Artificial Intelligence
A Modern Approach
Third Edition
Stuart Russell & Peter Norvig

Chapter 3
Solving Problems by Searching

3.1 Problem-Solving Agents

- Formulate a real-world problem to a **search** problem
- A **simple problem-solving** agent
  - **Formulate** a **goal** and a **problem**
  - **Search** for a **sequence of actions** that solves the problem
  - **Execute** the **actions** one by one

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

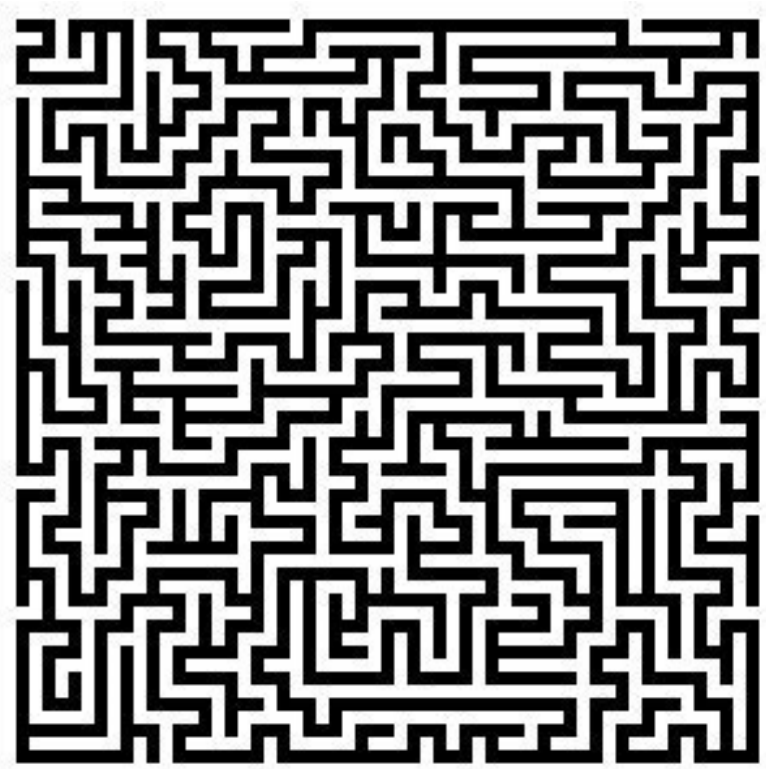
from "current state" to the "goal state"

action sequence = $\{a_1, a_2, a_3, \ldots a_n\}$

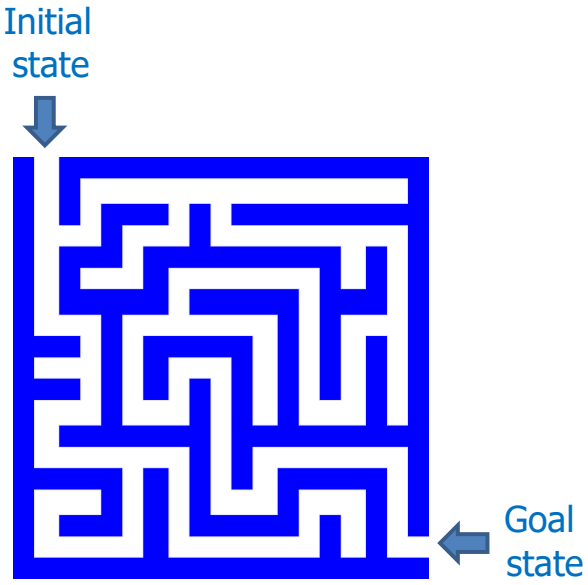
action sequence = $\{a_2, a_3, \ldots a_n\}$

Search

- Solving problems by **searching**

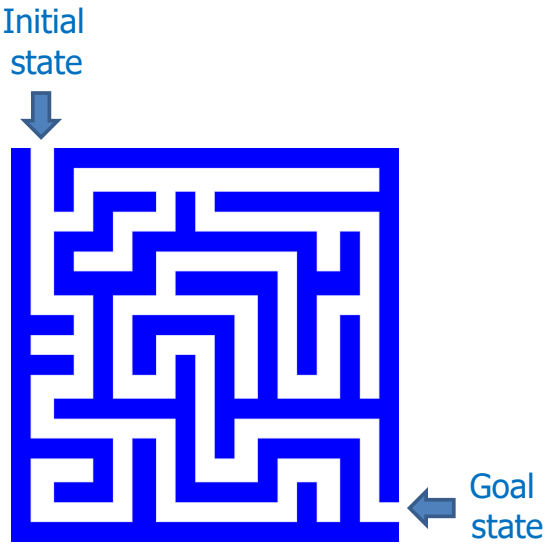


- **Search**
  - Consider the problem of designing **goal-based agents** in **observable**, **deterministic**, **discrete** environments
  - **Solution** : a **fixed sequence of actions**
  - **Search** : the process of looking for the **sequence of actions** that reaches the **goal**



- Search problem components
  - **States**
  - **Initial state**
  - **Actions**
  - **Transition model** $f: S \times A \rightarrow S$
    - What is the **result** of performing a given action in a given state?
  - **Goal state**
  - **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

May be one goal or multiple goals
— Find an optimal path to reach the one goal
— Find an optimal goal of the multiple goals

