No. E.g., the path it found from Sibiu to Bucharest is (Sibiu-Fagaras-Bucharest); longer than the path Sibiu-Rimnicu Vilcea-Pitesti -Buchares.
- Time and Space complexity: $O(|V|)$ (for Romania example), but with a good heuristic maybe down to $O(bm)$ on some problems

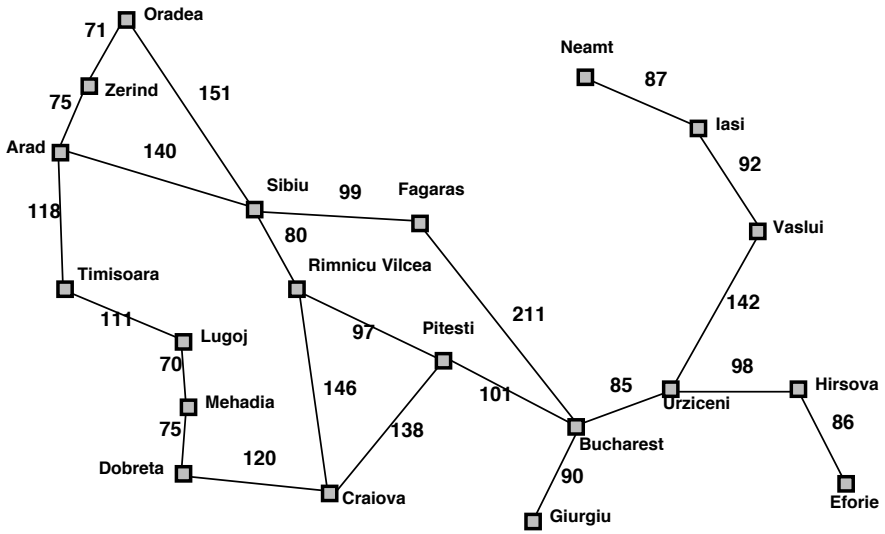
A* search

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

$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost of the cheapest path from n to goal node
$f(n)$ = estimated cost of the cheapest solution through n to goal

