# ARTIFICIAL INTELLIGENCE

2023/2024 Semester 2

**Solving Problems by Searching:**

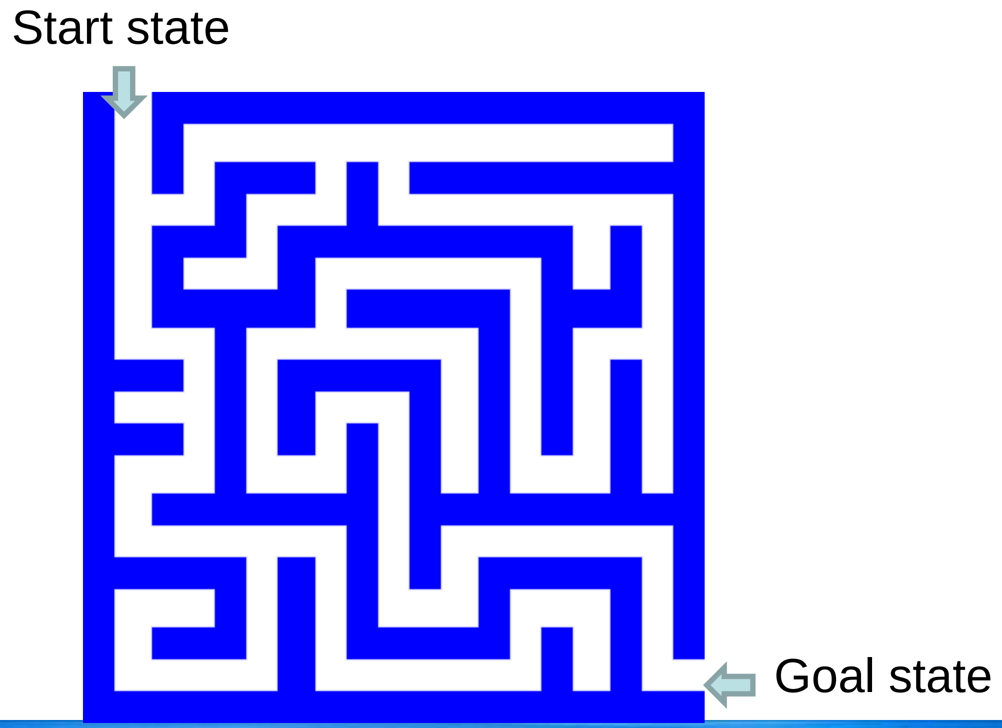
Chapter 3

# Outline

- **Problem-solving agents**
- **Example problems**
- **Searching for solutions**
- **Uninformed search strategies**
- **Informed (Heuristic) search strategies**

# Search

- We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete**, **known** environments
- Example:



# Search

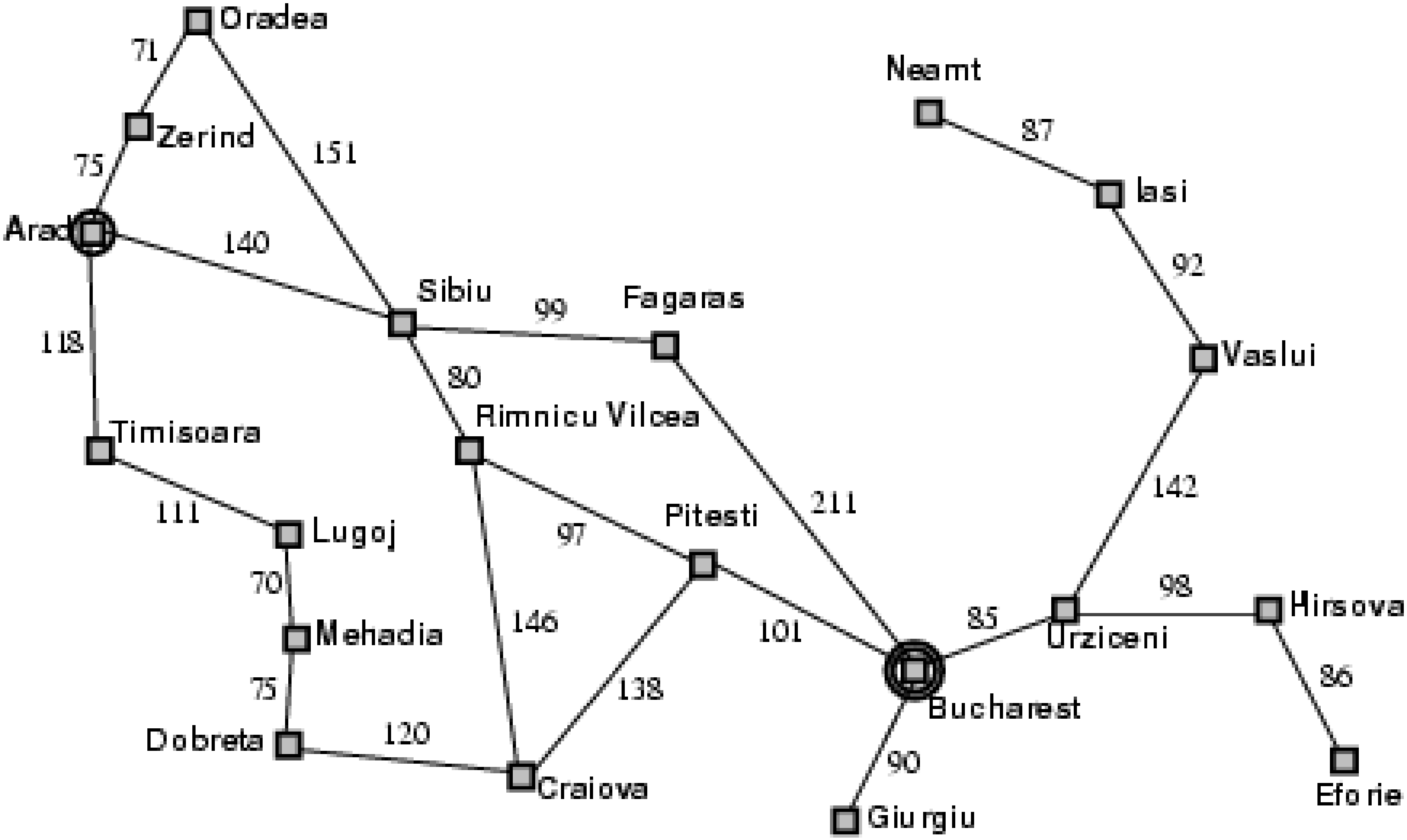
- We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete**, **known** environments
  - The solution is a fixed sequence of actions
  - Search is the process of looking for the sequence of actions that reaches the goal
  - Once the agent begins executing the search solution, it can ignore its percepts (**open-loop system**)

# Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

# Example: Travel in Romania



# Example: Travel in Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest

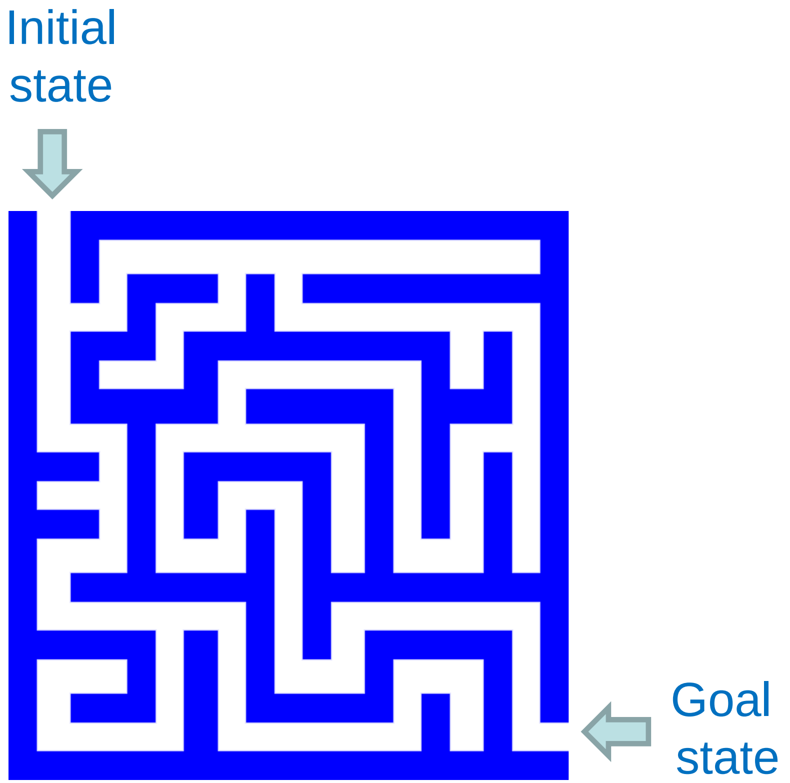
- **Formulate goal:** be in Bucharest

- **Formulate problem:**
  - states: various cities
  - actions: drive between cities

- **Find solution:**
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

# Search problem components

- **Initial state**
- **Actions**
- **Transition model**
  – What is the result of performing a given action in a given state?
- **Goal test**
- **Path cost**
  – Assume that it is a sum of nonnegative *step costs*

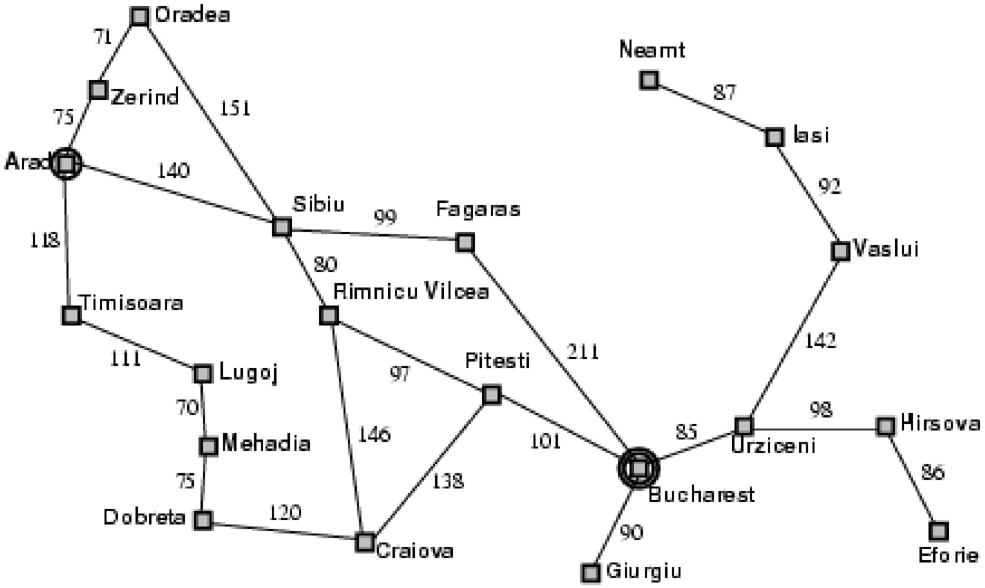



- The **optimal solution** is the sequence of actions that gives the lowest path cost for reaching the goal

# Example: Romania

- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

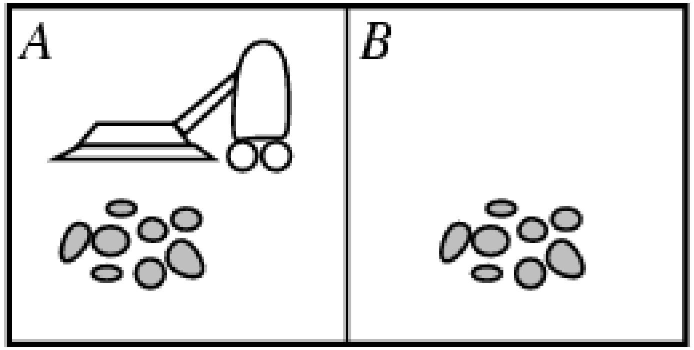

- **Initial state**
  - Arad
- **Actions**
  - Go from one city to another
- **Transition model**
  - If you go from city A to city B, you end up in city B
- **Goal state**
  - Bucharest
- **Path cost**
  - Sum of edge costs



# State space

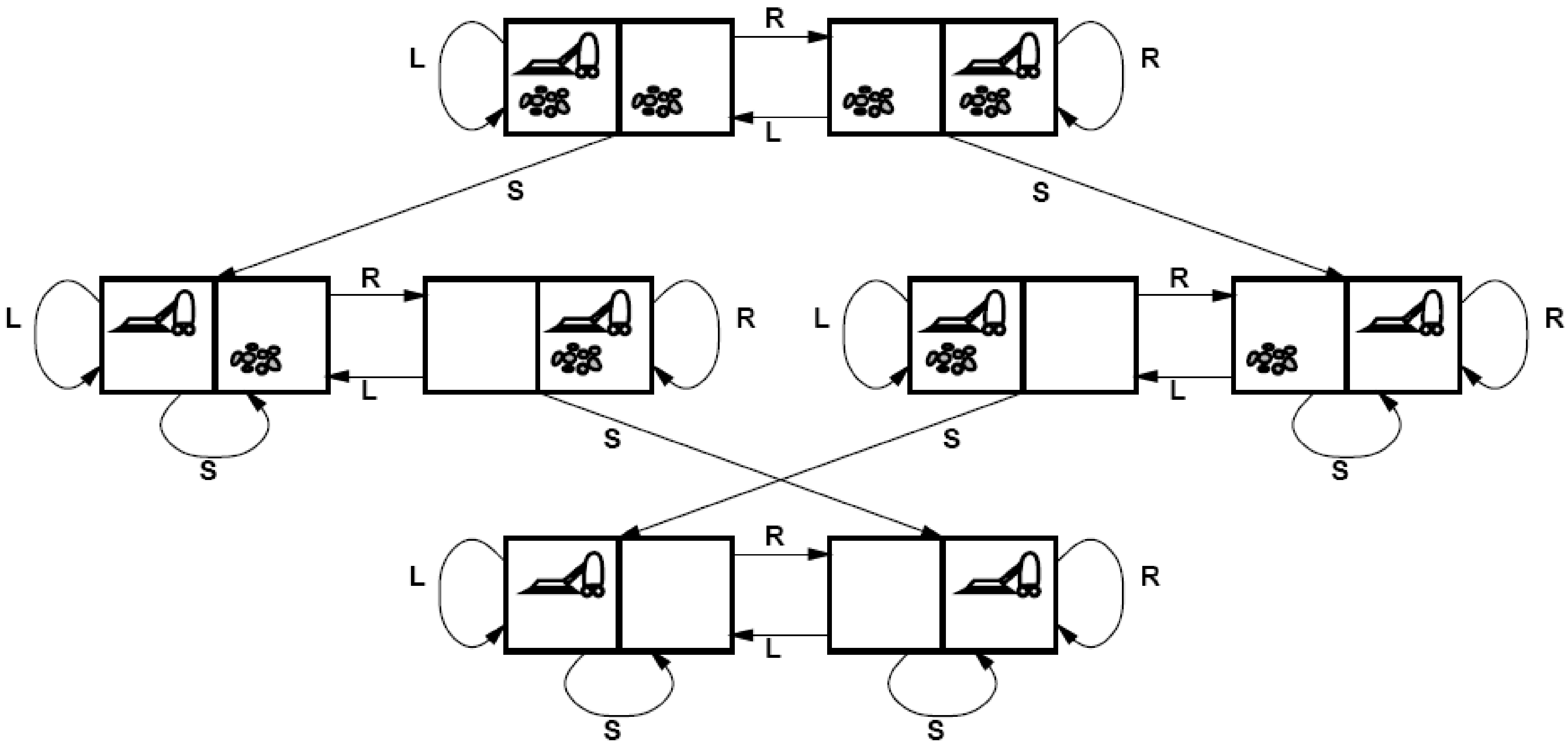
- The initial state, actions, and transition model define the **state space** of the problem
  - The set of all states reachable from initial state by any sequence of actions
  - Can be represented as a **directed graph** where the nodes are states and links between nodes are actions
- What is the state space for the Romania problem?

# Example: Vacuum world



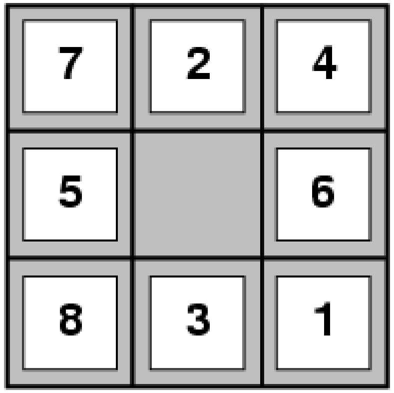

- **States**
  - Agent location and dirt location
  - How many possible states? $2\times2^2=8$
  - What if there are *n* possible locations?
- **Actions**
  - Left, right, suck
- **Transition model**

# Vacuum world state space graph

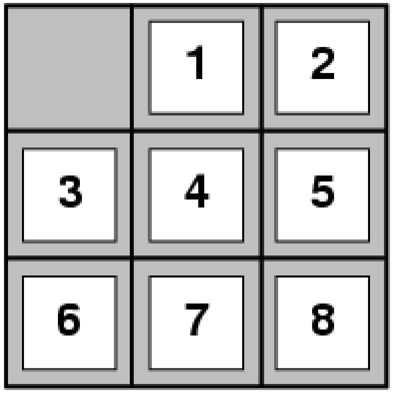


# Example: The 8-puzzle

- **States**
  - Locations of tiles
    - 8-puzzle: 181,440 states
    - 15-puzzle: 1.3 trillion states
    - 24-puzzle: $10^{25}$ states
- **Actions**
  - Move blank left, right, up, down
- **Transition model**
  - Given a state and action, returns the resulting state
- **Path cost**
  - 1 per move
- Finding the optimal solution of n-Puzzle is NP-hard




Start State




Goal State

## 15 - Puzzle

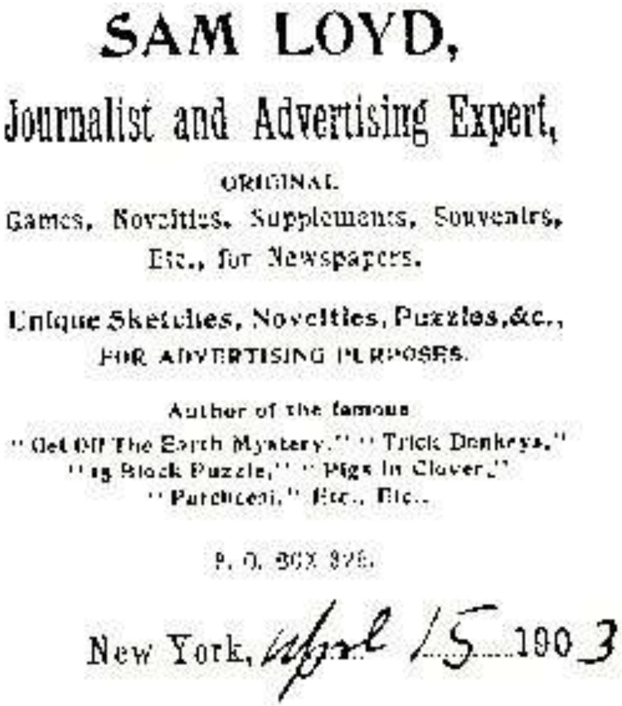
Sam Loyd 自掏腰包悬赏，第一个解决下面 15 数码问题的人将得到 $1,000 的赏金：

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

? →

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 15 | 14 | |

8



SAM LOYD,

Journalist and Advertising Expert,

ORIGINAL

Games, Novelties, Supplements, Souvenirs, Etc., for Newspapers.

Unique Sketches, Novelties, Puzzles, &c., FOR ADVERTISING PURPOSES.