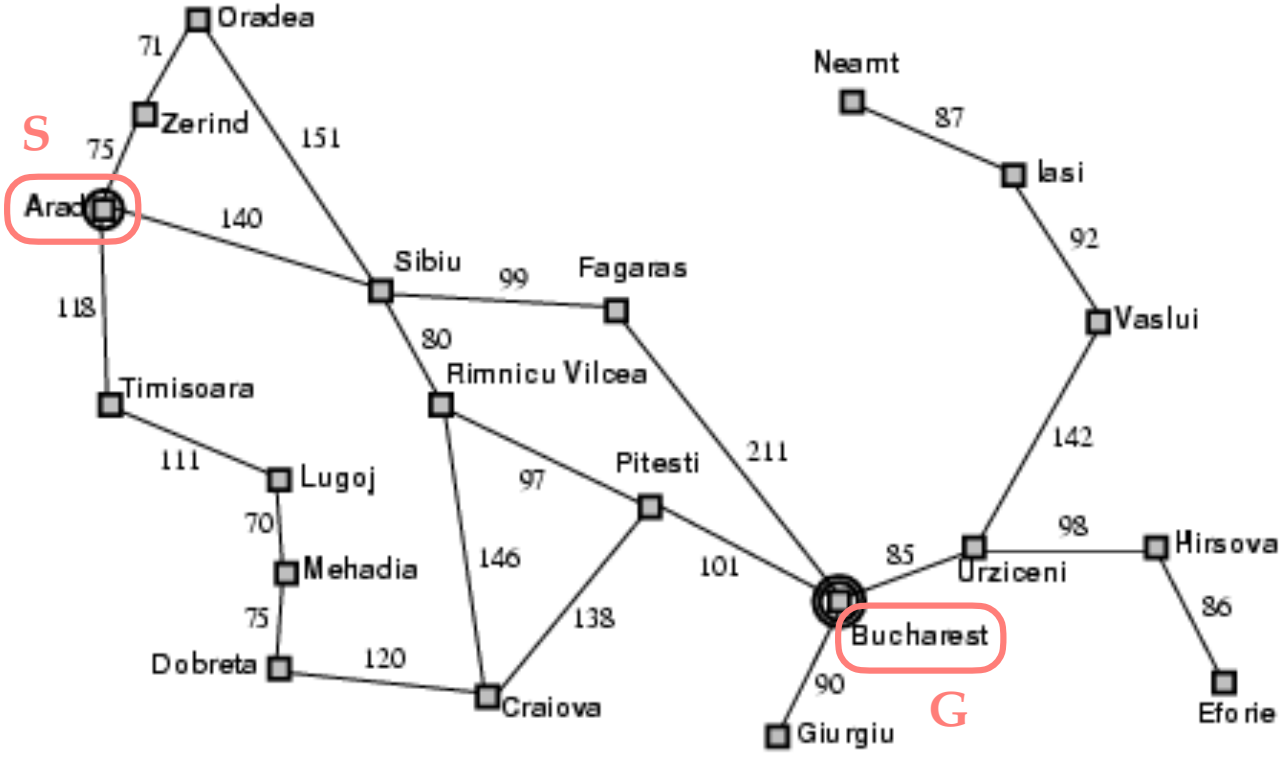


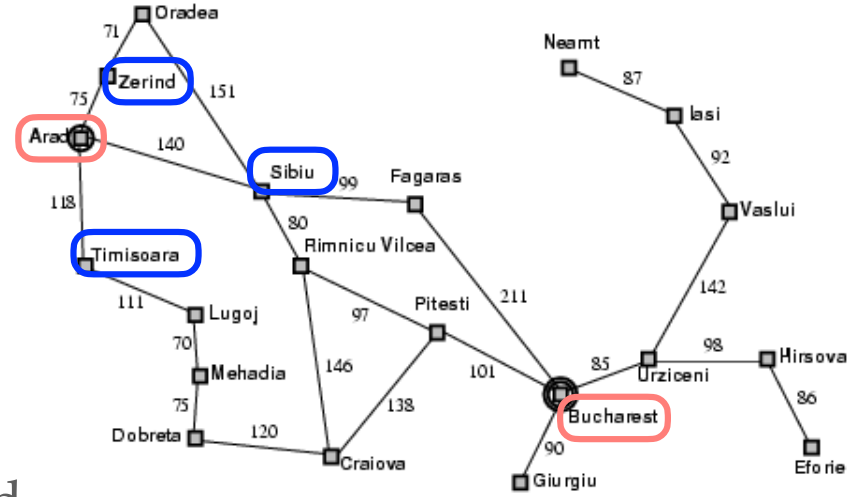
3.2 Example Problems

- Example: Romania
- Example: Vacuum world
- Example: 8-puzzle
- Example: 8-Queen Puzzle
- Real-world problems

Example: Romania

- On vacation in Romania
  - Currently in Arad
  - Flight leaves tomorrow from Bucharest





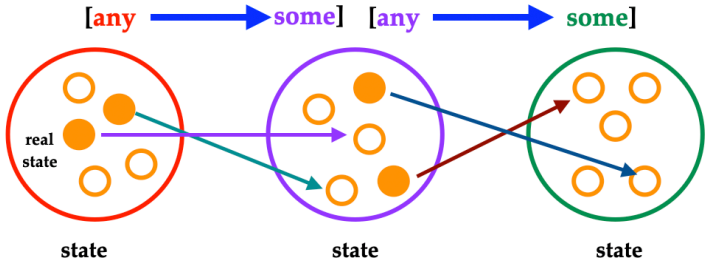
- **Formulate goal**
  - Be in Bucharest
- **Formulate problem**
  - **States**: various cities with initial state: Arad
  - ***Actions* (*s*)**: **set of actions** available in state ***s*** [go from one city to another]
- **Transition model**