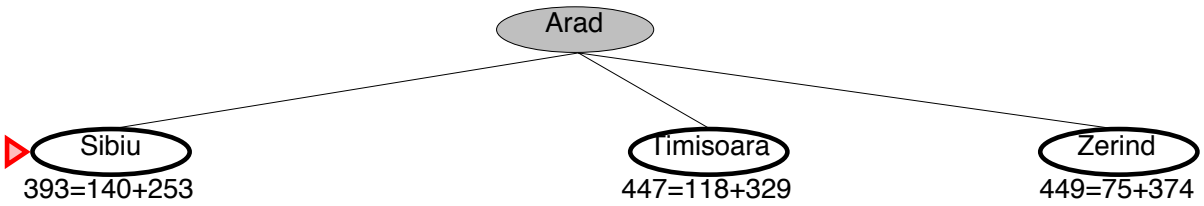
A* search example

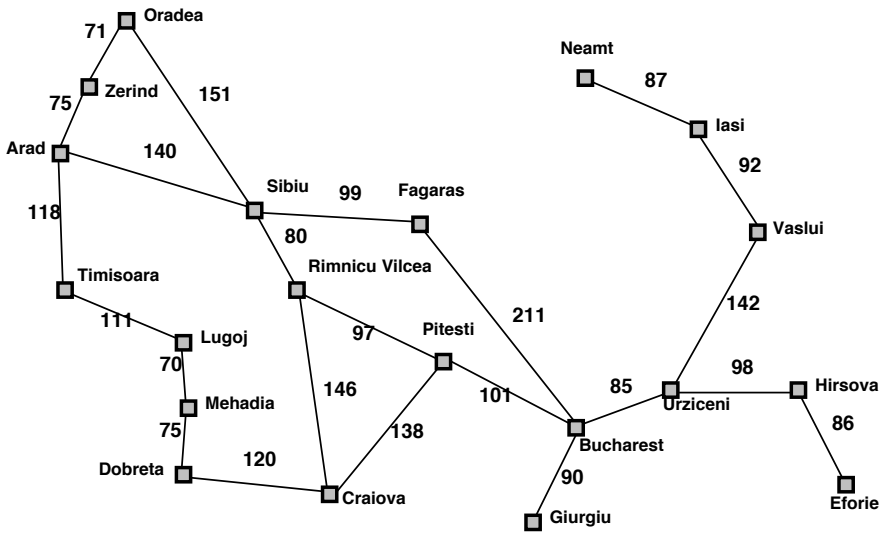




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

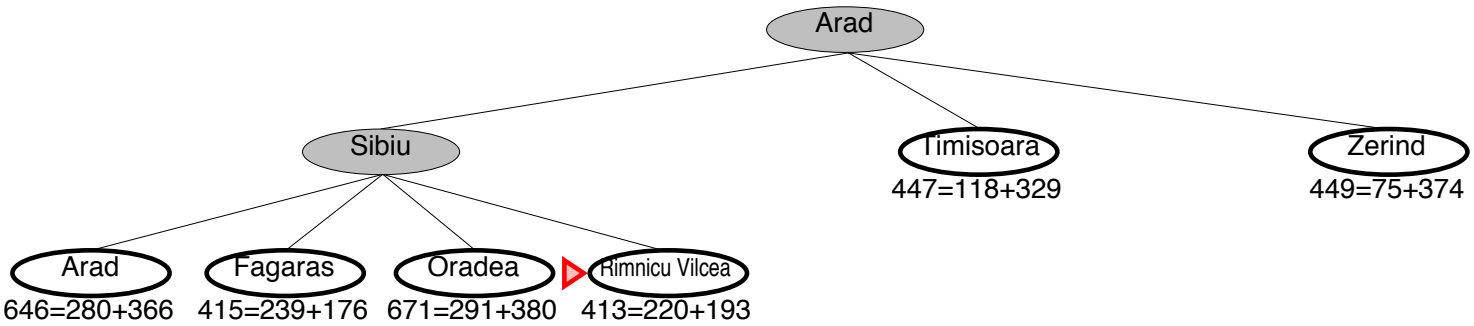


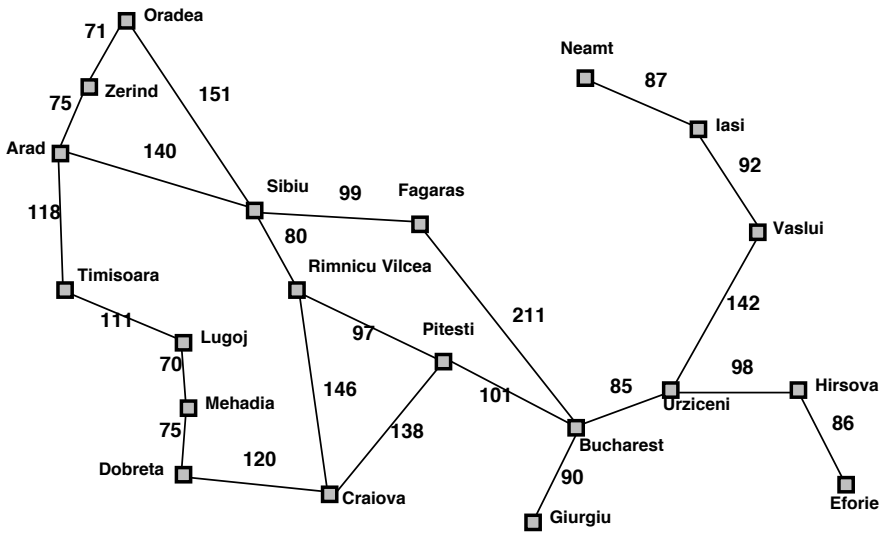


Straight–line distance to Bucharest