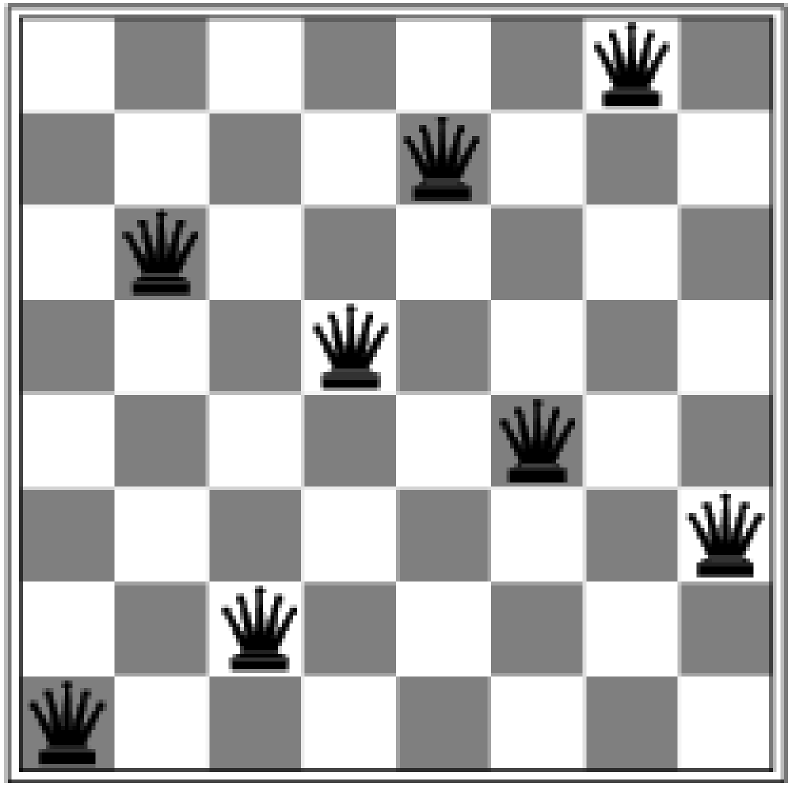
Author of the famous

"Get Off The Earth Mystery," "Trick Donkeys," "15 Block Puzzle," "Pigs In Clover," "Parcheesi," Etc., Etc.,

P. O. BOX 826.

New York, April 15 1903

# Example: 8-queens problem

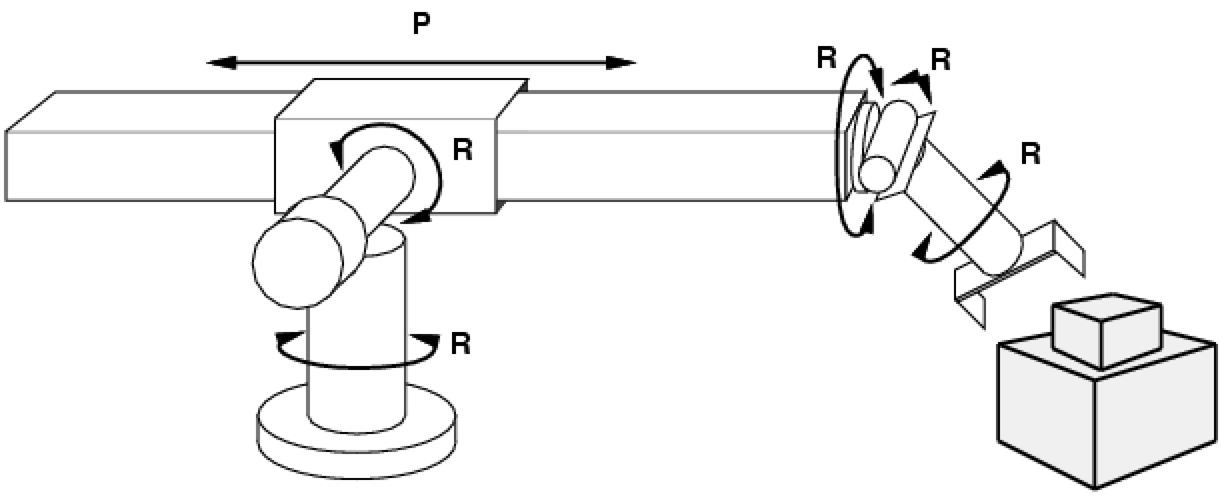


Place 8-queens in the position such that no queen can attack the others

# Example: 8-queens problem

- **States**
  - Any arrangement of 0 to 8 queens on the board is a state
- **Initial state**
  - No queens on the board
- **Actions**
  - Add a queen to any empty
- **Transition model**
  - Returns the board with a queen added to the specified square
- **Goal test**
  - 8 queens are on the board
  - None attacked

# Example: Robot motion planning




- **States**
  – Real-valued coordinates of robot joint angles
- **Actions**
  – Continuous motions of robot joints
- **Goal state**
  – Desired final configuration (e.g., object is grasped)
- **Path cost**
  – Time to execute, smoothness of path, etc.

# Search

- Given:
  - **Initial state**
  - **Actions**
  - **Transition model**
  - **Goal state**
  - **Path cost**

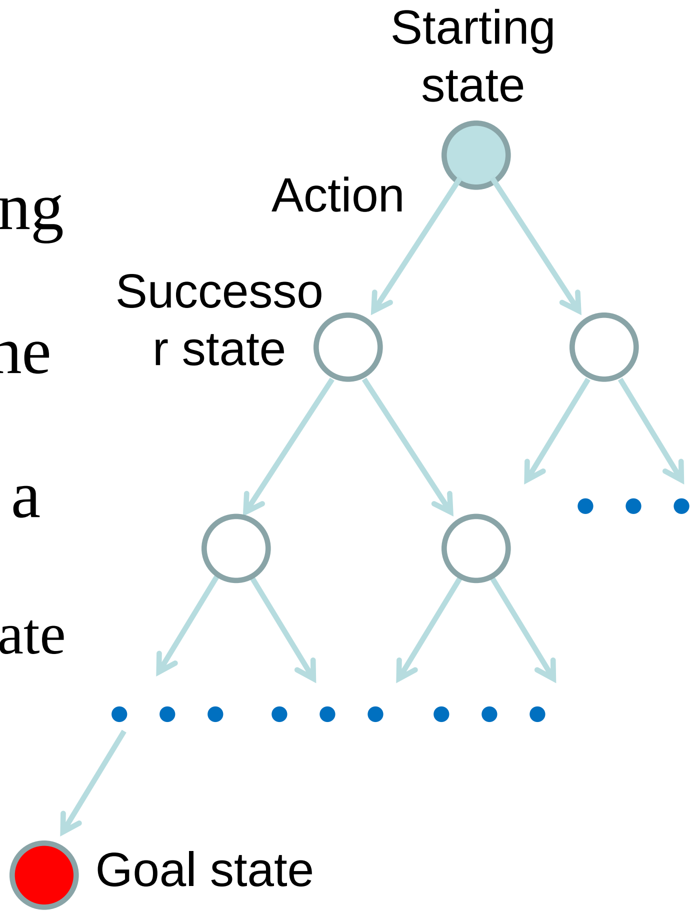
- How do we find the optimal solution?
  - How about building the state space and then using Dijkstra's shortest path algorithm?
    - The state space may be huge!
    - Complexity of Dijkstra's is $O(E + V \log V)$, where $V$ is the size of the state space

# Tree Search

- Let’s begin at the start node and **expand** it by making a list of all possible successor states
- Maintain a **frontier** or a list of unexpanded states
- At each step, pick a state from the frontier to expand
- Keep going until you reach the goal state
- Try to expand as few states as possible

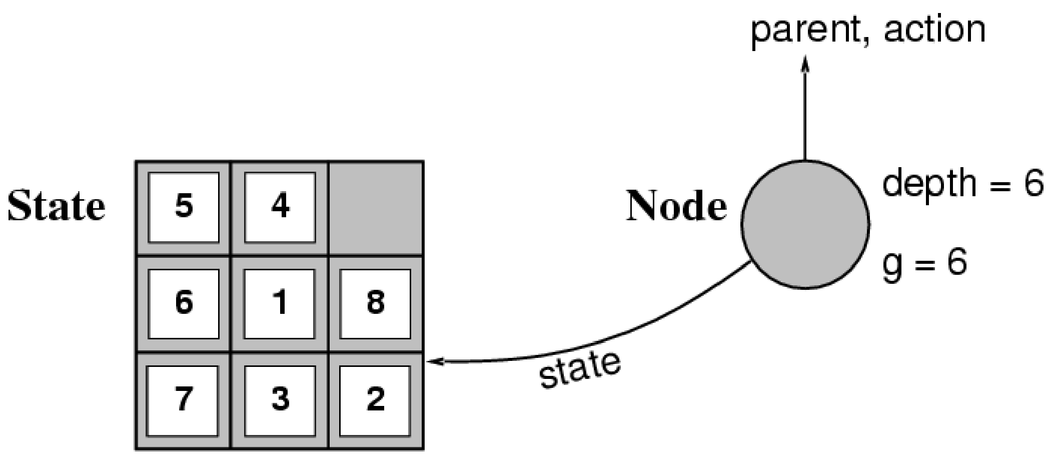
# Search tree

- “What if” tree of possible actions and outcomes
- The root node corresponds to the starting state
- The children of a node correspond to the **successor states** of that node’s state
- A path through the tree corresponds to a sequence of actions
  – A solution is a path ending in the goal state
- Nodes vs. states
  – A state is a representation of a physical configuration, while a node is a data structure that is part of the search tree



# states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, (depth)



# Tree Search Algorithm Outline

- Initialize the **fringe(frontier)** using the **starting state**
- While the fringe is not empty
  - Choose a fringe node to expand according to **search strategy**
  - If the node contains the **goal state**, return solution
  - Else **expand** the node and add its children to the fringe

# Tree search algorithms

**function** TREE-SEARCH (*problem*, *fringe*) returns a solution, or failure

initialize the frontier(fringe) using the initial state of *problem*

**loop do**

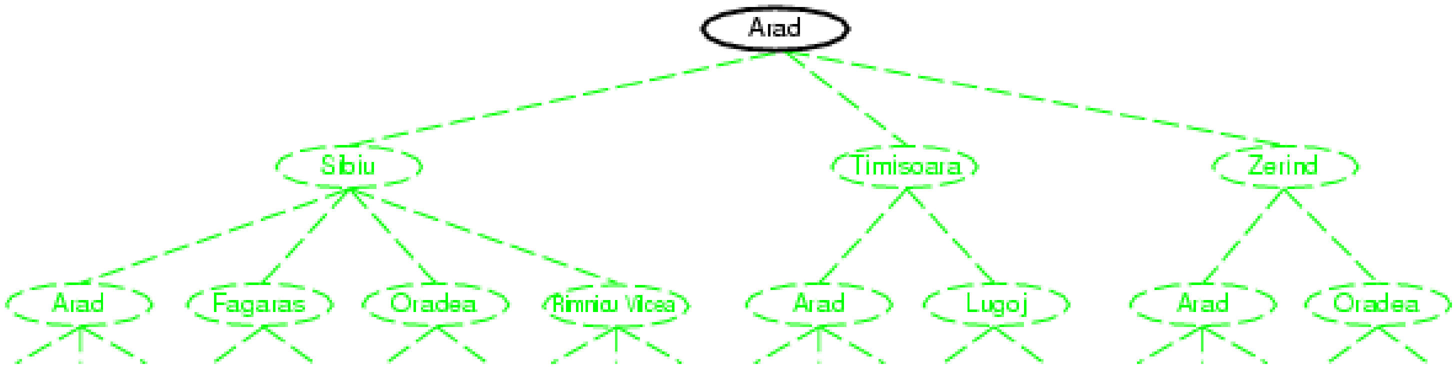
**if** the frontier(fringe) is empty **then return** failure

choose a leaf node and leave it from the frontier (fringe)

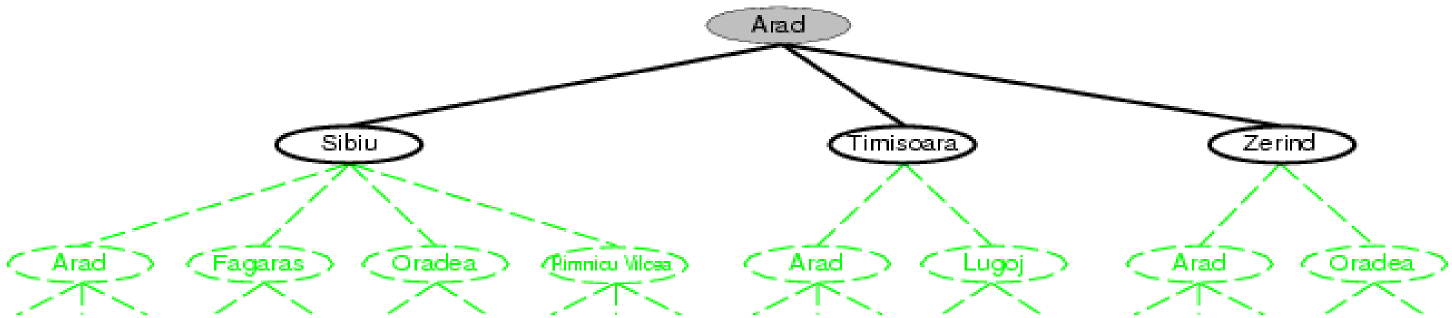
**if** the node contains a goal state **then return** the corresponding solution

expand the chosen node, adding the resulting nodes to the frontier(fringe)