  $f: S \times A \rightarrow S$
  - ***Result*(*s*,*a*)** = **state** that results from action ***a*** in state ***s***
  - Alternatively: **successor** function: $S(s)$ = set of action–state pairs
    e.g., $S(Arad) = \{<Arad \rightarrow Zerind, Sibiu, Timisoara>, \ldots\}$
  - [If you go from city A to city B, you end up in city B]
- **Goal state** (or **Goal test**: set of states)
  - e.g., *s* = "at Bucharest"

step cost: c_1, c_2, c_3
path cost: $c_1 + c_2 + c_3$

- **Path cost** (additive)
  - e.g., sum of **distances** (edge costs), number of actions executed, etc.
  - ***c*(*x*,*a*,*y*)** is the **step cost,** assumed to be ≥ 0, step cost is summed to yield path cost
- **Find solution**
  - ***solution*** = sequence of actions leading from initial state to a goal state
  - e.g. sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
- **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**

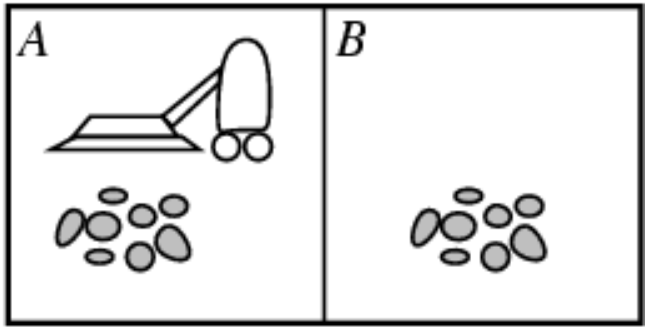
- **Select a state space**
  - Real world is absurdly complex
    - State space must be **abstracted** for problem solving
  - (**Abstract**) **state** ← set of **real** states
  - (**Abstract**) **action** ← complex combination of **real** actions
    - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - For guaranteed **realizability**, **any** real state "in Arad" must get to **some** real state "in Zerind"
  - (**Abstract**) **solution** ← set of **real** paths that are solutions in the real world
  - Each abstract action should be "easier" than the original problem



- **Environments types**
  - **Static / dynamic**
    - Static: no attention to changes in environment
  - **Observable / partially observable / unobservable**
    - Observable: it knows initial state
  - **Deterministic / stochastic**
    - Deterministic: no new percepts were necessary, we can predict the future perfectly given our actions
  - **Discrete / continuous**
    - Discrete: we can enumerate all possibilities