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

A* search example

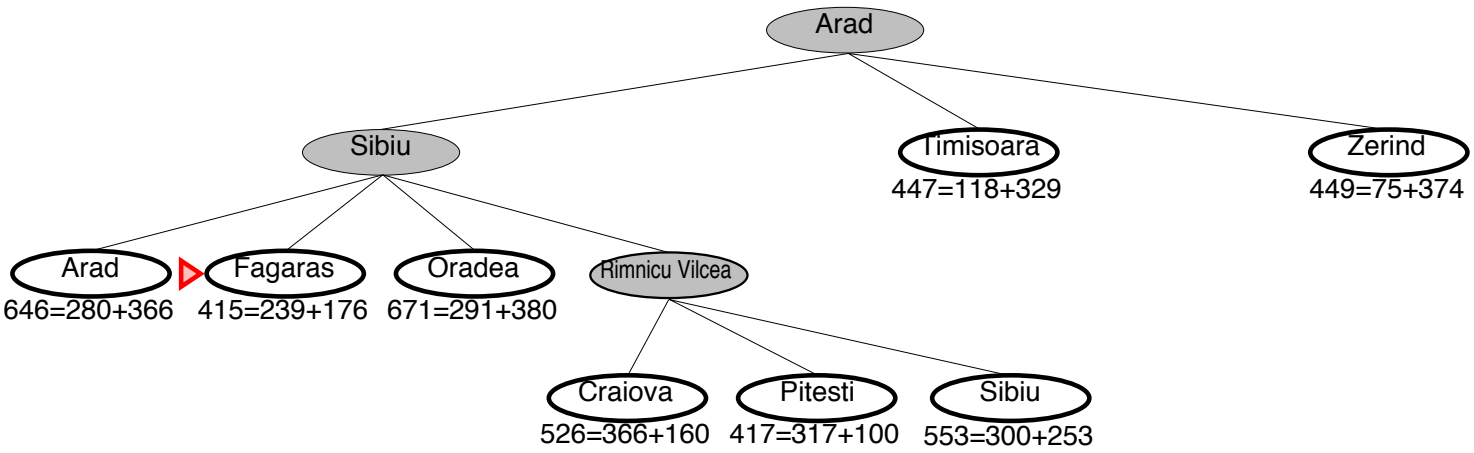




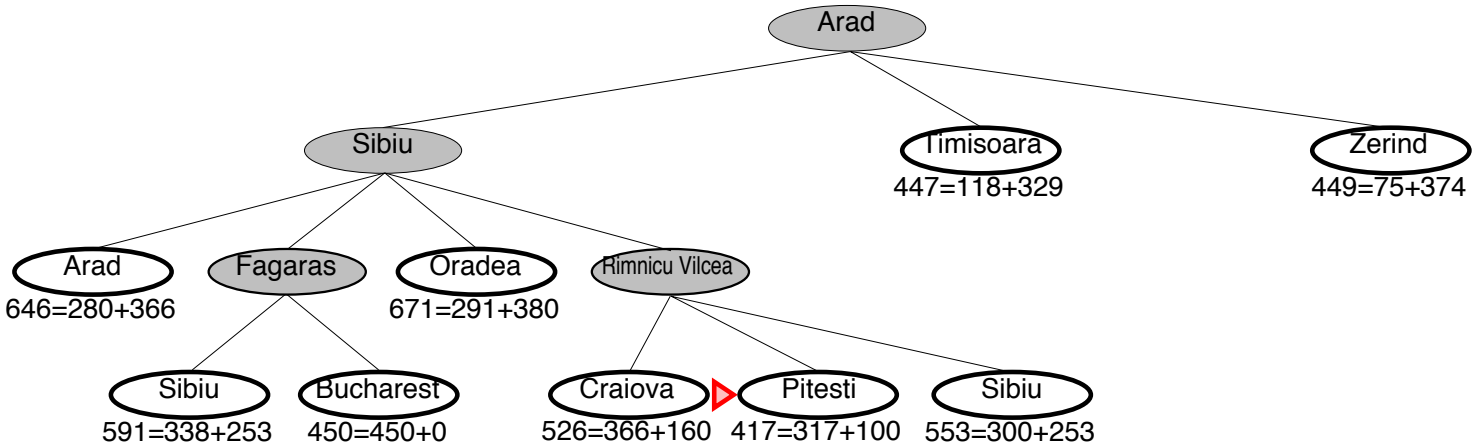
Straight–line distance to Bucharest

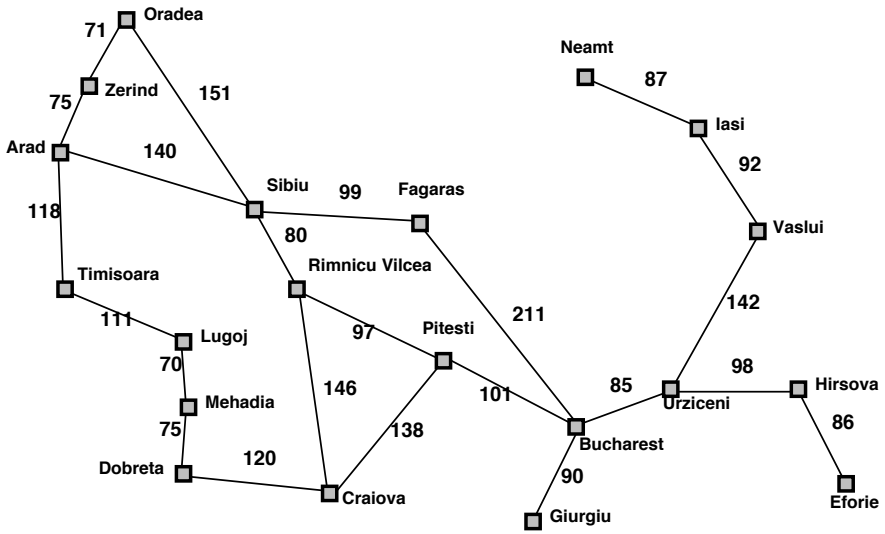
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example