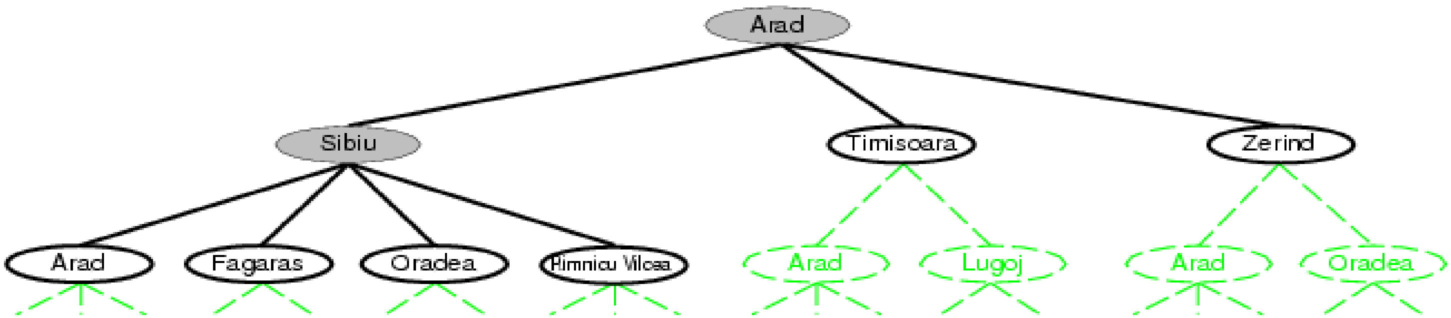
# Tree search example



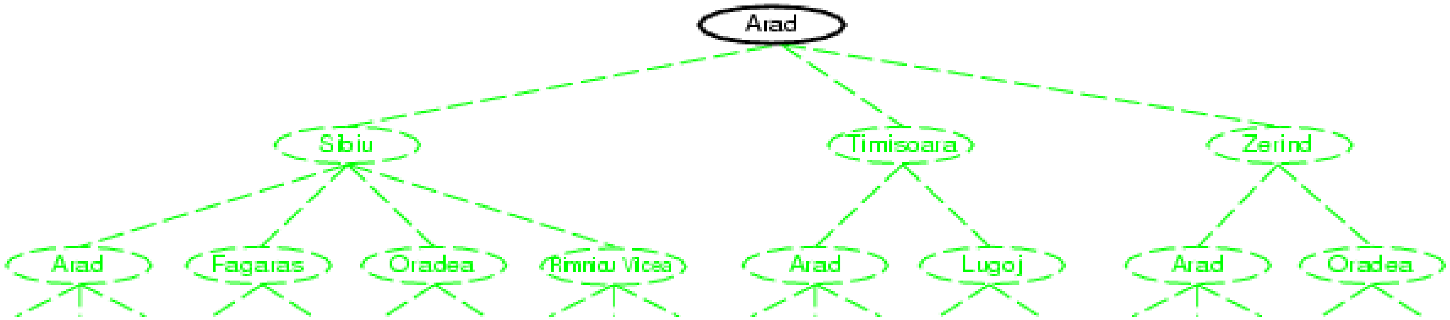
# Tree search example

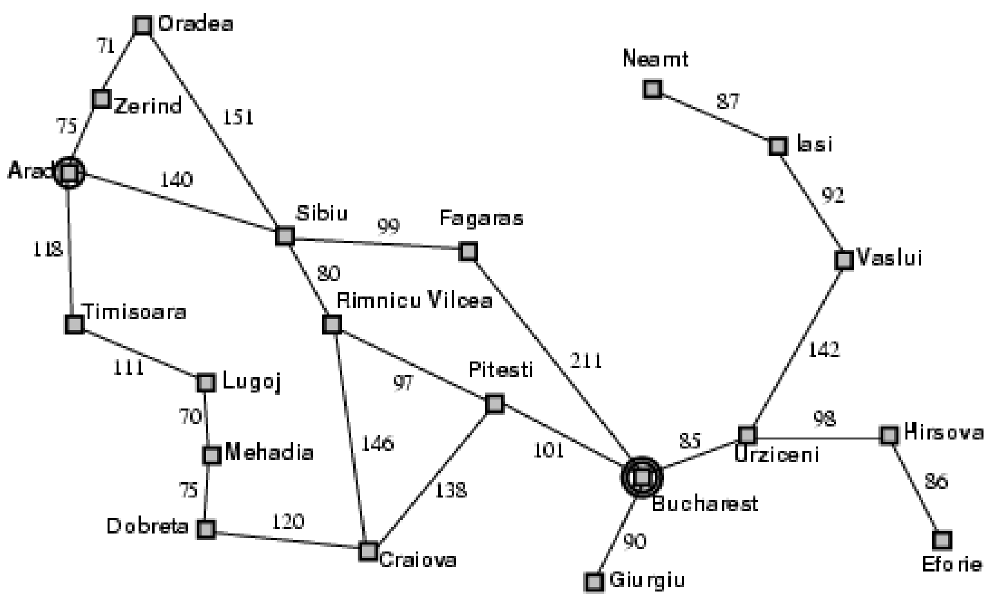


# Tree search example

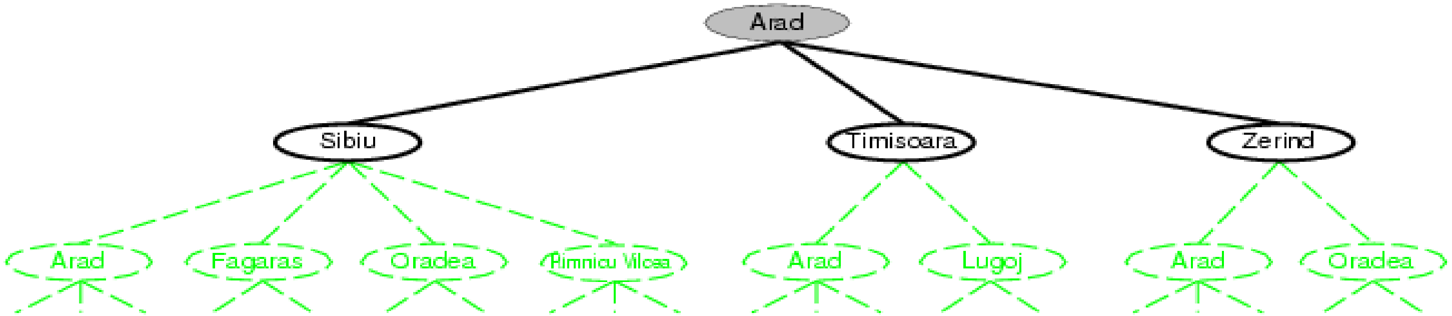


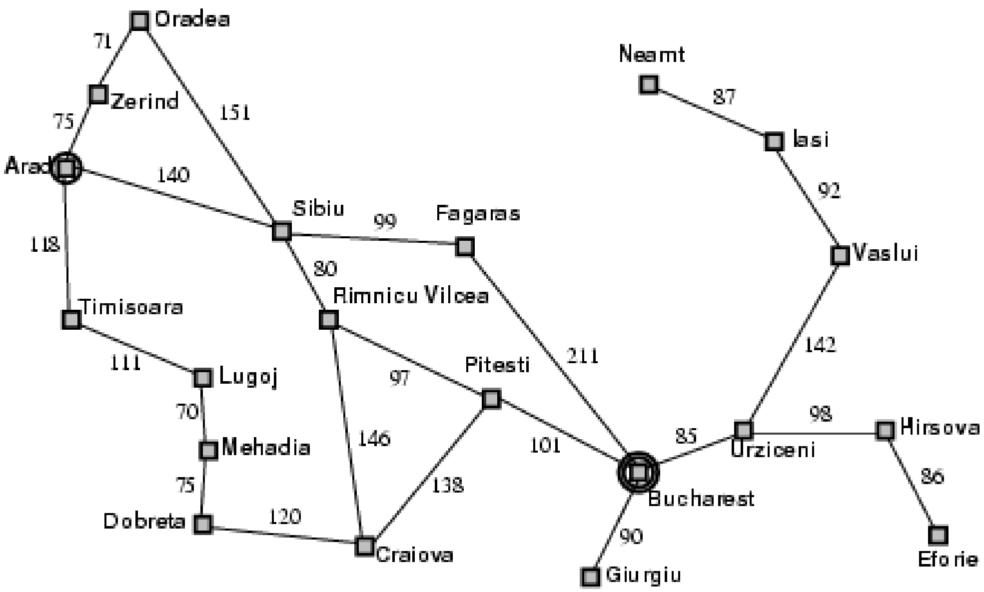
# Tree search example



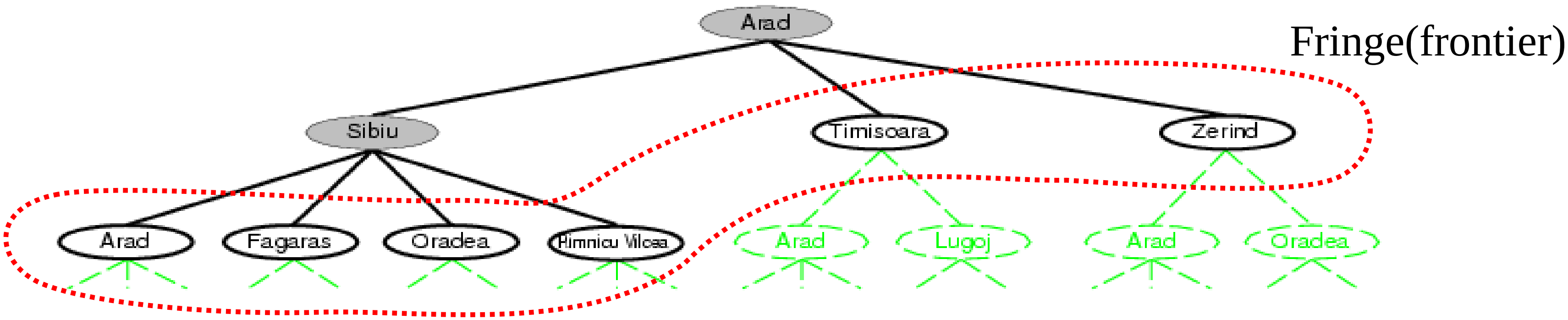



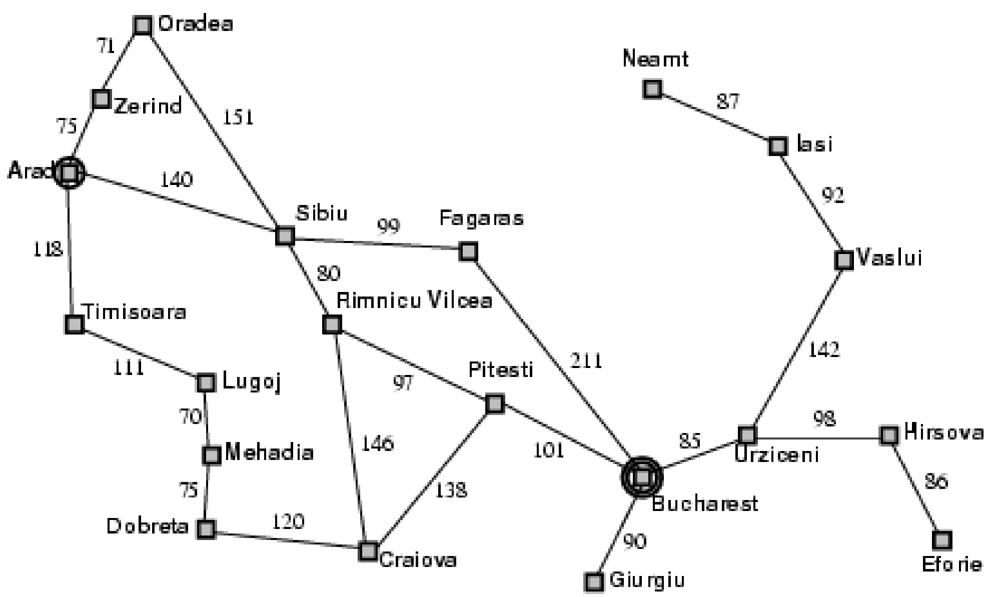
# Tree search example



# Tree search example



# Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

# Search strategies

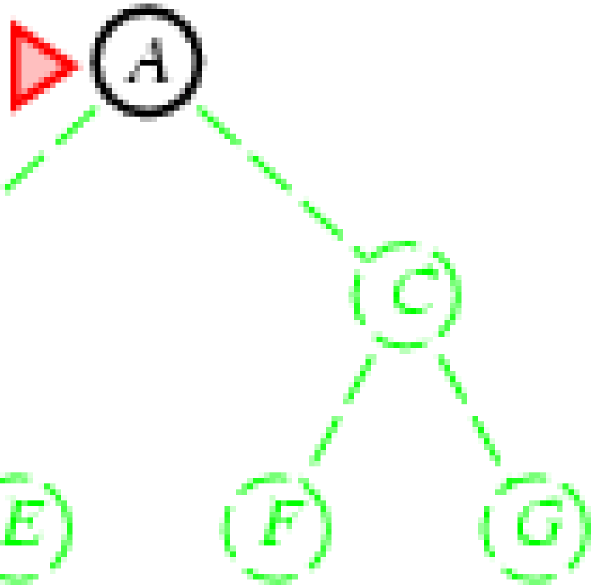
- A **search strategy** is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - **Completeness:** does it always find a solution if one exists?
  - **Optimality:** does it always find a least-cost solution?
  - **Time complexity:** number of nodes generated
  - **Space complexity:** maximum number of nodes in memory
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum length of any path in the state space (may be infinite)

# Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search

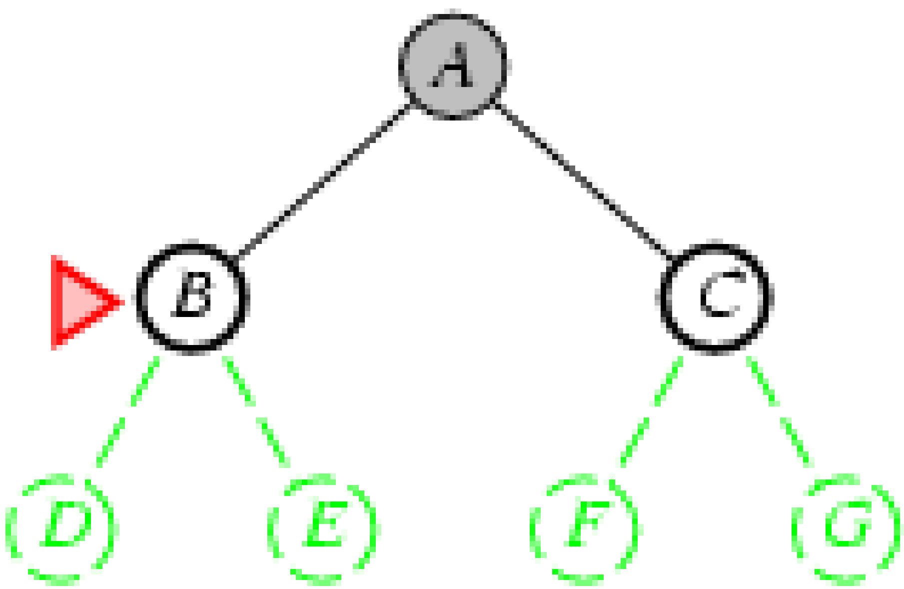
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end
  - 



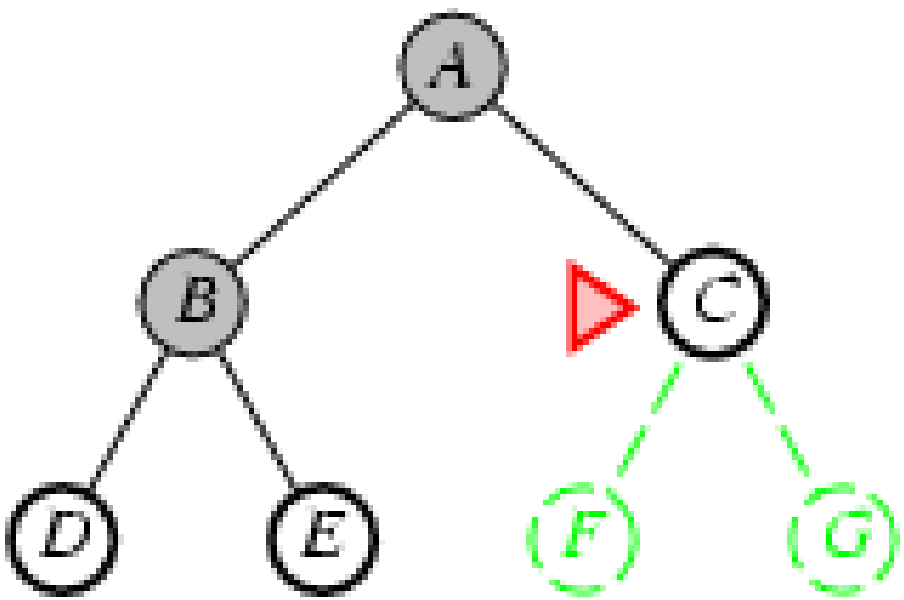
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end
  - 



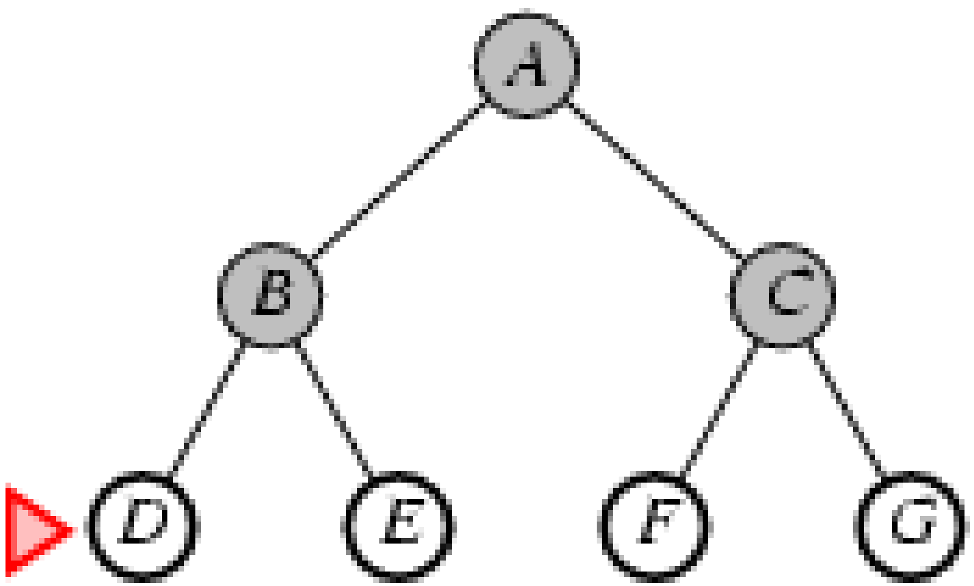
# Breadth-first search

- 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
  - 



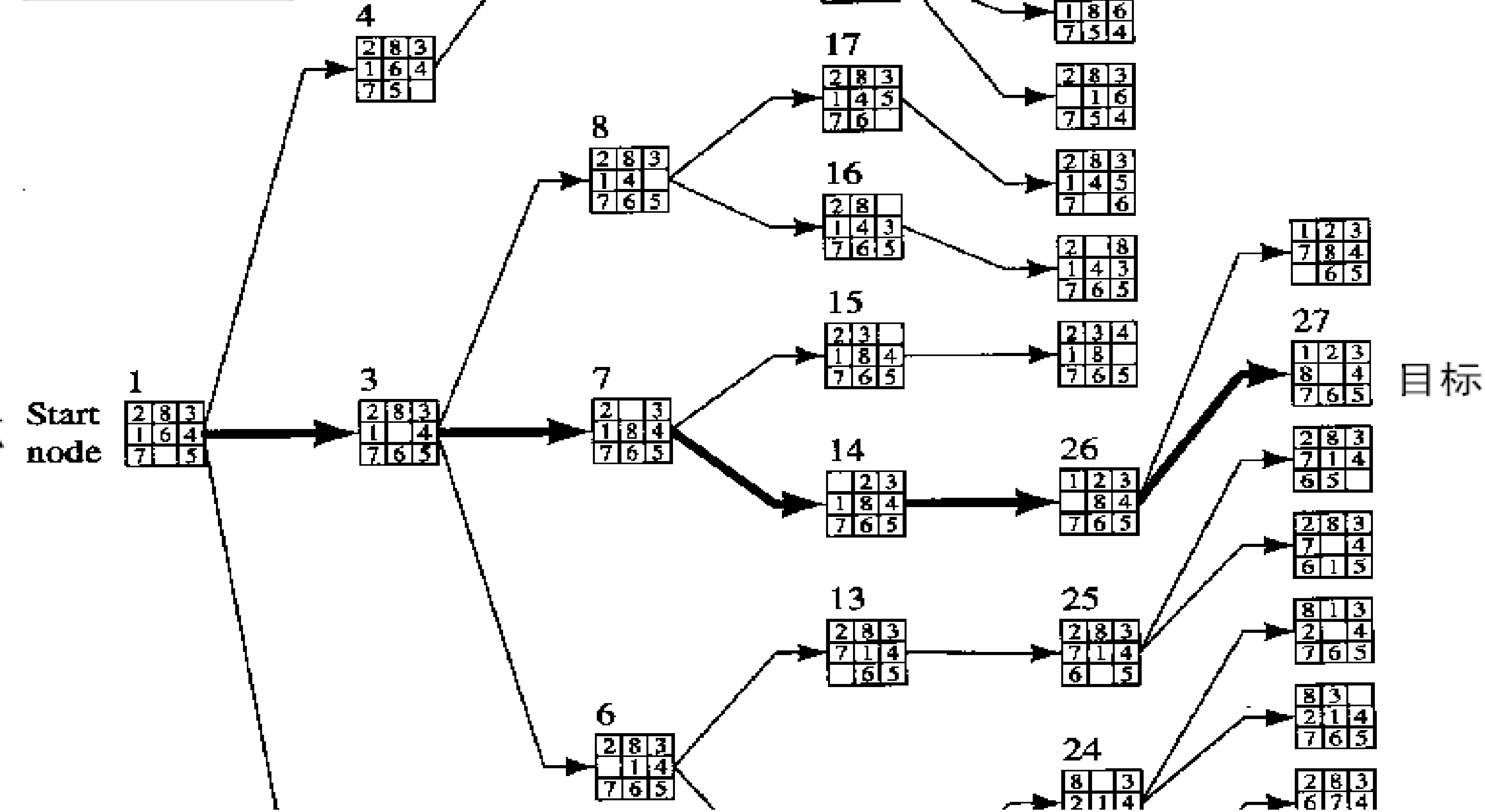
# Breadth-First Search

- Procedure
  1. Apply all possible operators (*successor function)* to the start node.
  2. Apply all possible operators to all the direct successors of the start node.
  3. Apply all possible operators to their successors till goal node found.

  ♠ *Expanding* : applying successor function to a node

| | | |
|---|---|---|
| **2** | **8** | **3** |
| **1** | **6** | **4** |
| **7** | | **5** |

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |



# Properties of breadth-first search

- Complete? Yes (if $b$ is finite)

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

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

- Optimal? Yes (if cost = 1 per step)

- Space is the bigger problem (more than time)
-

# Time and Memory Requirement for BFS

| d | # Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 msec | 107 Kbytes |
| 4 | 11,110 | 1 1 msec | 10.6 Mbytes |
| 6 | $\sim 10^6$ | 1.1 seconds | 1 Gbytes |
| 8 | $\sim 10^8$ | 2 minutes | 103 Gbytes |
| 10 | $\sim 10^{10}$ | 3 hours | 10 Tbytes |
| 12 | $\sim 10^{12}$ | 13 days | 1 Pbytes |
| 14 | $\sim 10^{14}$ | 3.5 years | 99 pbytes |

assume: b = 10; 1,000,000 nodes/sec; 1000bytes/node

# Breadth-First Search

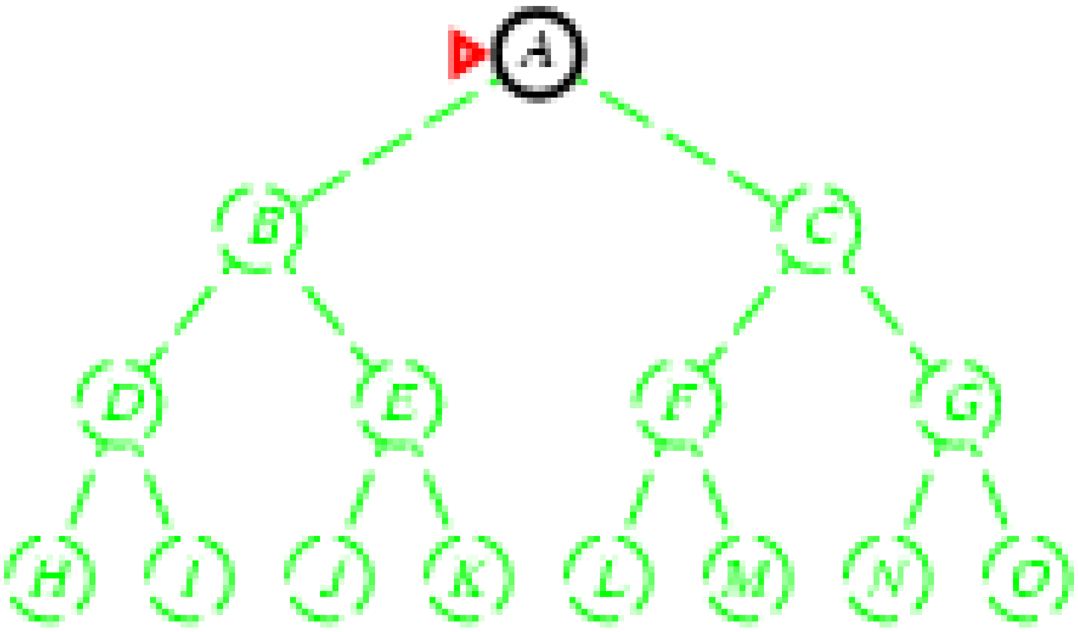
- **Advantage**
  – Finds the path of minimal length to the goal.
- **Disadvantage**
  – Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node
- ***Uniform-cost*** search [Dijkstra 1959]
  – Expansion by *equal cost* rather than equal depth

# Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation:**
  - ***frontier* = queue ordered by path cost**
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost $\geq \varepsilon$
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where $C^*$ is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

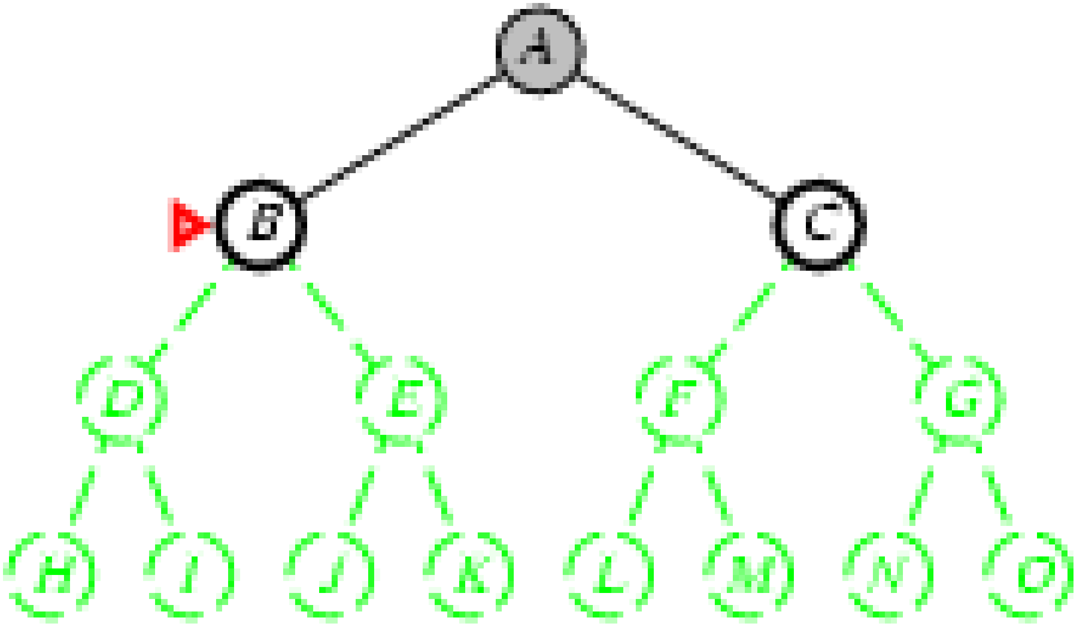
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  - 



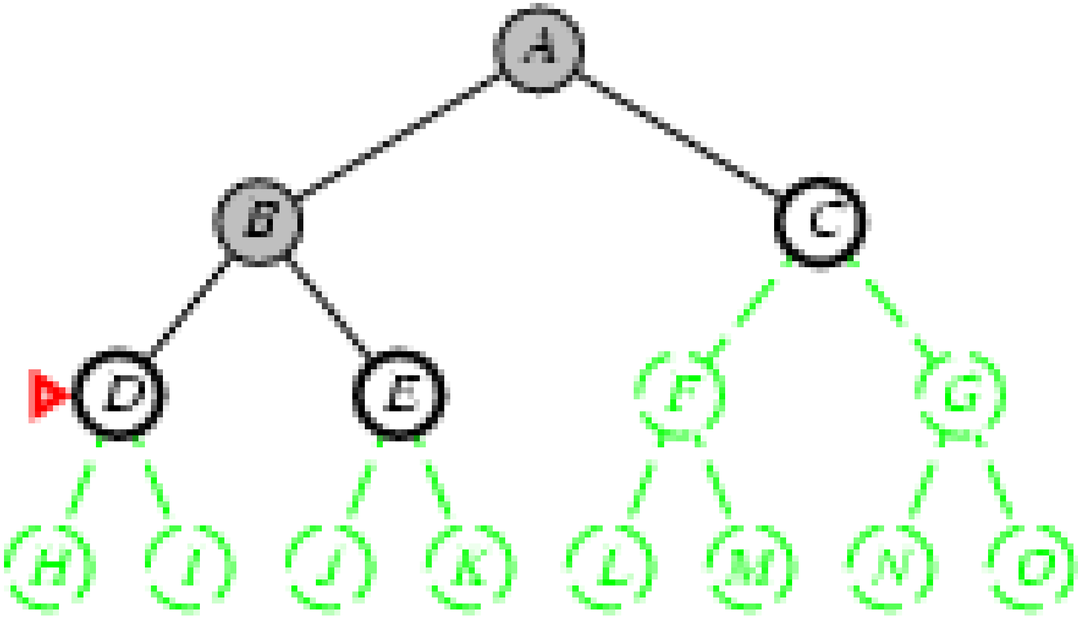
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier*= LIFO queue, i.e., put successors at front
  - 