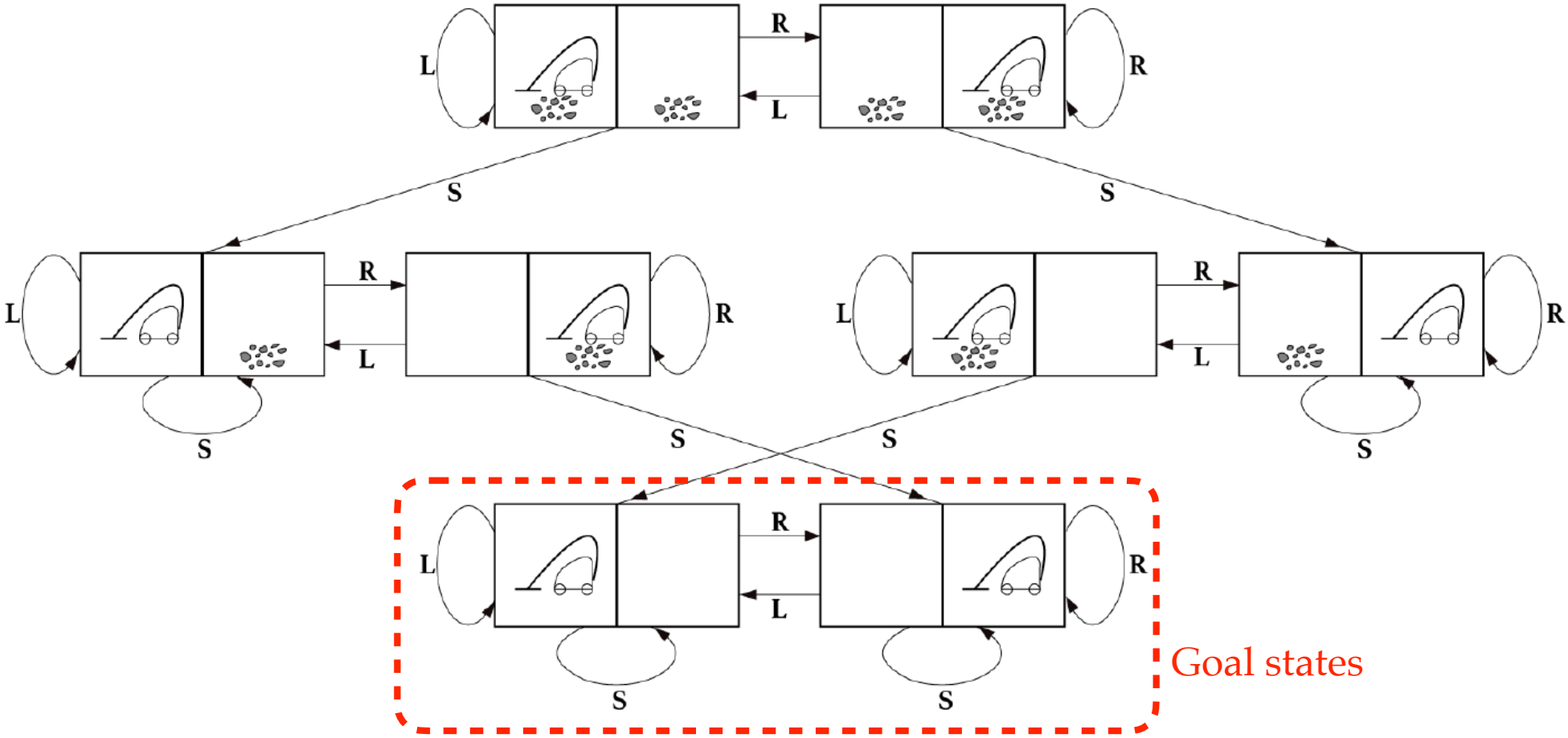
Example: Vacuum World

- **States**: dirt and robot location
- **Initial state**: any one of the above states. (ignore dirt amounts etc.)
- **Actions**: Left, Right, Suck, NoOp
- **Transition model**: the next figure
- **Goal test**: no dirt at all locations
- **Path cost**: 1 per action (0 for NoOp)

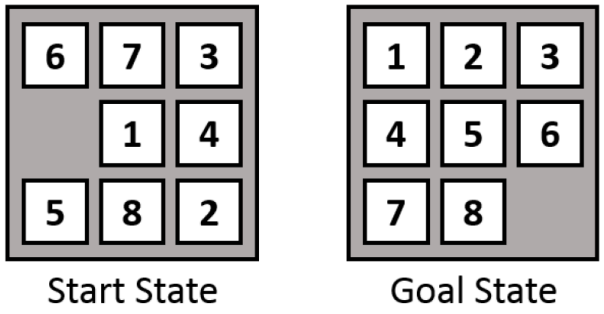


Vacuum World State Space Graph

L: Left
R: Right
S: Suck

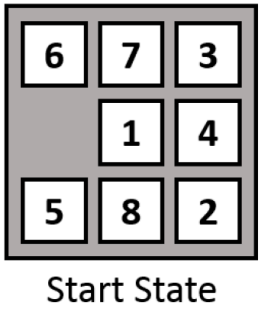


Example: 8-puzzle

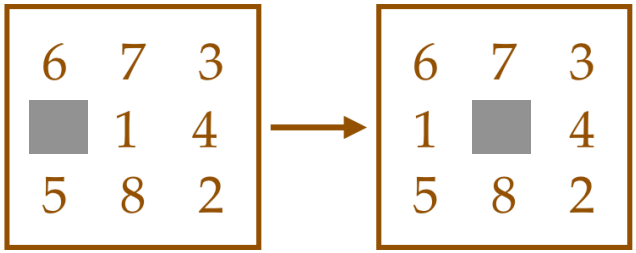


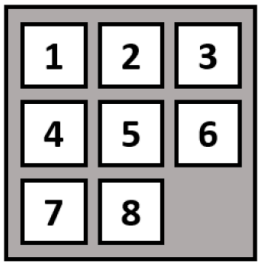
- **Initial state**: the left figure (given)
- **States** (locations of tiles)
  - 8-puzzle (3 × 3 board): 362,880 states (= 9!)
  - 15-puzzle (4 × 4 board): 2.6 trillion states (= 16!)
    - solved optimally in a few milliseconds
  - 24-puzzle (5 × 5 board): 10^{25} trillion states (= 25!)
    - solved optimally in several hours or less time

- **Actions**: move blank left, right, up, down (instead of 1 up, 6 down, …)
- **Transition model**: common sense
- **Goal test**: the right figure (given)
- **Path cost**: 1 per move
- Note: optimal solution of sliding-block *n*-puzzle is ***NP*-hard**



Start State



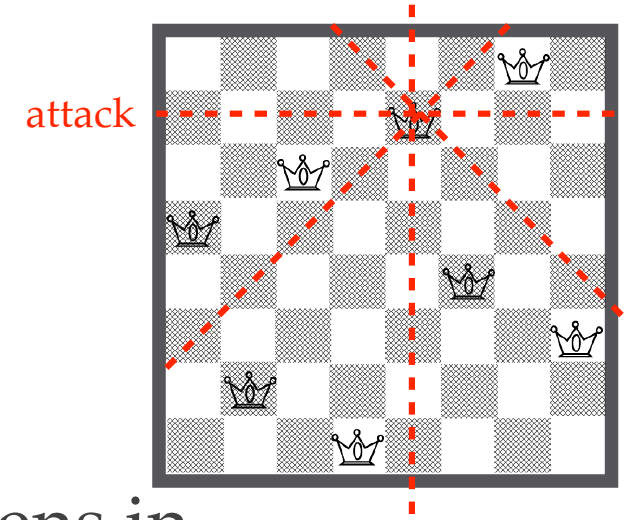


Goal State

NP-hardness (non-deterministic polynomial-time hardness): In computational complexity theory, NP-Hard is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP".