A* search example

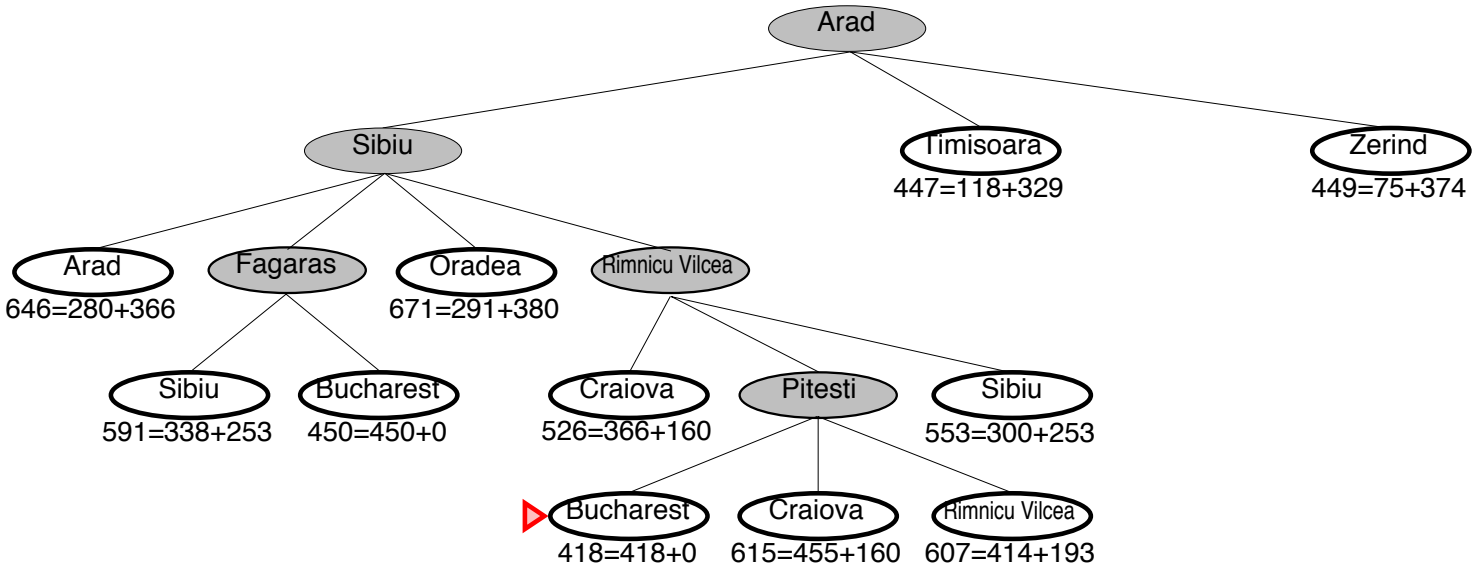


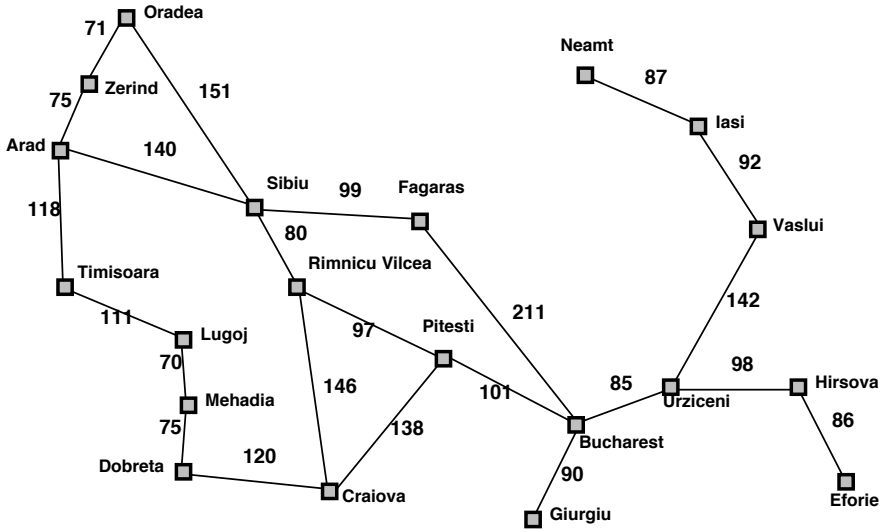


Straight–line distance to Bucharest

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

A* search example





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

Optimal path?