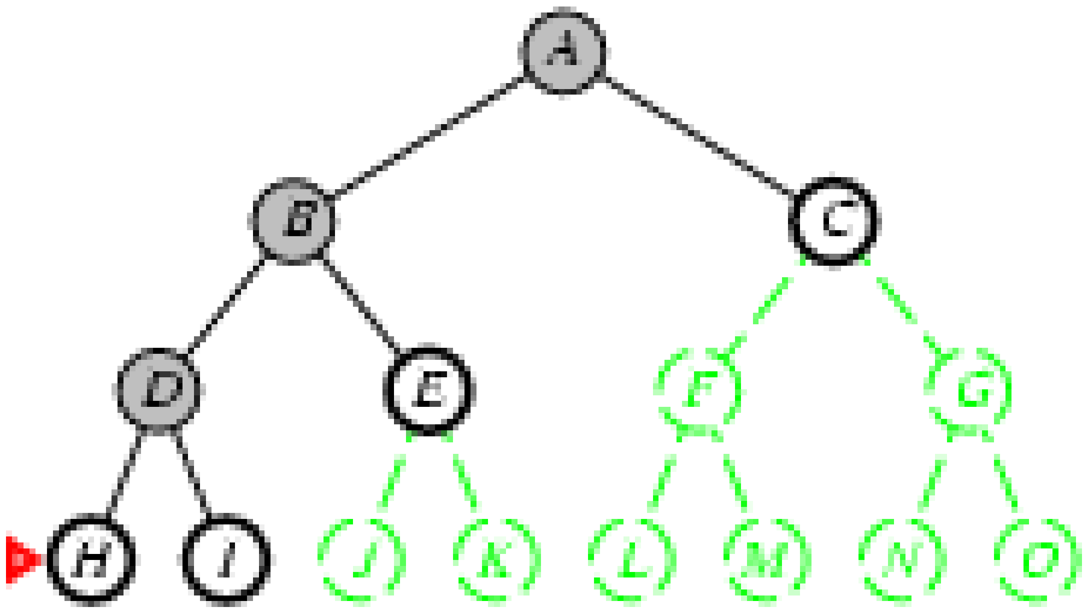
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



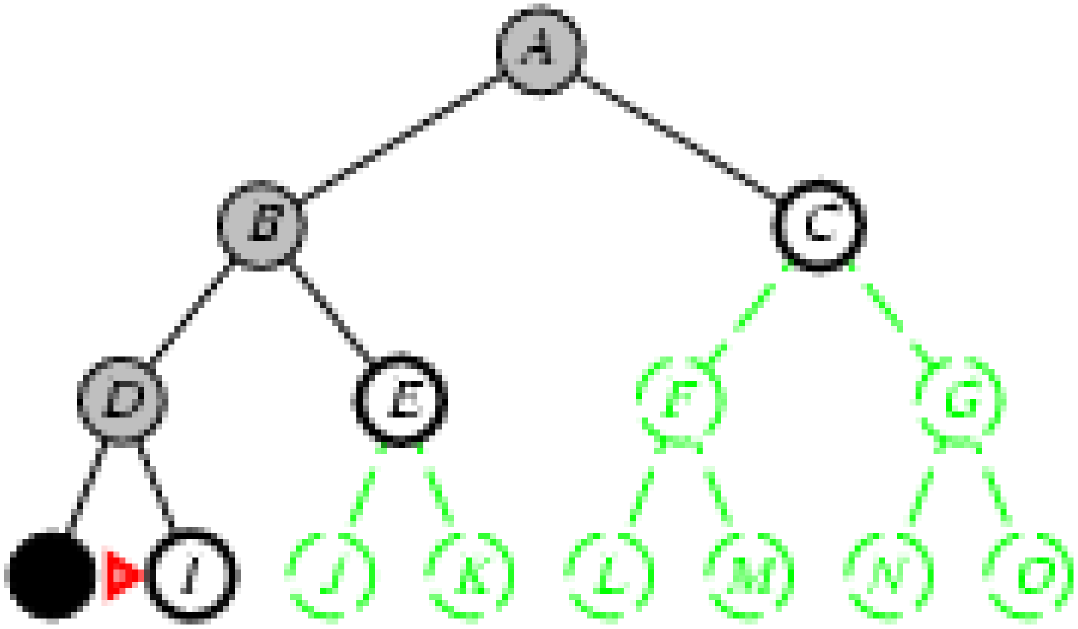
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



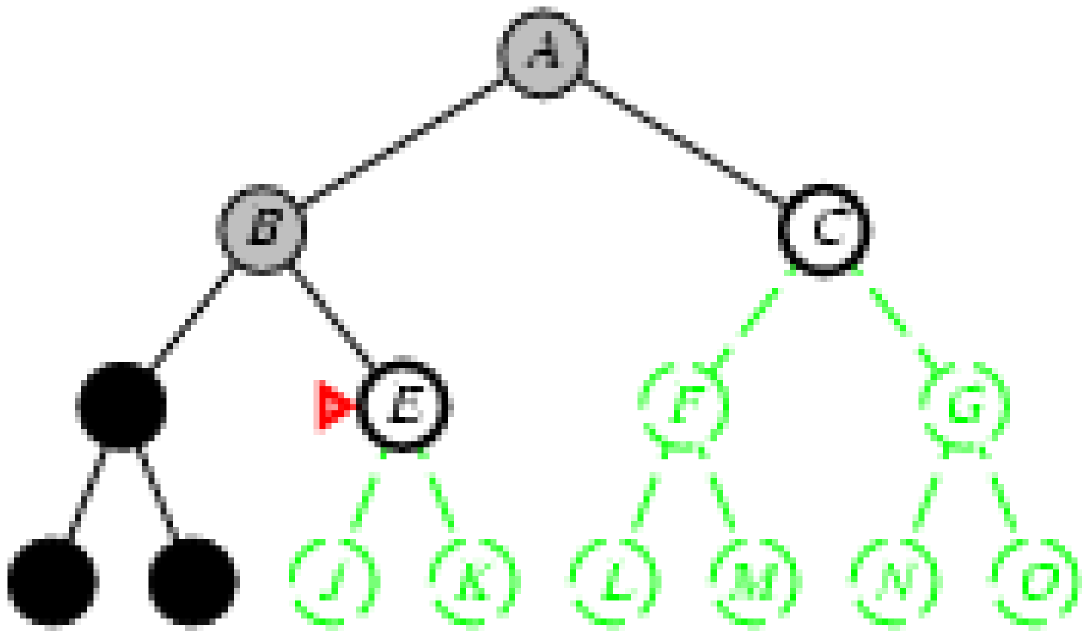
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



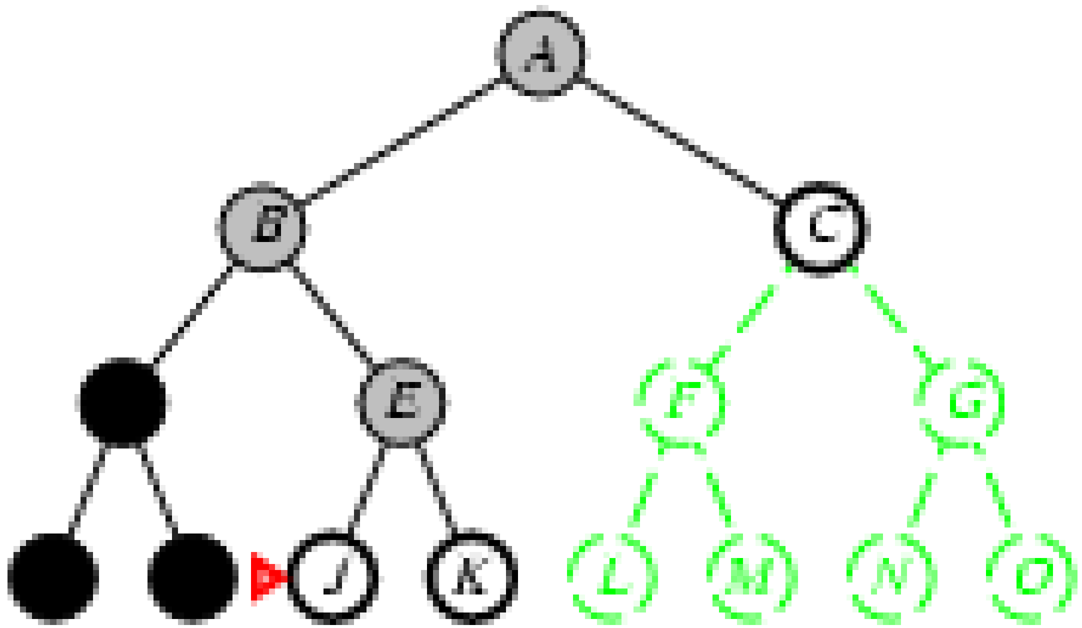
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier*= LIFO queue, i.e., put successors at front
  - 



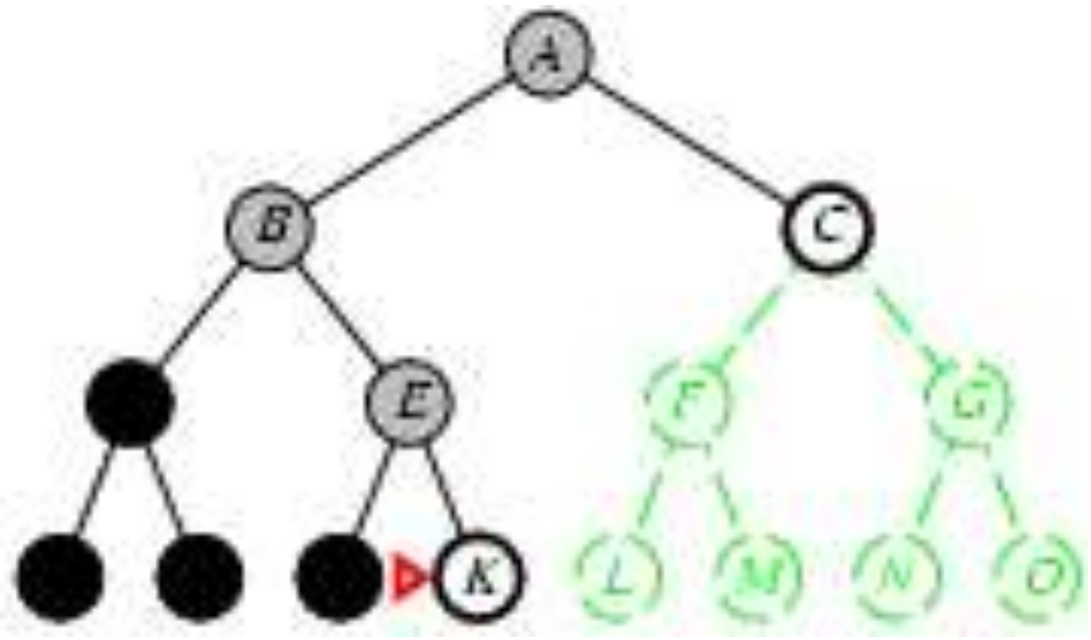
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  - 



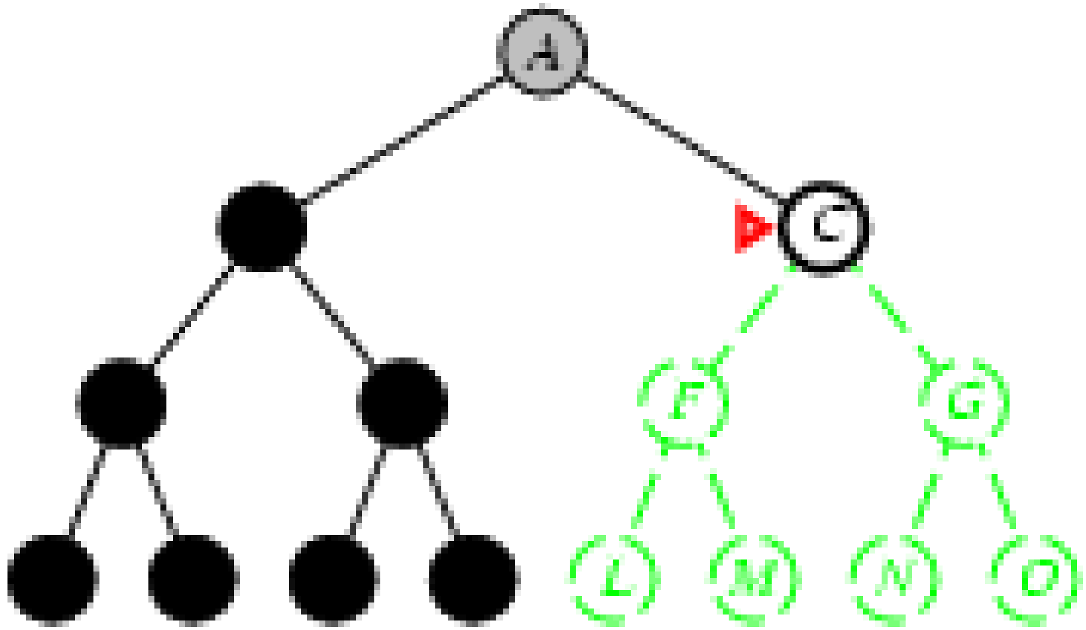
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  - 



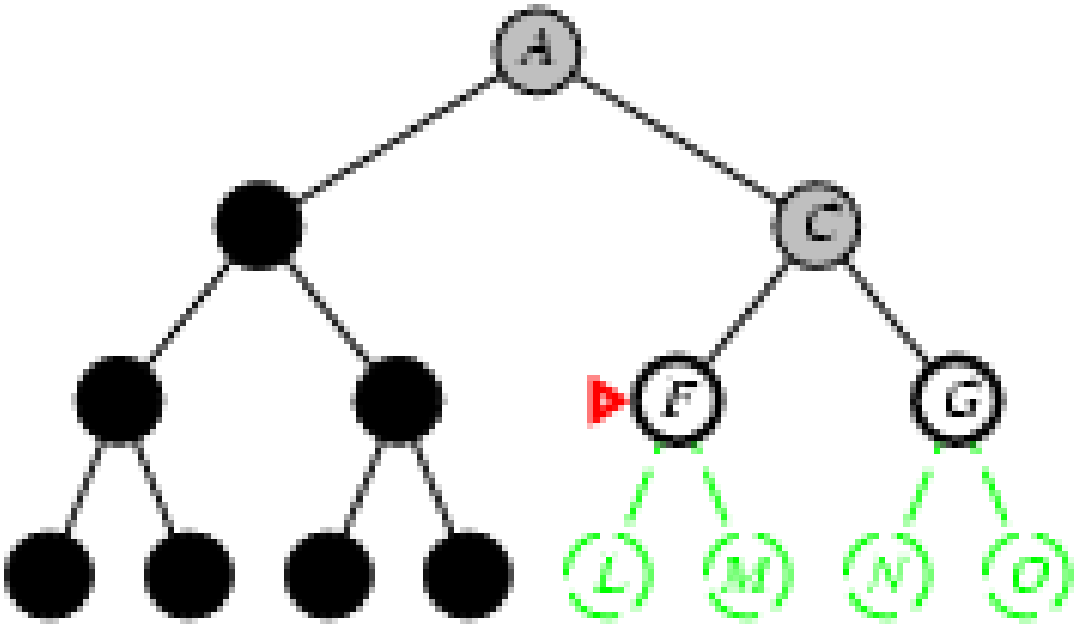
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  - 



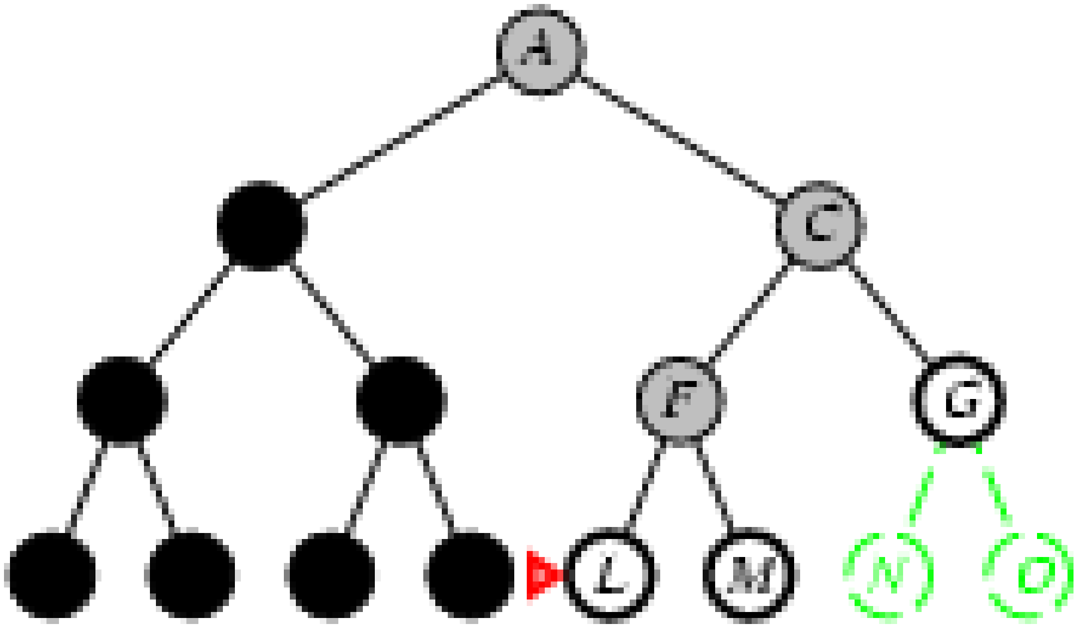
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  - 



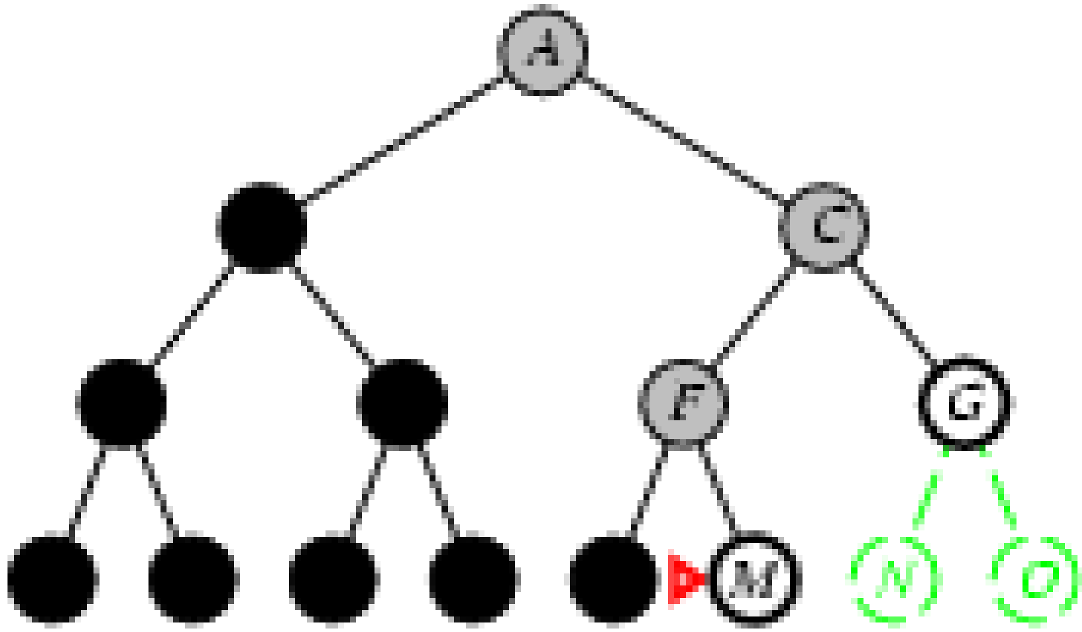
# Depth-first search

- 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, spaces with loops
  - Modify to avoid repeated states along path
  - 

    → complete in finite spaces

- Time? $O(b^m)$: terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first
  - 
- Space? $O(bm)$, i.e., linear space!