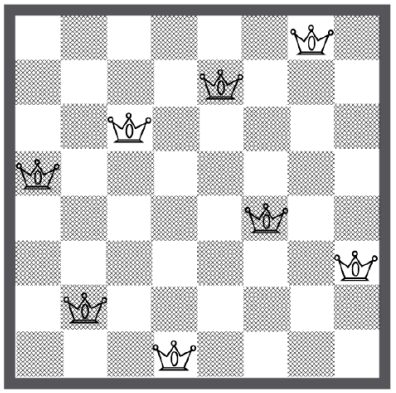
Example: 8-Queen Puzzle

- **States**:
  - [Type1] Any arrangement of $n \leqq 8$ queens, or
  - [Type2] Arrangements of $n \leqq 8$ queens in leftmost n columns, 1 per column, such that no queen attacks any other (BETTER!!)

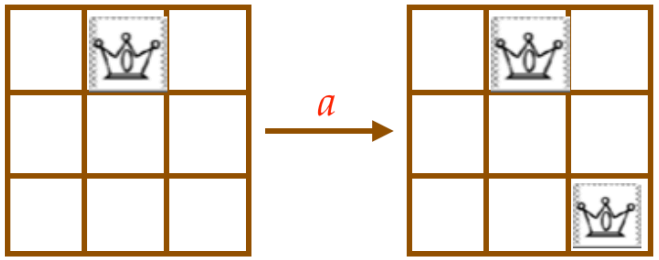


- **Initial state**: no queen on the board

- **Actions**:
  - [Type1] Add queen to **any empty square**
  - [Type2] Add queen to **each column** from let to right (BETTER!)
  - [Type3] Add queen to **leftmost empty square**, such that it is not attacked by other queens (BEST!!).

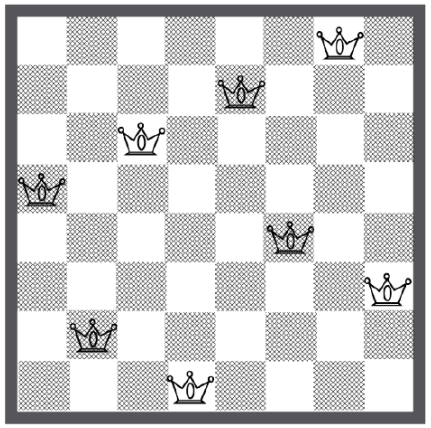


- **Transition model**:
  - Returns the board with a queen added to the specified square

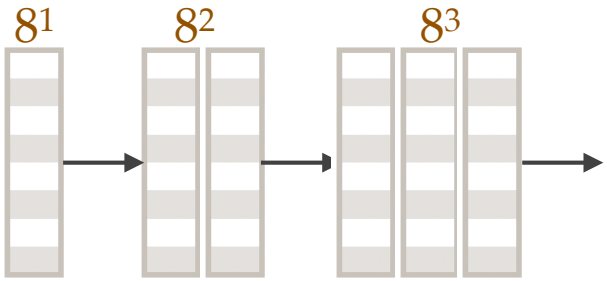


- **Goal test**: 8 queens are on the board, none attacked
- **Path cost**: number of trials (1 per move)

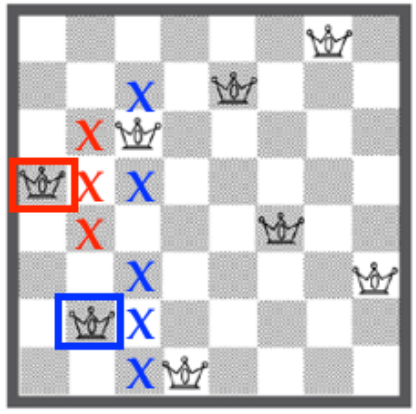
- [Type1] **States**: any 0~8 queens on the board
  - State space: $C_0^{64} + C_1^{64} + C_2^{64} + \cdots + C_8^{64} \simeq 5.1 \times 10^9$ (0 queen .... 8 queens on the board)
  - Solution space: $64 \cdot 63 \cdots 57 \simeq 1.8 \times 10^{14}$



- [Type2] **States**: one queen per column (left → right)
  - State space: $8^0 + 8^1 + 8^2 + \cdots + 8^8 \simeq 1.9 \times 10^7$ (0 queen .... 8 queens on the left n columns)
  - Solution space: $8^8 \simeq 1.6 \times 10^7$



- [Type3] **States**: all possible arrangements of n ($0 \leqq n \leqq 8$) queens at leftmost n columns with no queens attacked
  - **Actions**: Add a queen to the next column with no queens attacked, or backtrack if attacked
  - State space: 2057

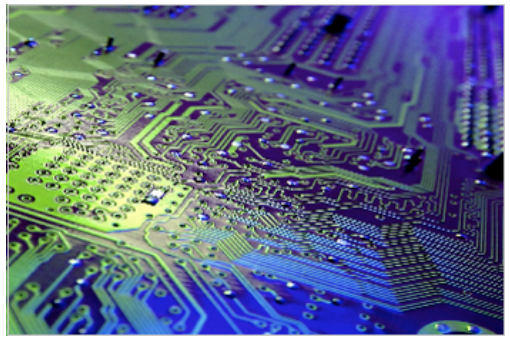


8 x (8-3) x (8-5) x .....

Real-World Problems

- **Rout finding**
  - Routing video streams in **computer networks**
  - **Military** operations planning
  - **Airline** travel-planning systems
- **Touring problems**
  - Closely related to route-finding problems, but with an important difference, e.g. **visit** every city **at least once**, starting and ending in the same location.

- **VLSI layout**
  - Position millions of components (transistors) and connections on a chip to
    - **Minimize** area
    - **Minimize** circuit delays
    - **Minimize** stray capacitances
    - **Maximize** manufacturing yield



- **Robot navigation**
  - Rather than following a discrete set of routes, a robot can move in a **continuous space** with an infinite set of possible actions and states
- **Automatic assembly sequencing**
  - Find an **order** in which to assemble the parts of some object

3.3 Searching for Solutions

- Search can be classified into **uninformed** search and **informed** search
  - **Uninformed search**
    - Doesn't predict **what might happen** if I do action A and action B?
    - The only information is **how much does it cost** if I do action A, and how much does it cost if I do action B?
  - **Informed search** or **heuristic search**
    - Use **problem-specific knowledge** beyond the definition of the problem itself — can find solutions more efficiently
    - A **heuristic** is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an **approximate solution** when classic methods fail to find any exact solution

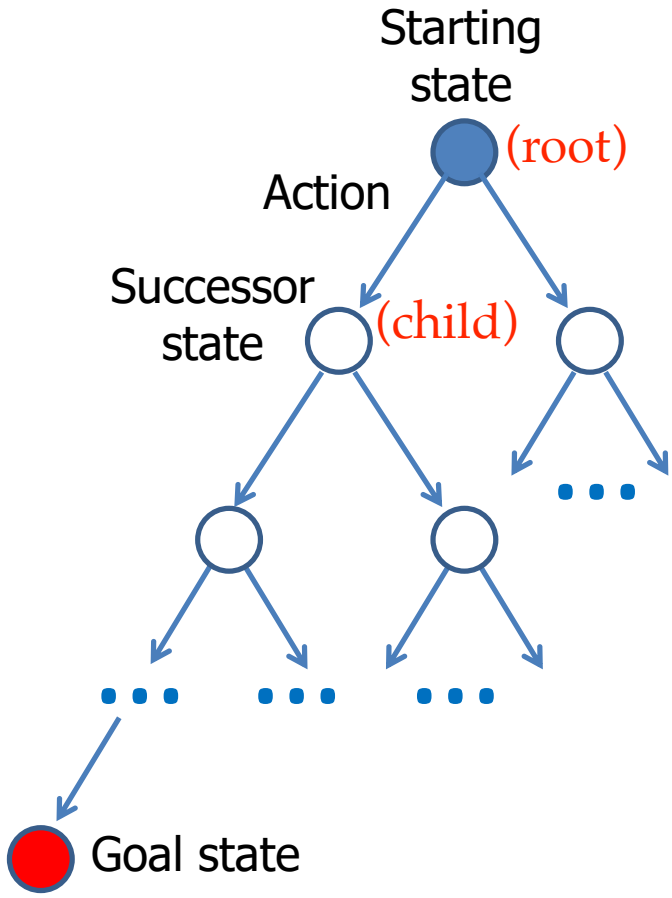
- **Tree search** doesn't check repeated states (may have infinite states)
- **Graph search** checks repeated states and is more efficient (should have sufficient space)

Tree search algorithms

1. Begin at the **start node**
2. **Explore state space** by generating a list of **all possible successors** of **already-explored** states (a.k.a. **expanding states**)
3. Maintain a **frontier** or a list of **unexpanded states**
4. At each step, **pick a state** from the **frontier** to expand
5. Keep going and **evaluate** every generated state

   Is it a goal state?
6. 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**



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

initialize the frontier using the initial state of *problem*

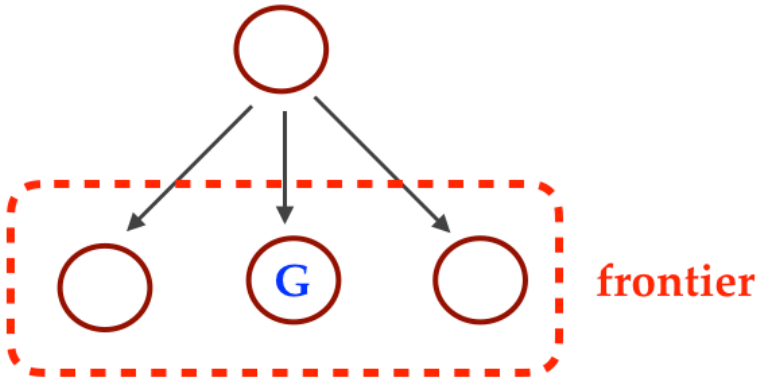
loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