- Optimal? No

# Depth-limited search

## Depth-first search with depth limit l

i.e., nodes at depth *l* have no successors

```
function DEPTH-LIMITED-SEARCH (problem, limit) returns solution or
    fail/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem,
    limit)

function RECURSIVE-DLS (node, problem, limit) returns solution or
    fail/cutoff
  if problem.GOAL-TEST (node.STATE) then return SOLUTION(node)
  else if limit=0 then return cutoff
  else cutoff-occurred? ← 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-occurred? ← true
     else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

# Iterative deepening depth-first search

- Use DFS as a subroutine
  1. Check the root
  2. Do a DFS searching for a path of length 1
  3. If there is no path of length 1, do a DFS searching for a path of length 2
  4. If there is no path of length 2, do a DFS searching for a path of length 3…

# Iterative deepening depth-first search

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

**inputs**: *problem*, a problem

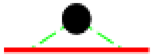
**for** *depth* ← 0 **to** ∞ **do**

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

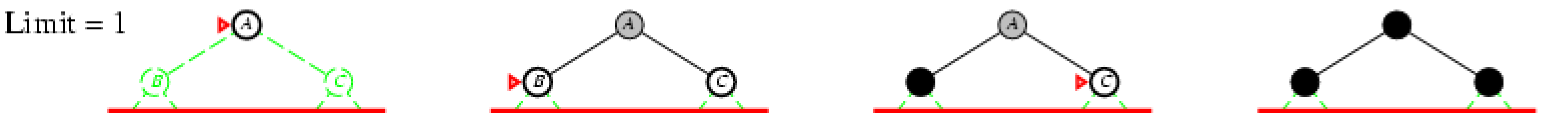
**if** *result* ≠ cutoff **then return** *result*

# Iterative deepening depth-first search

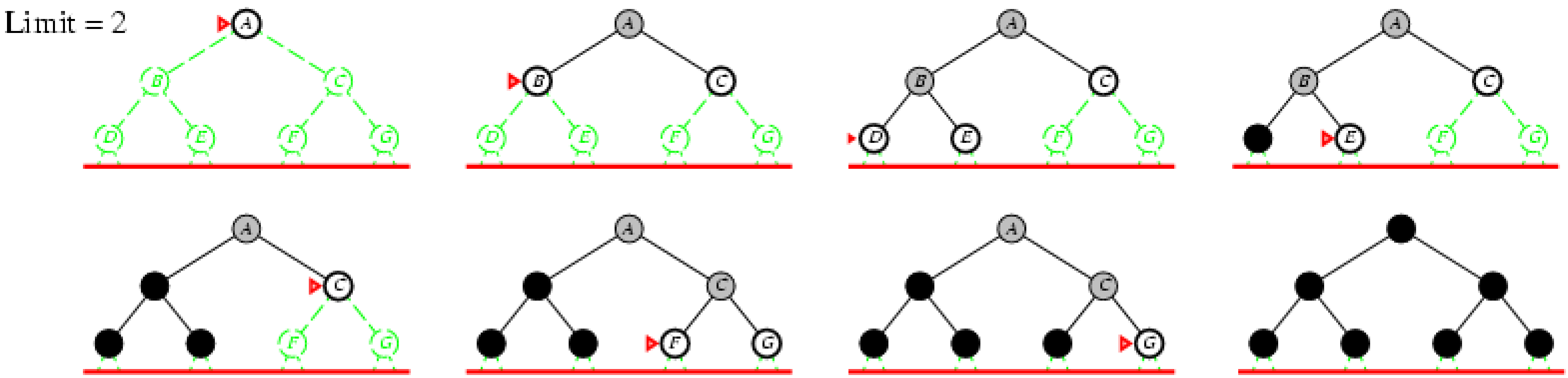
Limit = 0

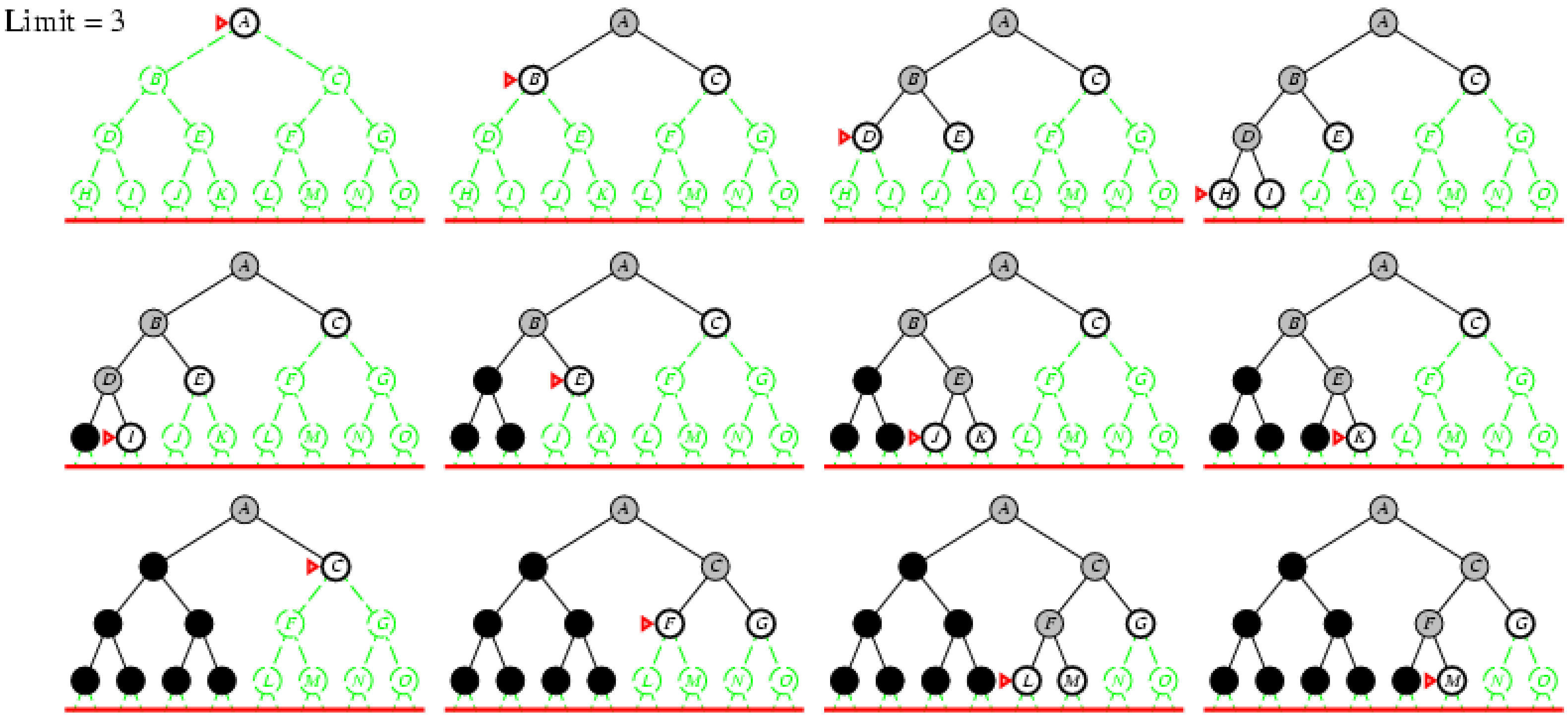
# Iterative deepening depth-first search



# Iterative deepening depth-first search

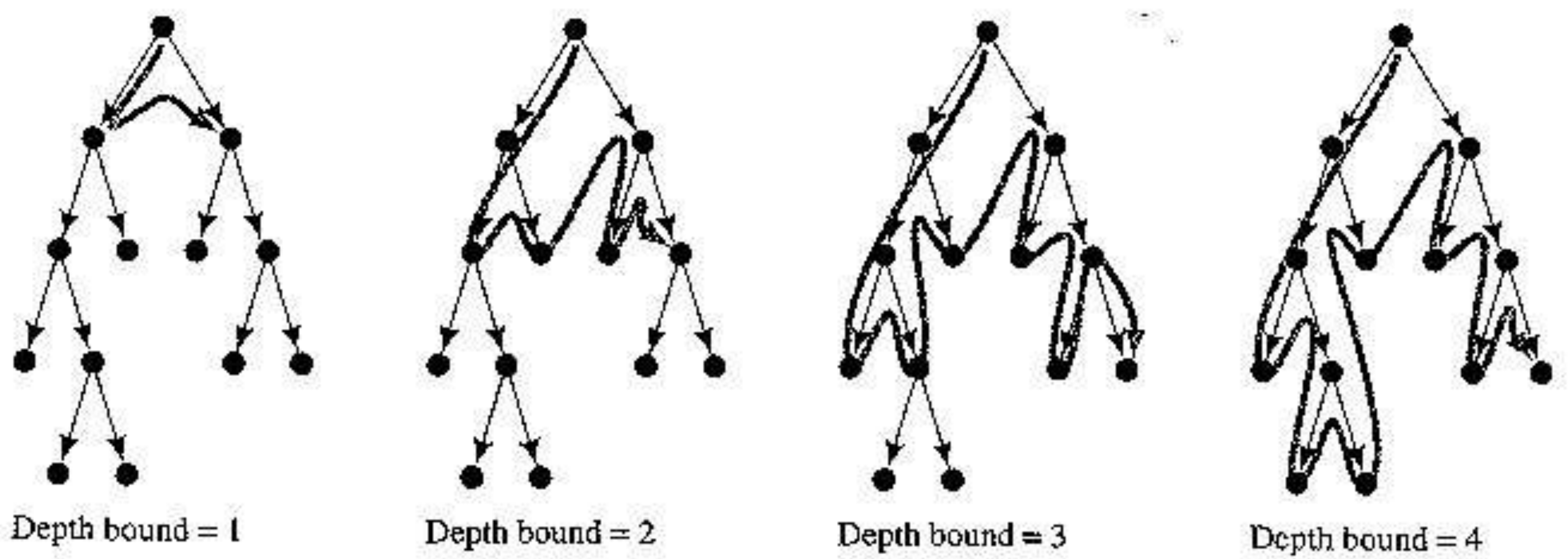


# Iterative deepening depth-first search



# Iterative Deepening

- Advantage
  - Linear memory requirements of depth-first search
  - Guarantee for goal node of minimal depth
- Procedure
  - Successive depth-first searches are conducted – each with depth bounds increasing by 1



# Iterative deepening search

- Number of nodes generated in a breadth-first search to depth *d* with branching factor *b*:

$$N_{BFS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

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

$$N_{IDS} = d\,b + (d\text{-}1)b^{\wedge 2} + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{BFS}$ = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110
  - $N_{IDS}$ = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

- Overhead = (123,450 - 111,110)/111,110 = 11%

# Iterative deepening depth-first search

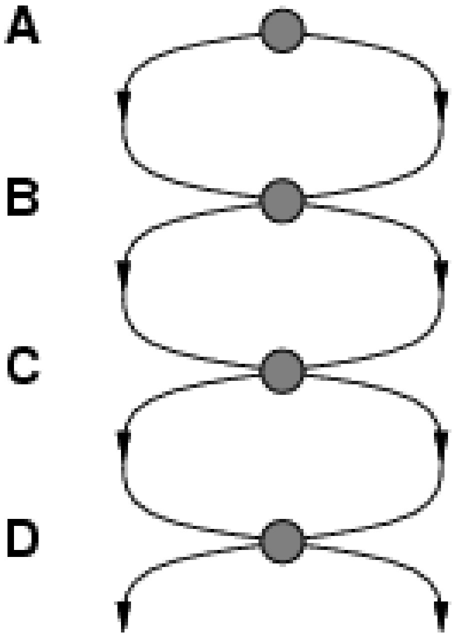
- Complete? Yes
- Time? $d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

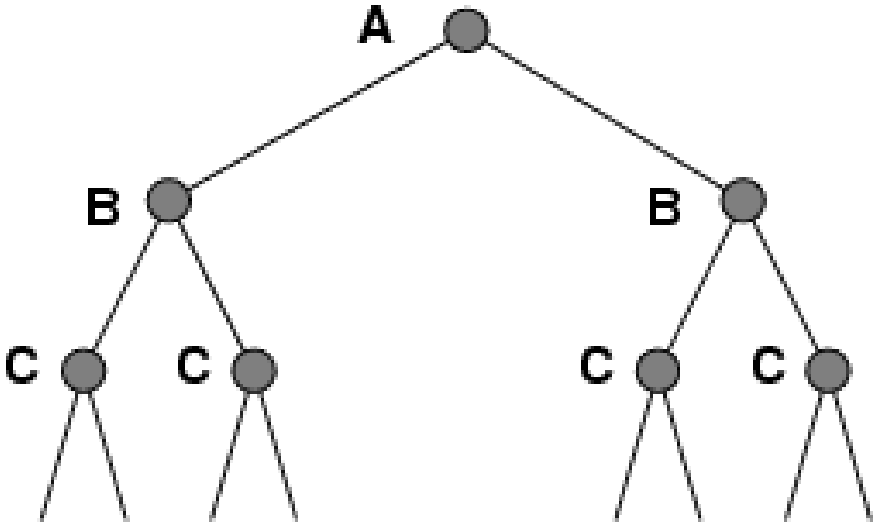
# Summary of algorithms

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening |
|---|---|---|---|---|---|
| Complete? | Yes | Yes | No | No | Yes |
| Time | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ |
| Space | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(bl)$ | $O(bd)$ |
| Optimal? | Yes | Yes | No | No | Yes |

# Problem: Repeated states

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



# Problem: Repeated states

- To handle repeated states:
  - Keep an **explored set (also known as the closed list)**; add each node to the explored set every time you expand it
  - Every time you add a node to the frontier, check whether it already exists in the frontie with a higher path cost, and if yes, replace that node with the new one

# Graph search

**function** GRAPH-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 leave 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

# Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

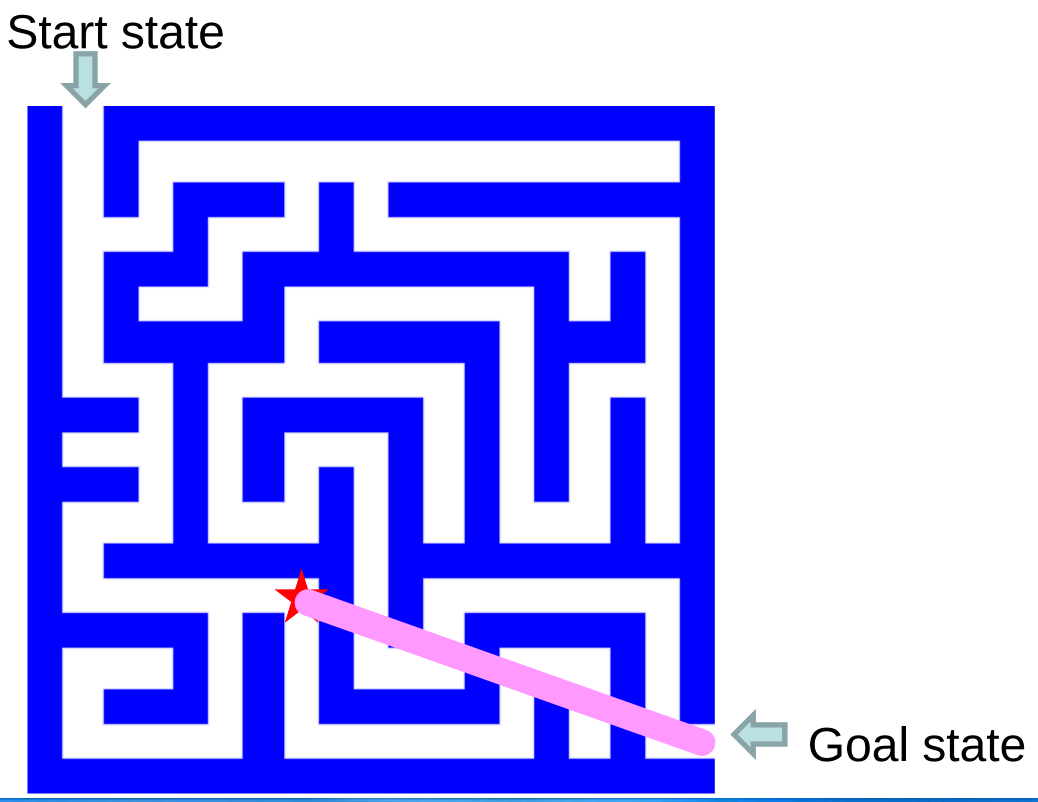
# Informed search

- Idea: give the algorithm "hints" about the desirability of different states
  - Use an *evaluation function* to rank nodes and select the most promising one for expansion

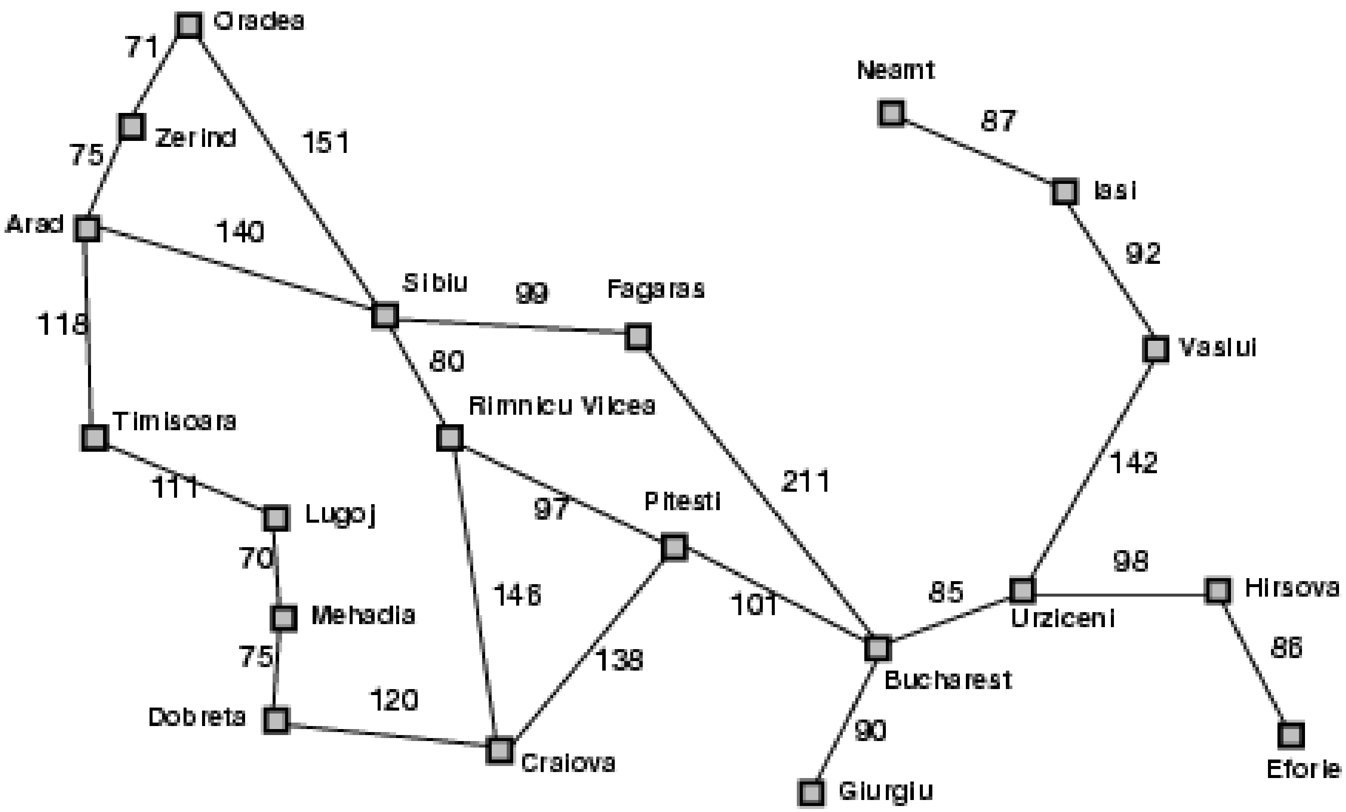
- Greedy best-first search
- A* search

# Heuristic function

- Heuristic function $h(n)$ estimates the cost of reaching goal from node $n$
- Example:



# Heuristic for the Romania problem



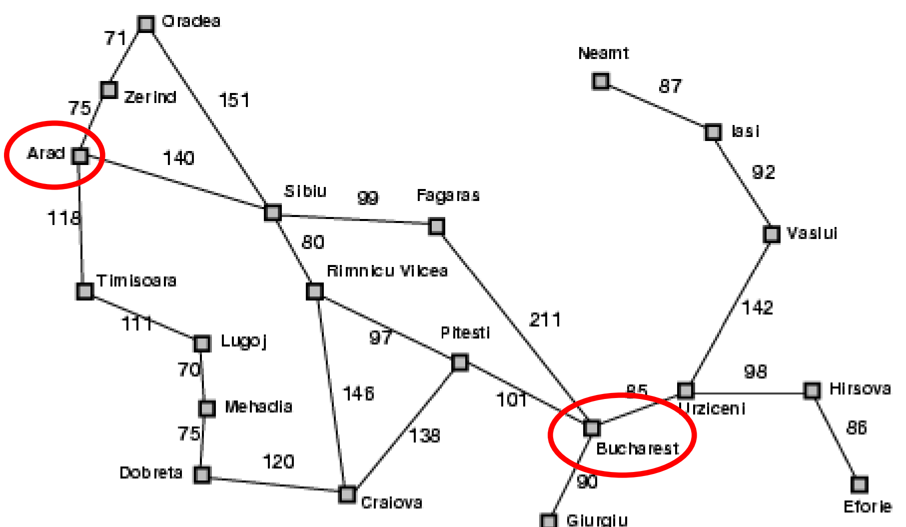

Straight–line distance to Bucharest

| | |
|---|---|
| **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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Greedy best-first search

- Expand the node that has the lowest value of the heuristic function $h(n)$

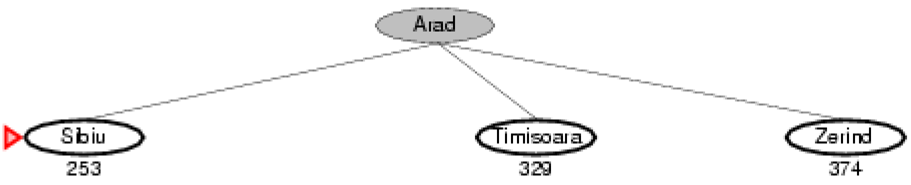
# Greedy best-first search example

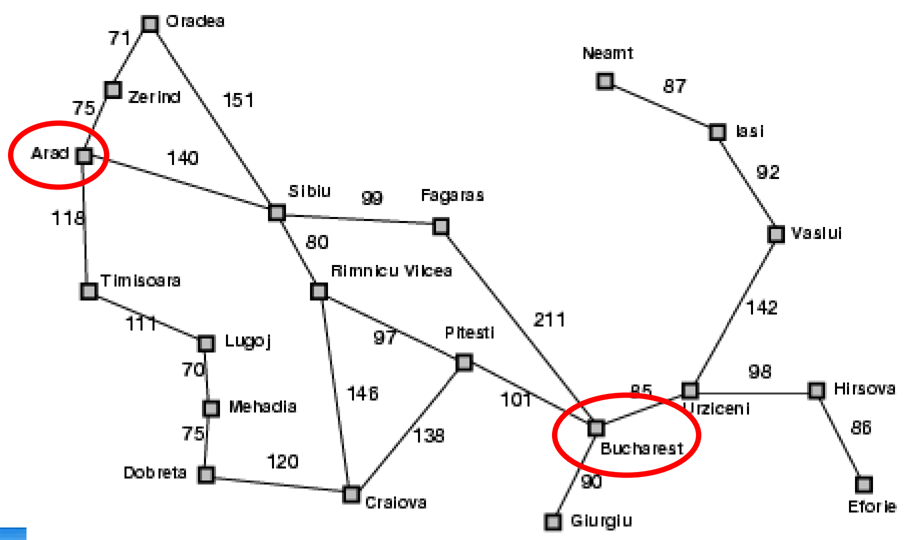




Straight–line distance to Bucharest

| | |
|---|---|
| **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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Greedy best-first search example

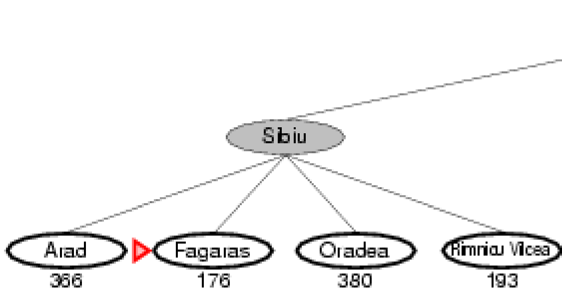


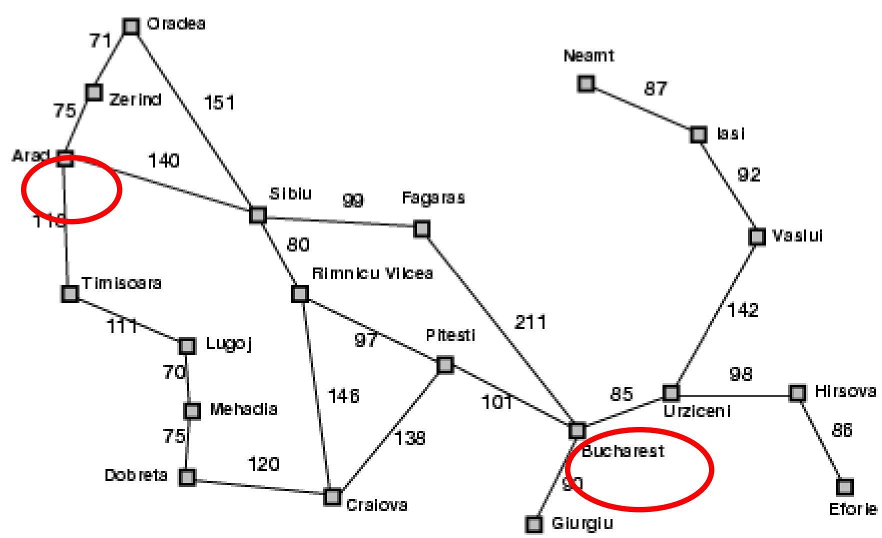




Straight–line distance to Bucharest

| | |
|---|---|
| **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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Greedy best-first search example

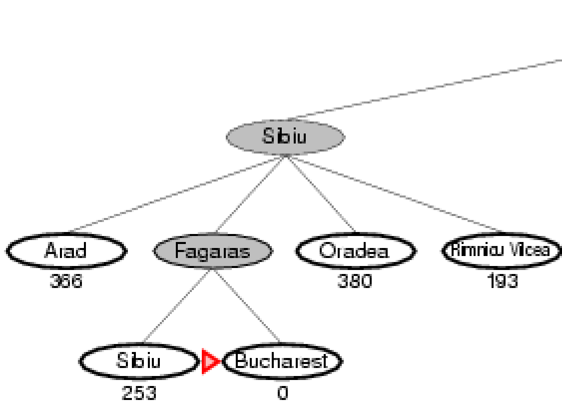


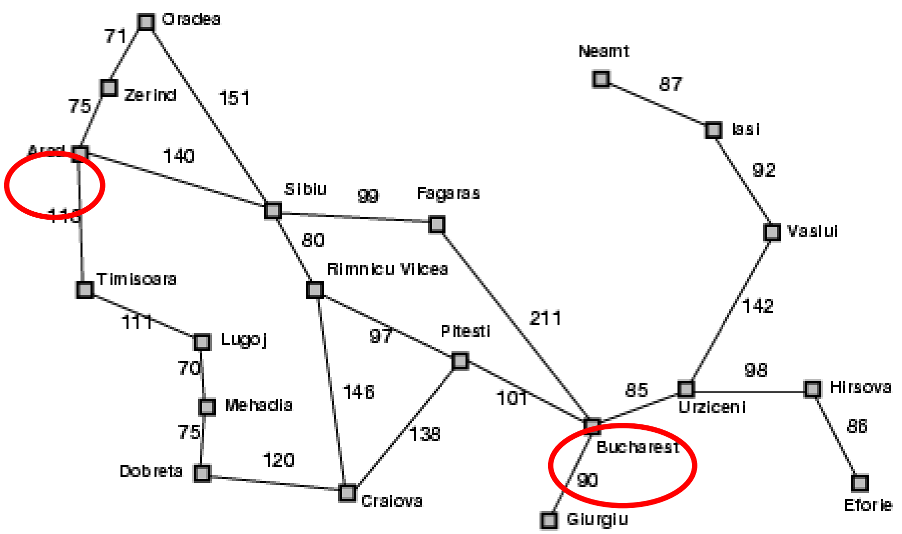




Straight–line distance to Bucharest

| | |
|---|---|
| **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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Greedy best-first search example



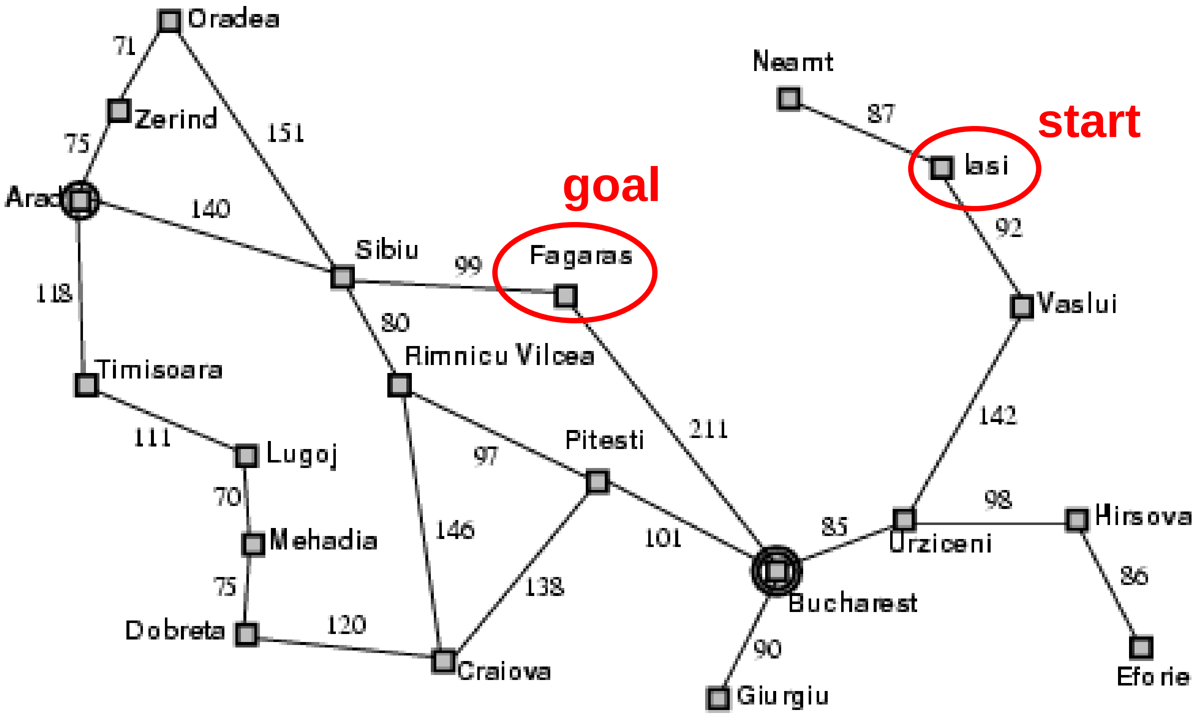

Straight–line distance to Bucharest

| | |
|---|---|
| **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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Properties of greedy best-first search

- **Complete?**

  No – can get stuck in loops



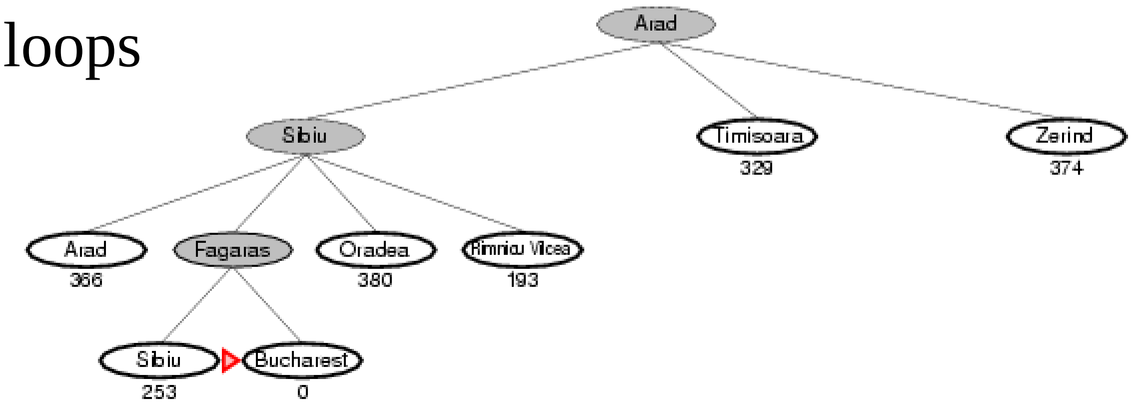
# Properties of greedy best-first search

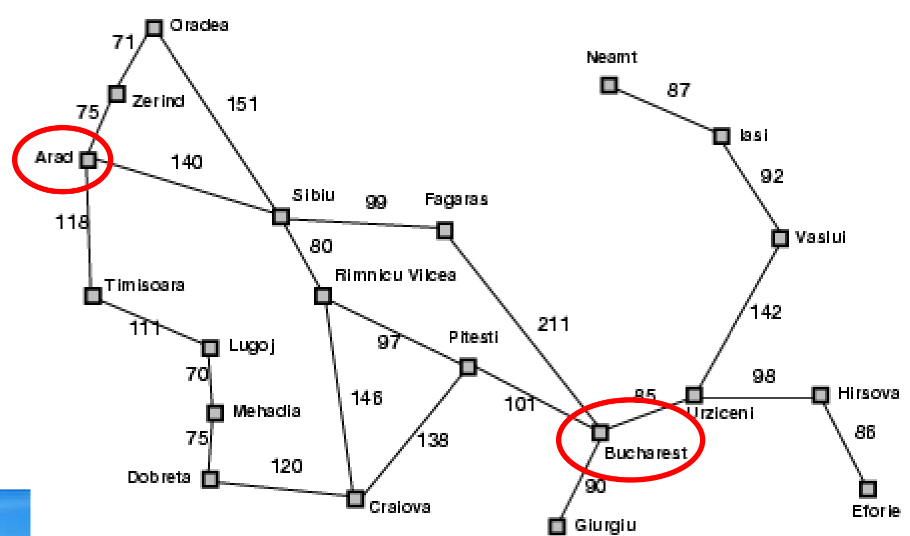
- **Complete?**

  No – can get stuck in loops

- **Optimal?**

  No




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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Properties of greedy best-first search

- **Complete?**

  No – can get stuck in loops

- **Optimal?**

  No

- **Time?**

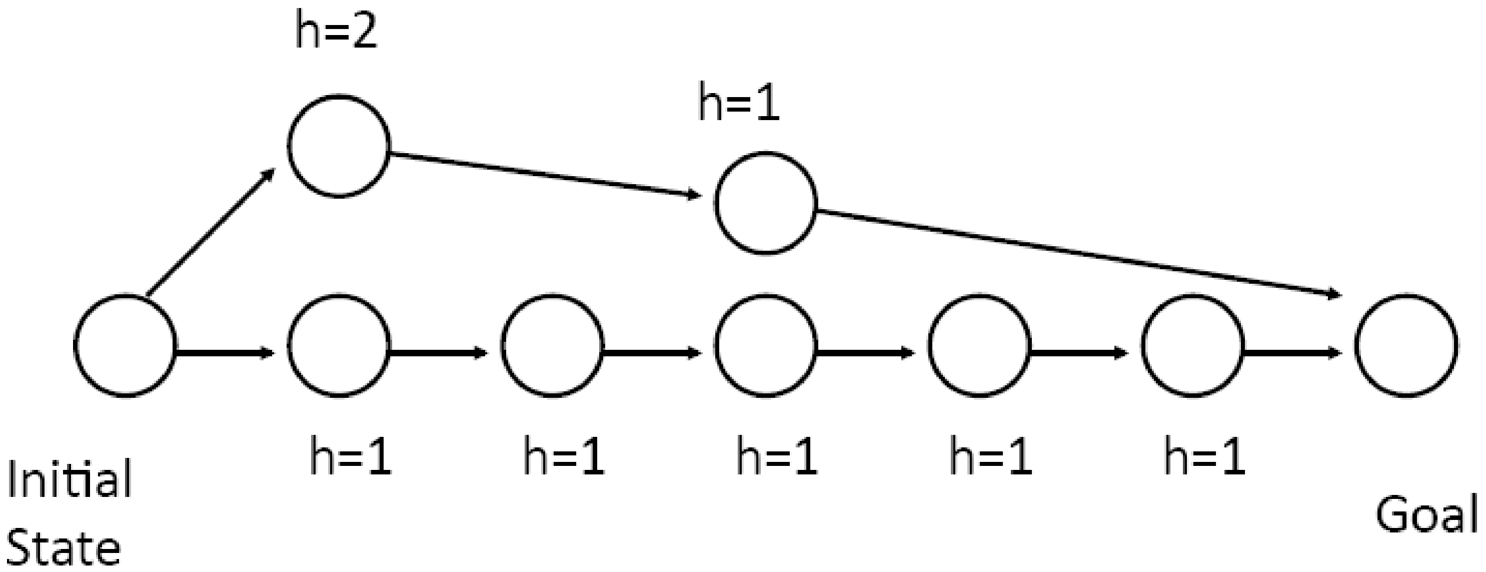
  Worst case: $O(b^m)$

  Best case: $O(bd)$ – If $h(n)$ is 100% accurate

- **Space?**

  Worst case: $O(b^m)$

# How can we fix the greedy problem?



# $A^*$ search

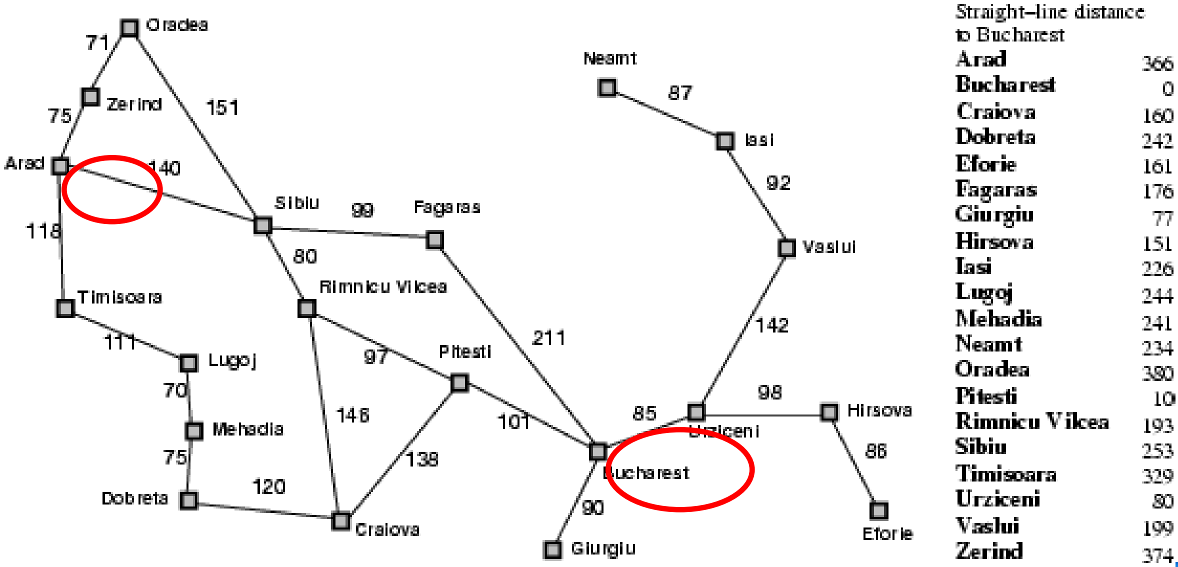
- Idea: avoid expanding paths that are already expensive
- The evaluation function $f(n)$ is the estimated total cost of the path through node $n$ to the goal:

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

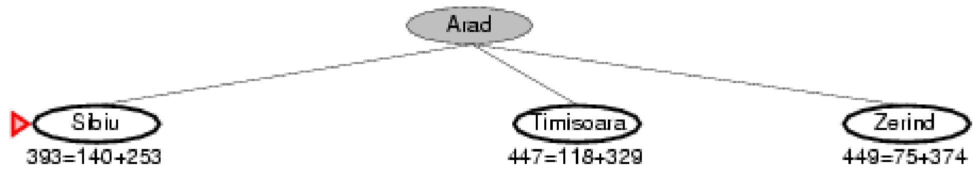
$g(n)$: cost so far to reach $n$ (path cost)

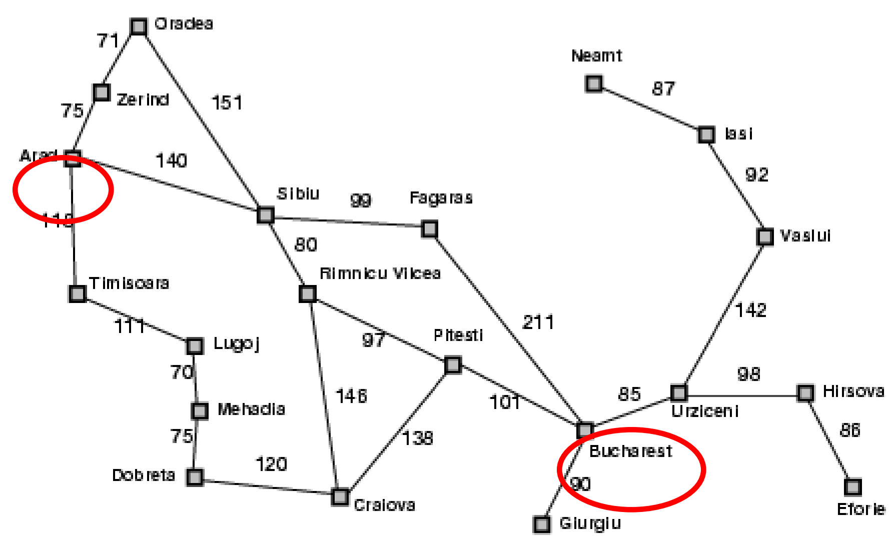
$h(n)$: estimated cost from $n$ to goal (heuristic)

# A* search example

# A* search example

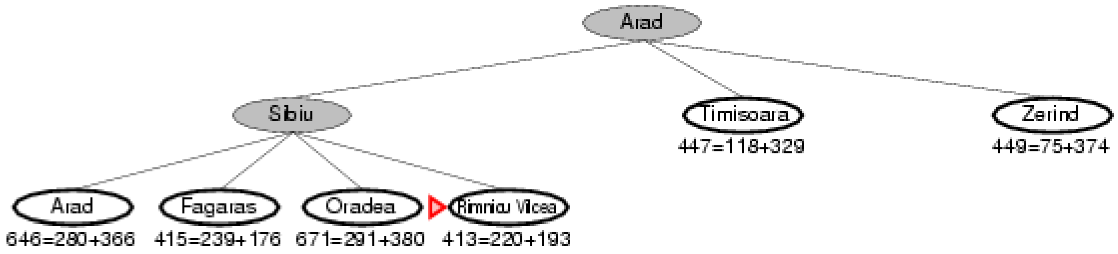


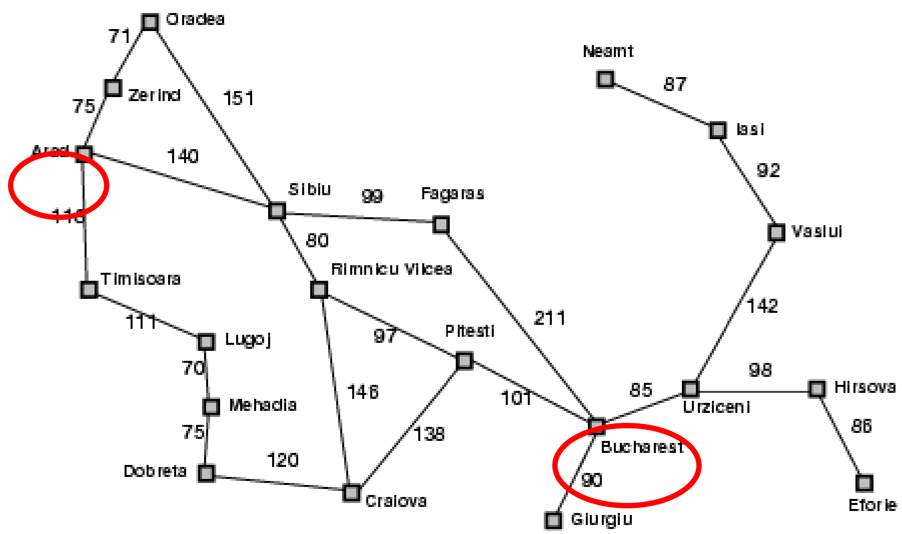




Straight–line distance to Bucharest

| | |
|---|---|
| **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** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# A* search example

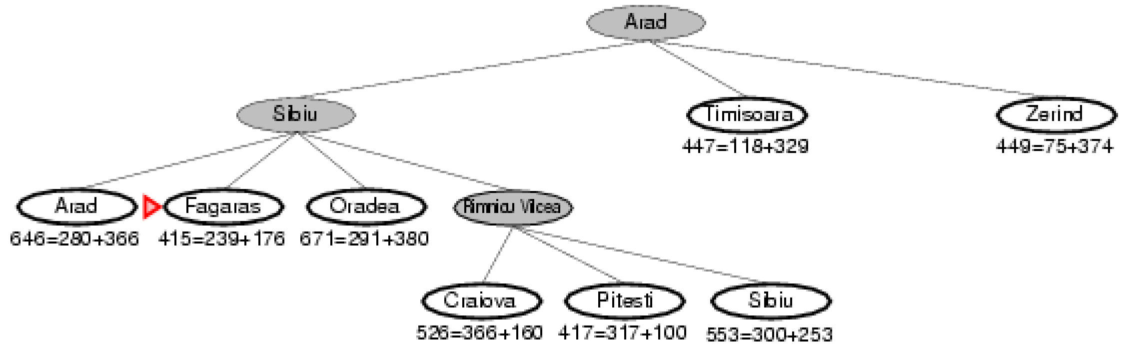


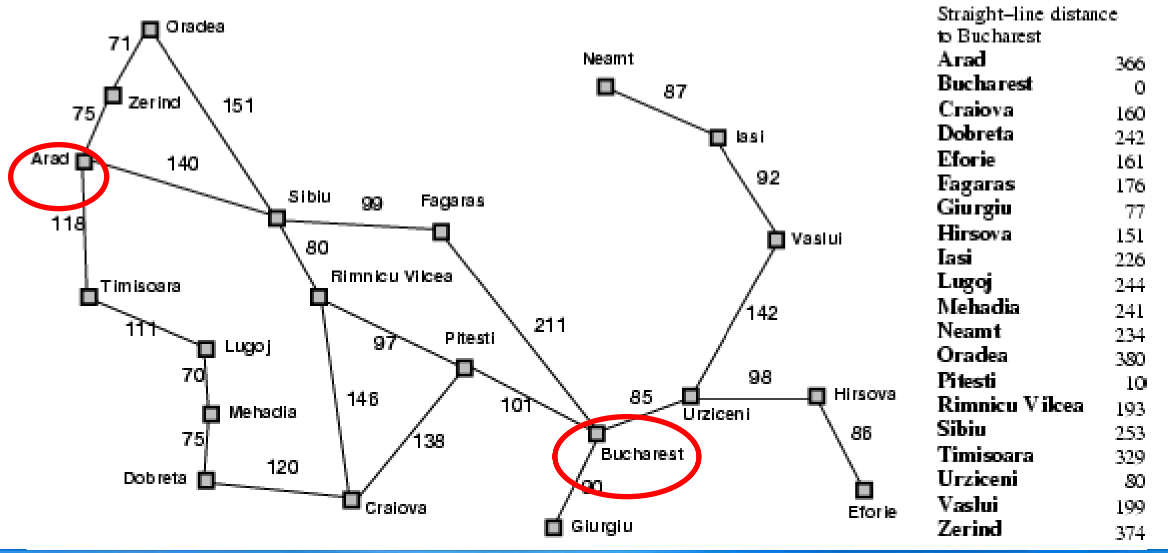




Straight–line distance to Bucharest

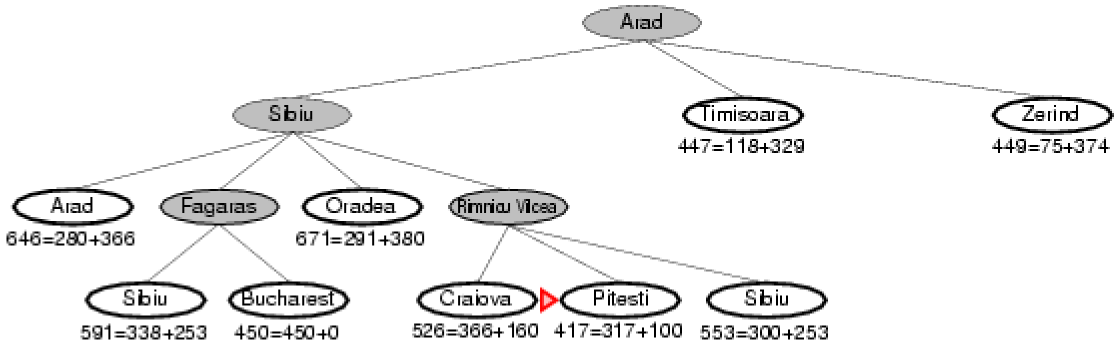
| | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

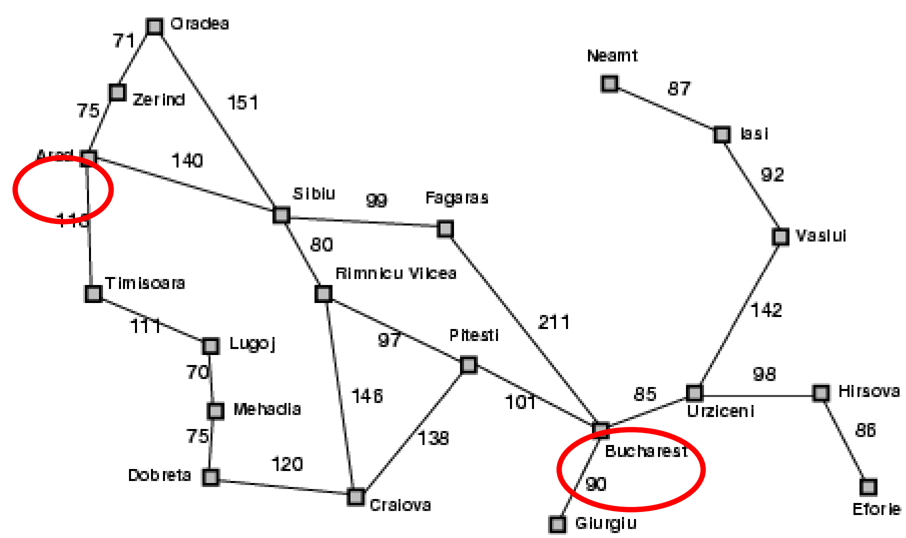
# A* search example



# A* search example

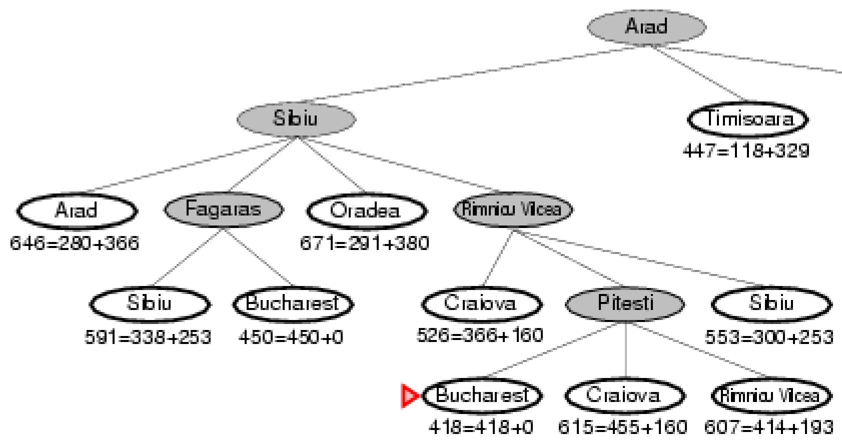


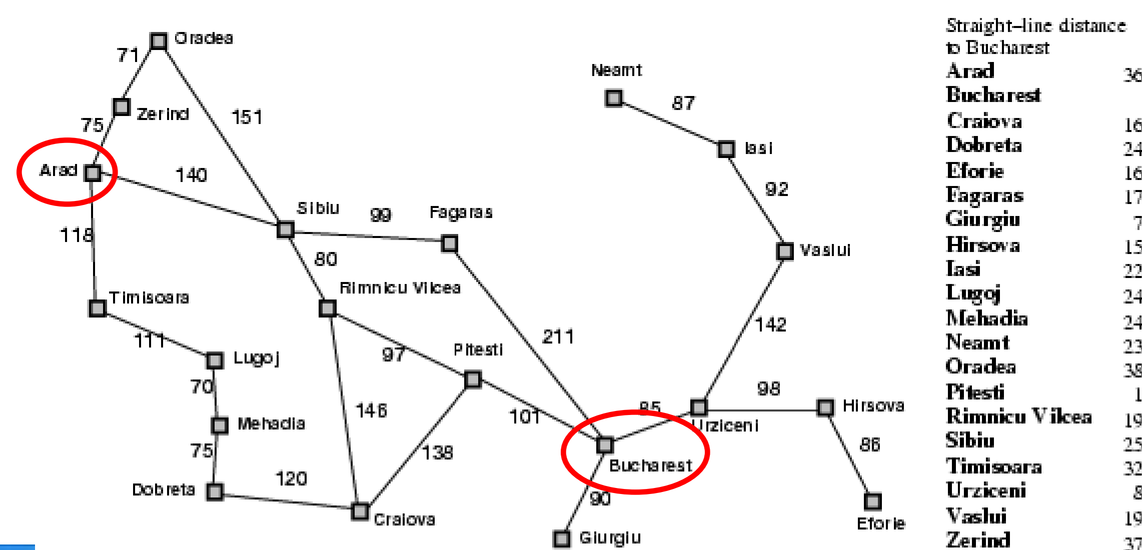




Straight–line distance to Bucharest

| | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# A* search example

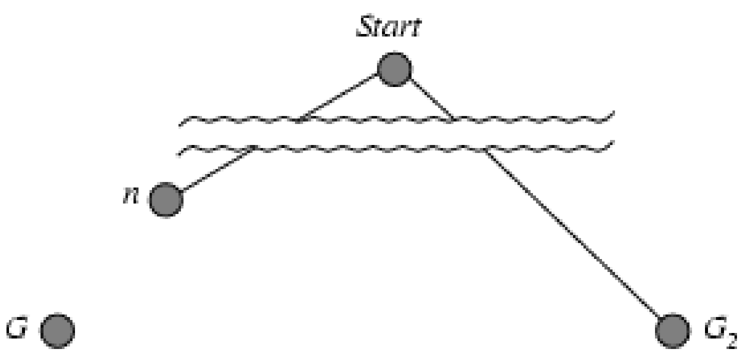


# Admissible heuristics

- 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 $n$
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: straight line distance never overestimates the actual road distance
- **Theorem:** If $h(n)$ is admissible, $A^*$ is optimal

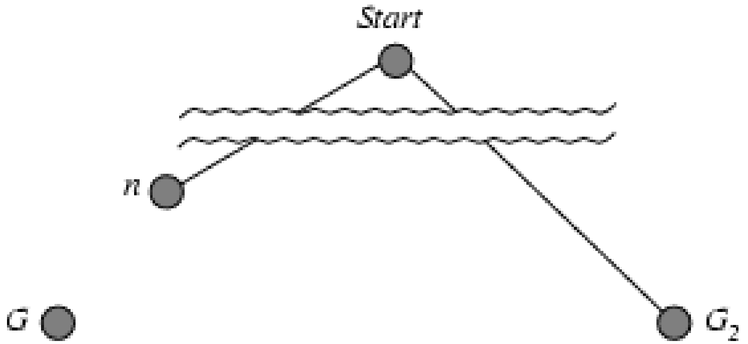
# Optimality of A* (proof)

- Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.
- 


- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since $G_2$ is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

# Optimality of A* (proof)

- Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.




- $f(G_2) > f(G)$ from above
- $h(n) \leq h^*(n)$ since h is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$

Hence $f(G_2) > f(n)$, and A* will never select $G_2$ for expansion

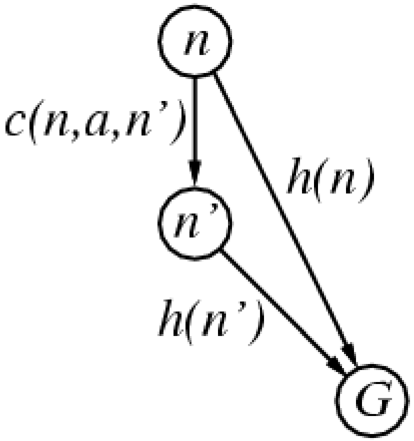
# Optimality of A*

- A* is optimally efficient – no other tree-based algorithm that uses the same heuristic can expand fewer nodes and still be guaranteed to find the optimal solution
  – Any algorithm that does not expand all nodes with $f(n) < C^*$ risks missing the optimal solution

# Consistent Heuristics

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

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




- If *h* is consistent, we have

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

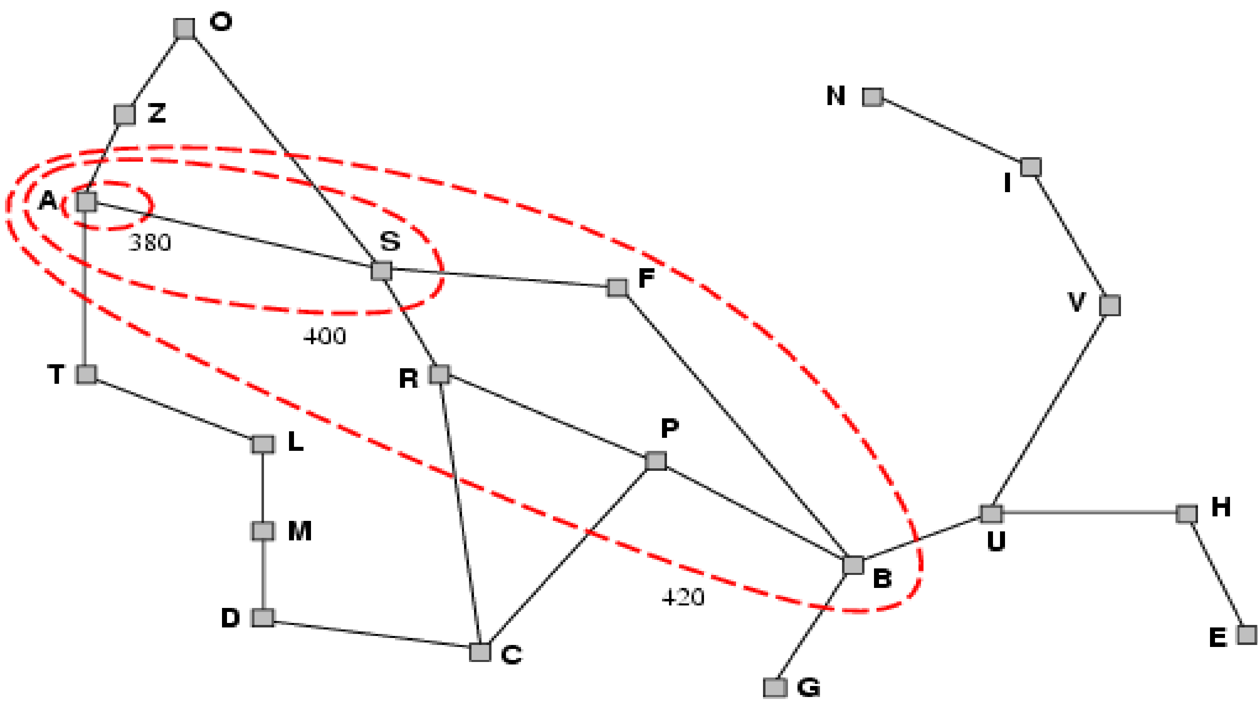
i.e., *f(n)* is non-decreasing along any path.

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

# 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 $f_i < f_{i+1}$



# Properties of A*

- **Complete?**

  Yes – unless there are infinitely many nodes with $f(n) \leq C^*$
- **Optimal?**

  Yes
- **Time?**

  Number of nodes for which $f(n) \leq C^*$ (exponential)
- **Space?**

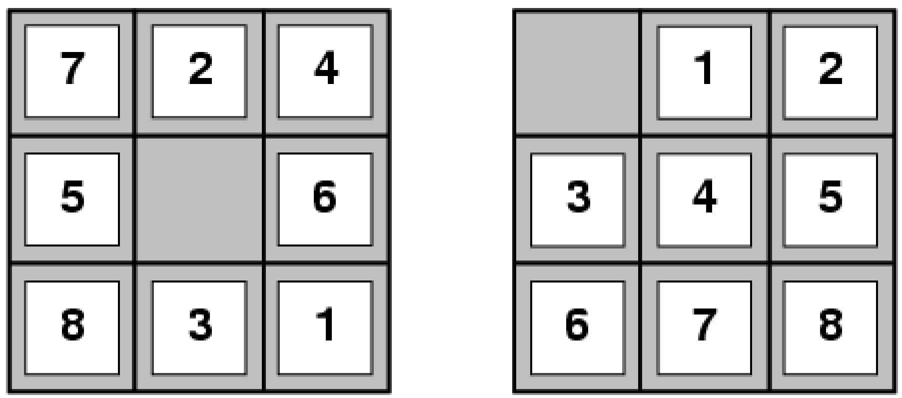
  Exponential

# Designing heuristic functions

- Heuristics for the 8-puzzle

  $h_1(n)$ = number of misplaced tiles

  $h_2(n)$ = total Manhattan distance (number of squares from desired location of each tile)




Start State Goal State

$$h_1(\text{start}) = 8$$

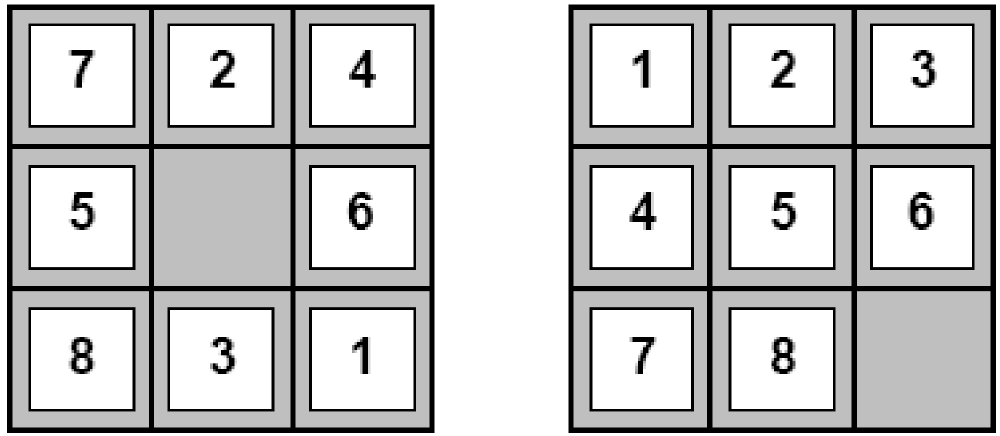
$$h_2(\text{start}) = 3+1+2+2+2+3+3+2 = 18$$

- Are $h_1$ and $h_2$ admissible?

# Designing heuristic functions

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



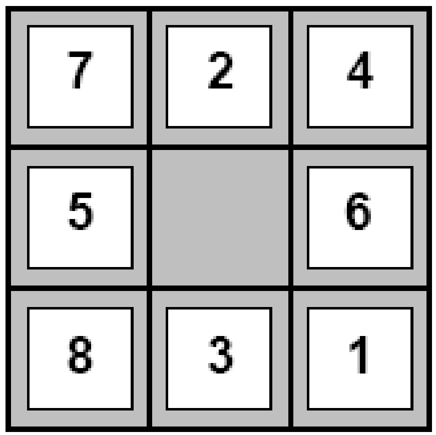

Start State Goal State

$\underline{h_1(S) =}??$
$\underline{h_2(S) =}??$

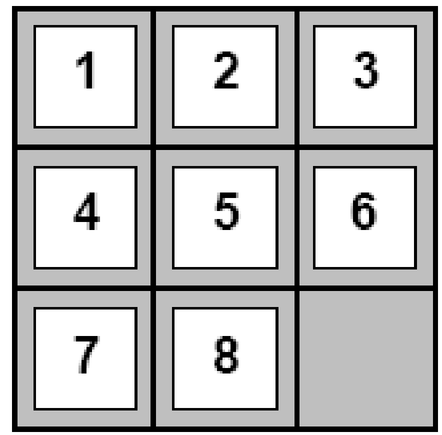
# Designing heuristic functions

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)




Start State



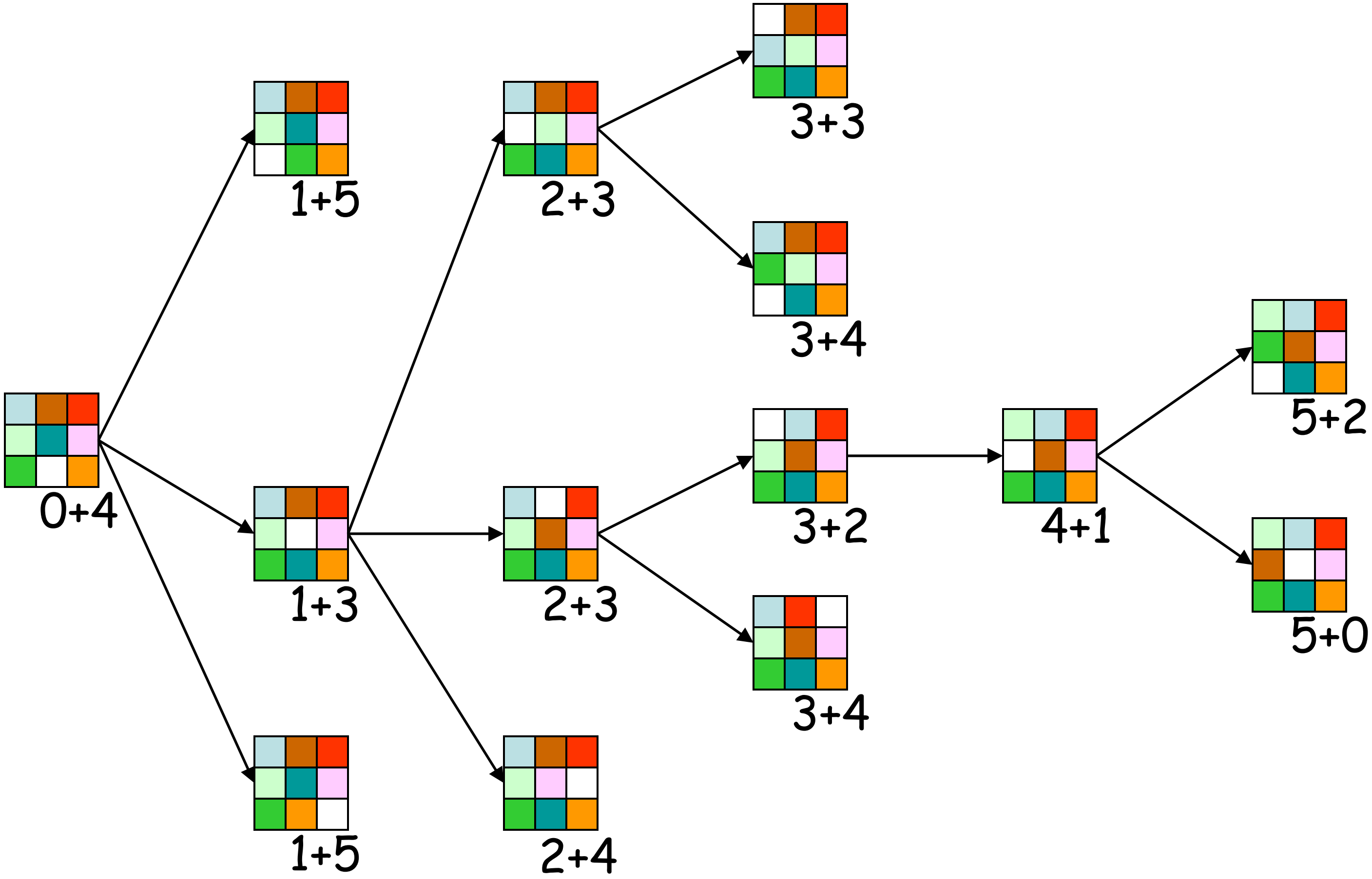

Goal State

$\underline{h_1(S) = ??}$ 6
$\underline{h_2(S) = ??}$ 4+0+3+3+1+0+2+1 = 14

# 8-Puzzle

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

$h(N)$ = Number of misplaced tiles



# Graph search

**function** GRAPH-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 leave it from the frontier

**if** the node contains a goal state **then return** the corresponding solution

**add the node to the explored set**

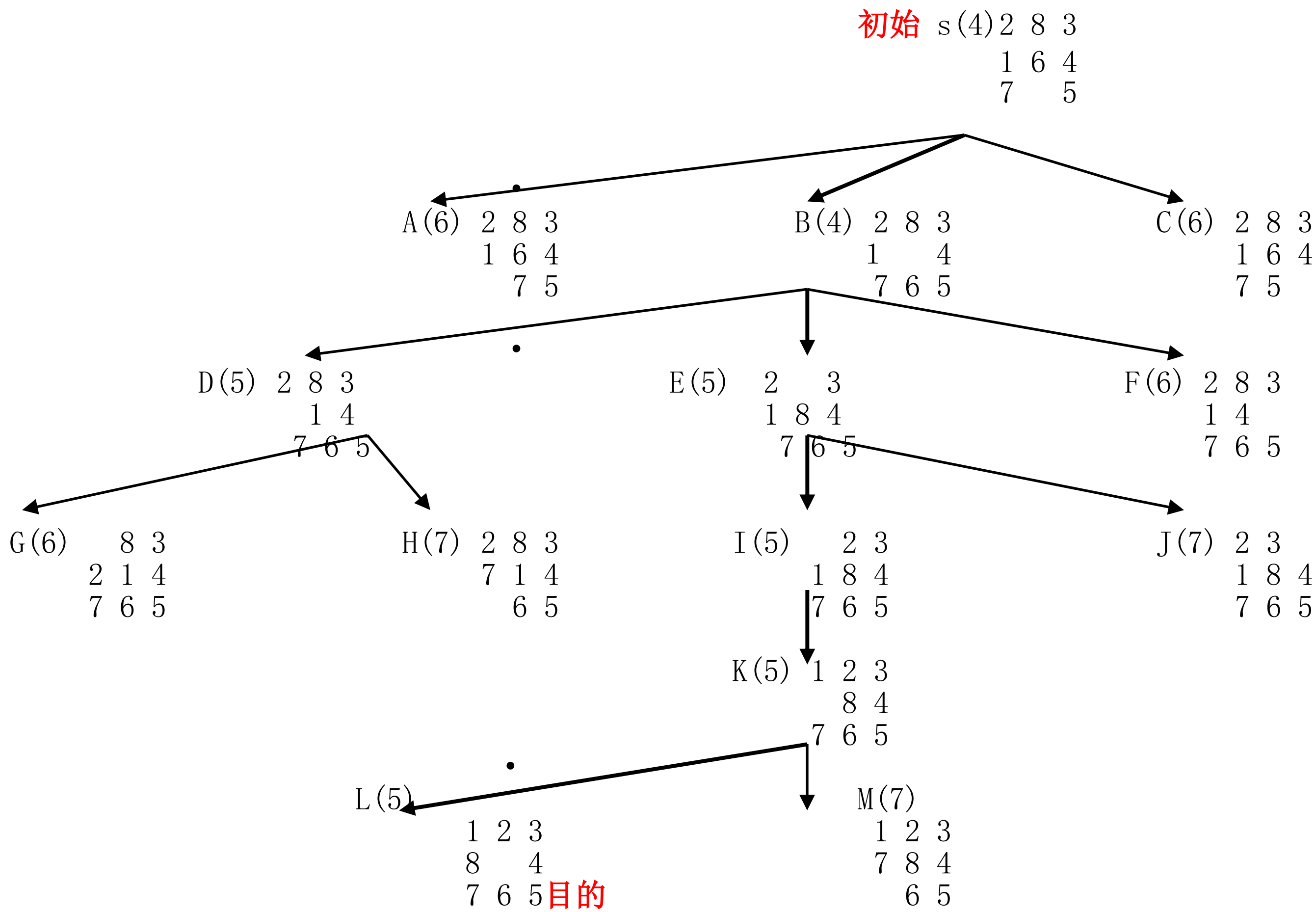
expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

# A搜索算法