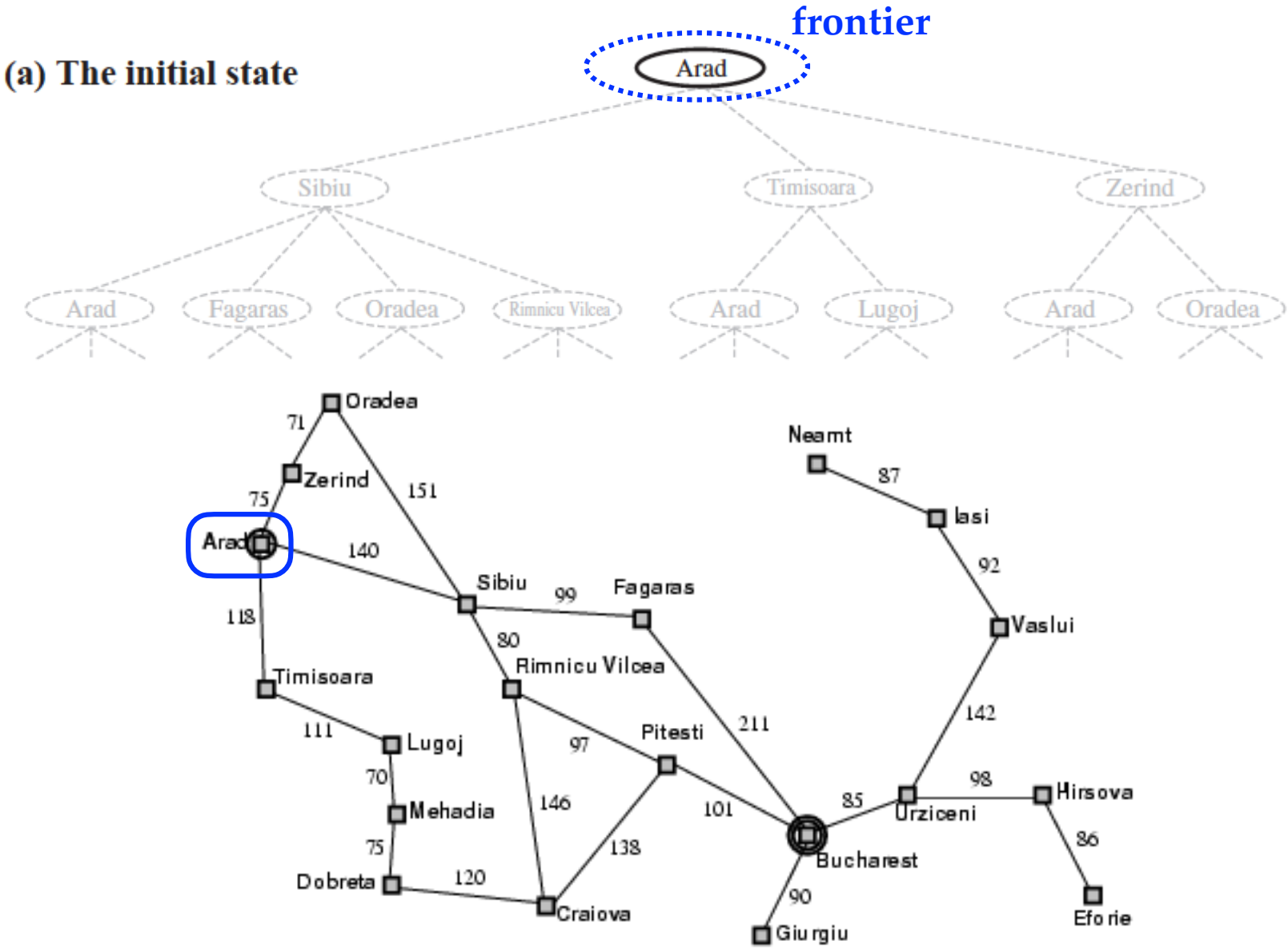
if the node contains a goal state **then return** the corresponding solution

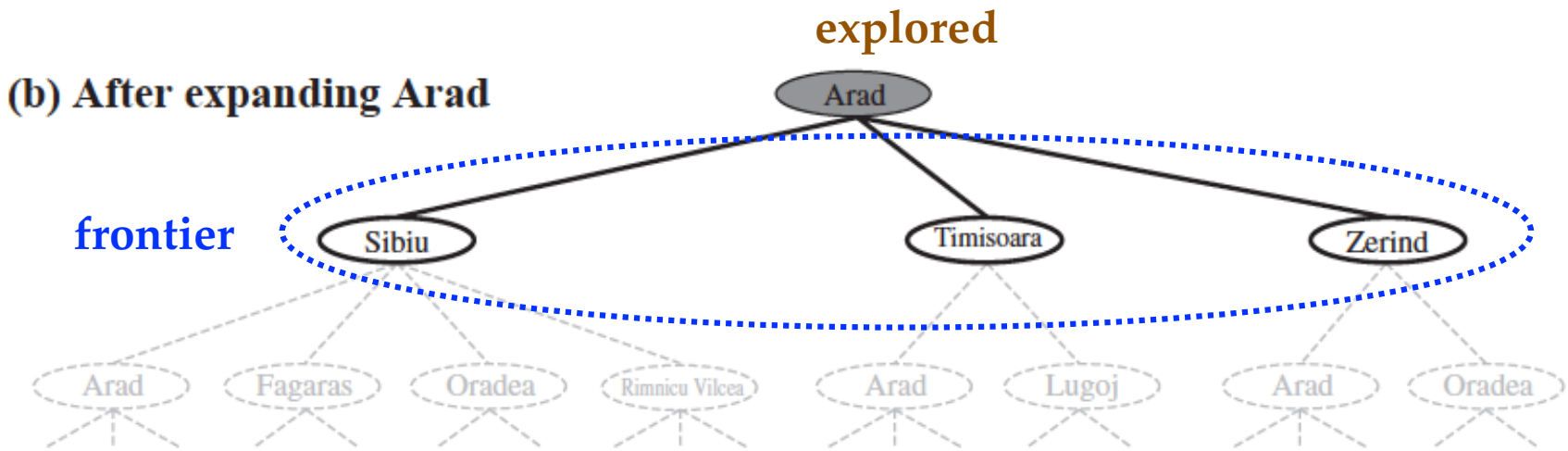
expand the chosen node, adding the resulting nodes to the frontier