```
procedure  heuristic_search
 Begin
  open：= [start]；closed：= [ ]；f(s)：= g(s)+h(s)；        *初始化
  While  open ≠ [ ]  do
    Begin
      从open表中删除第一个状态，称之为n；
      If  n = 目的状态 Then  Return（success）；
      生成n的所有子状态；
      If  n没有任何子状态  Then  Continue；
      For  n的每个子状态  Do
        Case 子状态 is not already on open表 or closed表：
          Begin 计算该子状态的估价函数值；将该子状态加到 open表中；End；
        Case 子状态 is already on open表：
          If  该子状态是沿着一条比在open 表已有的更短路径而到达
            Then  记录更短路径走向及其估价函数值；
         Case 子状态 is already on closed表：
          If  该子状态是沿着一条比在closed表已有的更短路径而到达 Then
                      Begin 将该子状态从closed表移到open表中；记录更
           短路径走向及 其估价函数值；End；  Case End；
      将n放入closed表中；根据估价函数值，从小到大重新排列open表；
    End；
  Return(failure)；                          *open表中结点已耗尽
 End.
```

# 八数码问题的搜索解树


初始 s(4) 2 8 3
1 6 4
7 5
A(6) 2 8 3
1 6 4
7 5
B(4) 2 8 3
1 4
7 6 5
C(6) 2 8 3
1 6 4
7 5
D(5) 2 8 3
1 4
7 6 5
E(5) 2 3
1 8 4
7 6 5
F(6) 2 8 3
1 4
7 6 5
G(6) 8 3
2 1 4
7 6 5
H(7) 2 8 3
7 1 4
6 5
I(5) 2 3
1 8 4
7 6 5
J(7) 2 3
1 8 4
7 6 5
K(5) 1 2 3
8 4
7 6 5
L(5) 1 2 3
8 4
7 6 5 目的
M(7) 1 2 3
7 8 4
6 5

# 八数码问题的搜索中open/cloesd表变化：

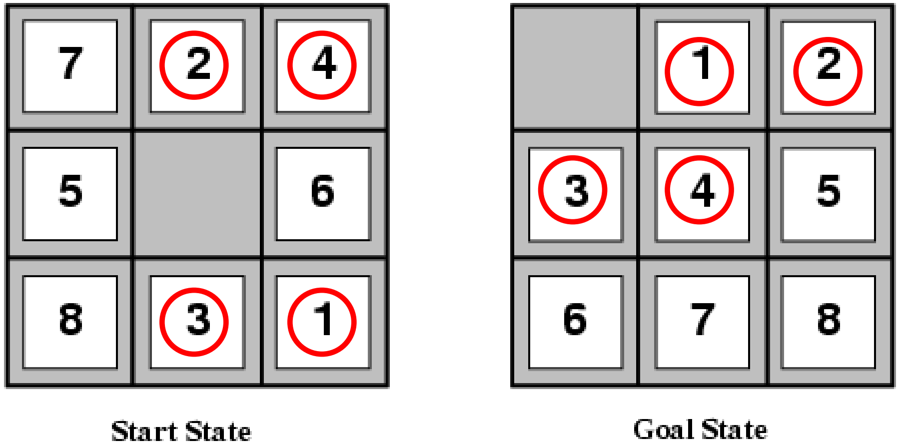
| Open表 | Closed表 |
|---|---|
| 初始化：(s(4)) | ( ) |
| 一次循环后： | |
| (B(4),A(6),C(6)) | (s(4)) |
| 二次循环后： | |
| (D(5),E(5),A(6),C(6),F(6)) | (s(4) B(4)) |
| 三次循环后： | |
| (E(5),A(6),C(6),F(6),G(6),H(7)) | (s(4) B(4) D(5)) |
| 四次循环后： | |
| (I(5),A(6),C(6),F(6),G(6),H(7),J(7)) | (s(4) B(4) D(5) E(5)) |
| 五次循环后： | |
| (K(5),A(6),C(6),F(6),G(6),H(7),J(7)) | (s(4) B(4) D(5) E(5) I(5)) |
| 六次循环后： | |
| (L(5),A(6),C(6),F(6),G(6),H(7),J(7),M(7)) | (s(4) B(4) D(5) E(5) I(5) K(5)) |
| 七次循环后： | |
| L为目的状态,则成功推出，结束搜索 | (s(4) B(4) D(5) E(5) I(5) K(5) L(5)) |

# 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

# Heuristics from subproblems

- Let $h_3(n)$ be the cost of getting a subset of tiles (say, 1,2,3,4) into their correct positions
- Can precompute and save the exact solution cost for every possible subproblem instance – *pattern database*



# Dominance

- If $h_1$ and $h_2$ are both admissible heuristics and $h_2(n) \geq h_1(n)$ for all $n$, (both admissible) then $h_2$ dominates $h_1$
- Which one is better for search?
  - A* search expands every node with $f(n) < C^*$ or $h(n) < C^* - g(n)$
  - Therefore, A* search with $h_1$ will expand more nodes

# Dominance

- Typical search costs for the 8-puzzle (average number of nodes expanded for different solution depths):

- $d$=12 IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes

- $d$=24 IDS ≈ 54,000,000,000 nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

# Combining heuristics

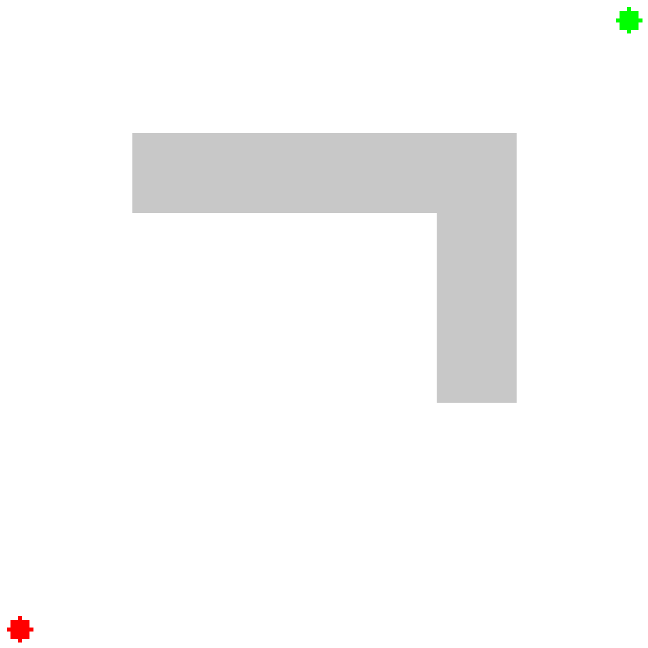
- Suppose we have a collection of admissible heuristics $h_1(n), h_2(n), \ldots, h_m(n)$, but none of them dominates the others
- How can we combine them?

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

# Weighted A* search

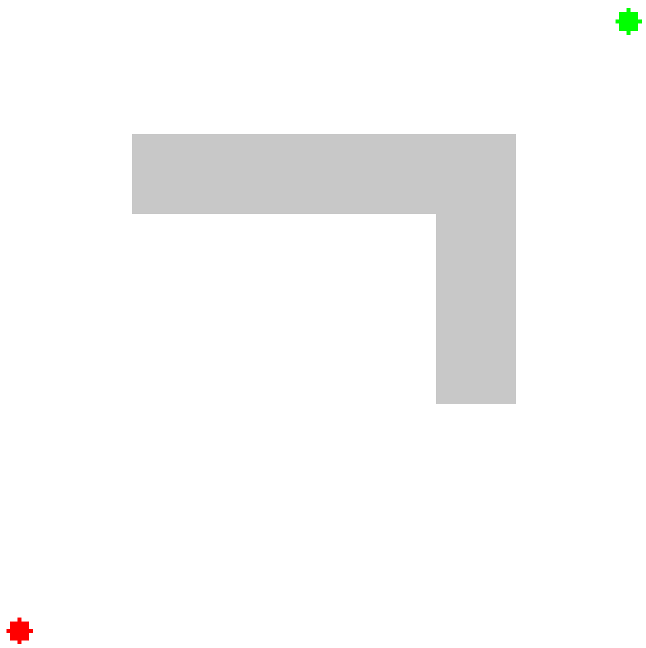
- Idea: speed up search at the expense of optimality
- Take an admissible heuristic, “inflate” it by a multiple $\alpha > 1$, and then perform A* search as usual
- Fewer nodes tend to get expanded, but the resulting solution may be suboptimal (its cost will be at most $\alpha$ times the cost of the optimal solution)

# Example of weighted A* search



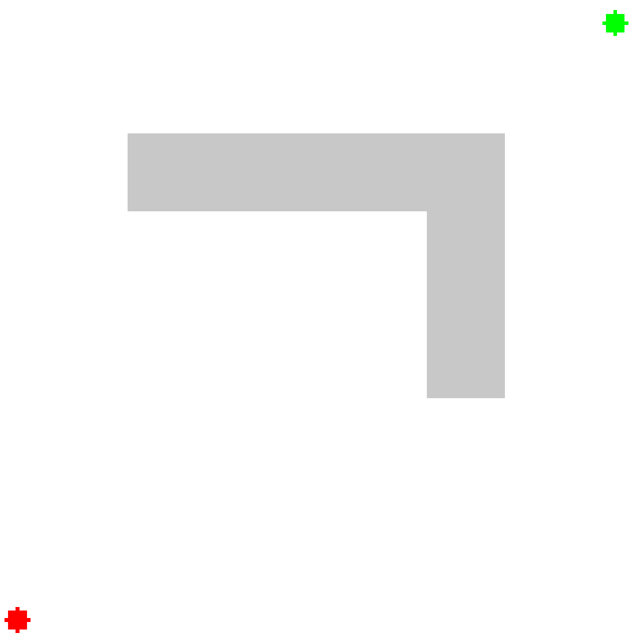
Heuristic: 5 * Euclidean distance from goal
Source: Wikipedia

# Example of weighted A* search



Heuristic: 5 * Euclidean distance from goal

Source: [Wikipedia](Wikipedia)



Compare: Exact A*

# Memory-bounded search

- The memory usage of A* can still be exorbitant
- How to make A* more memory-efficient while maintaining completeness and optimality?

- Iterative deepening A* search
- Recursive best-first search, SMA*
  - Forget some subtrees but remember the best f-value in these subtrees and regenerate them later if necessary

- Problems: memory-bounded strategies can be complicated to implement, suffer from “thrashing”

# All search strategies

| Algorithm | Complete? | Optimal? | Time complexity | Space complexity |
|---|---|---|---|---|
| **BFS** | Yes | If all step costs are equal | $O(b^d)$ | $O(b^d)$ |
| **UCS** | Yes | Yes | Number of nodes with $g(n) \leq C^*$ | |
| **DFS** | No | No | $O(b^m)$ | $O(bm)$ |
| **IDS** | Yes | If all step costs are equal | $O(b^d)$ | $O(bd)$ |
| **Greedy** | No | No | Worst case: $O(b^m)$<br>Best case: $O(bd)$ | |
| **A*** | Yes | Yes | Number of nodes with $g(n)+h(n) \leq C^*$ | |

# Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies

Iterative deepening search uses only linear space
and not much more time than other uninformed algorithms

Graph search can be exponentially more efficient than tree search

# Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest $h$
- incomplete and not always optimal

A$^*$ search expands lowest $g + h$
- complete and optimal
- also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

# 作业

1) 3.6
2) 3.9
3) 3.21
4) 3.25