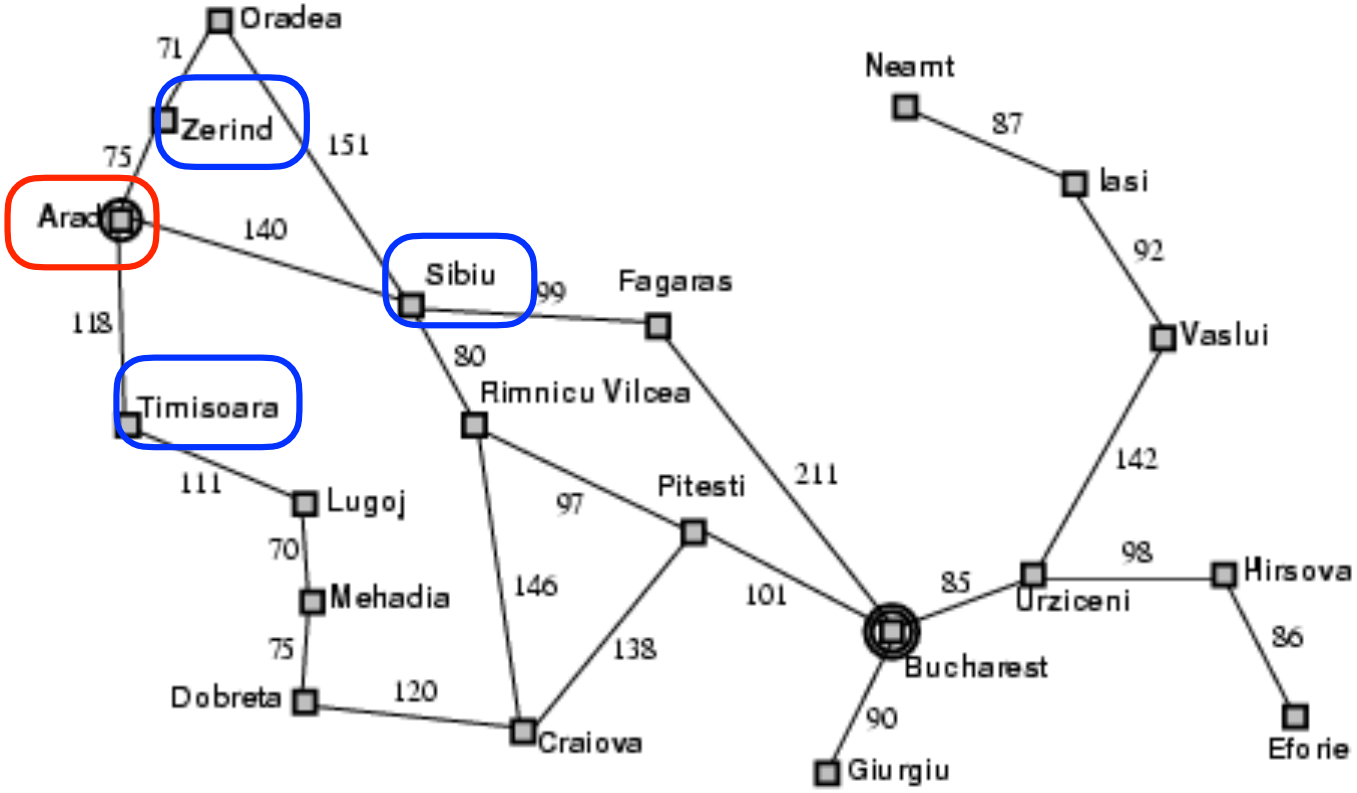


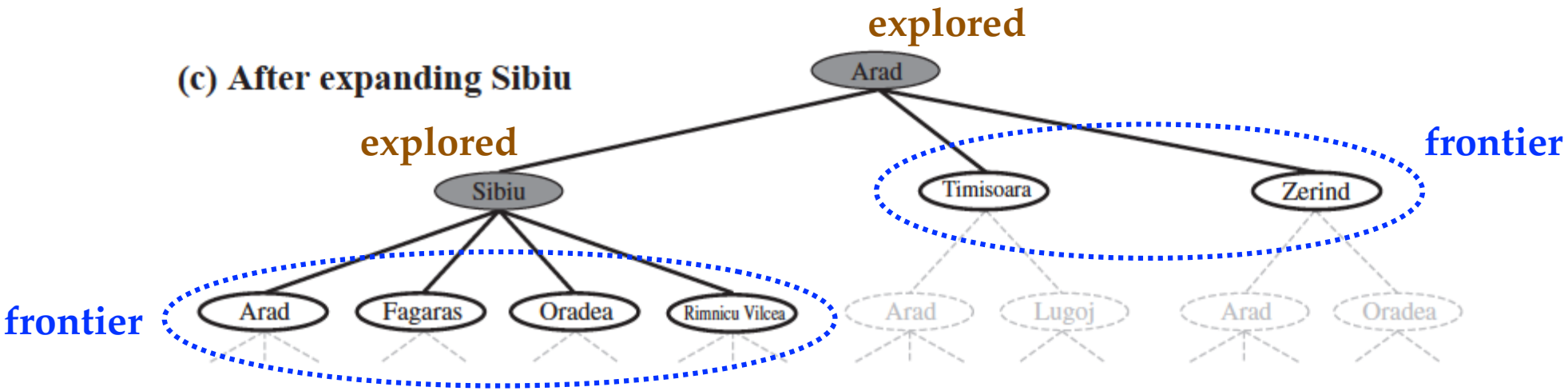
- Usually, we check whether a node is a "**goal**" (goal test) when it is **popped out** from the frontier
- This is because sometimes we cannot guarantee **optimal solution** if we do the "goal test" at the time we put a state into a frontier

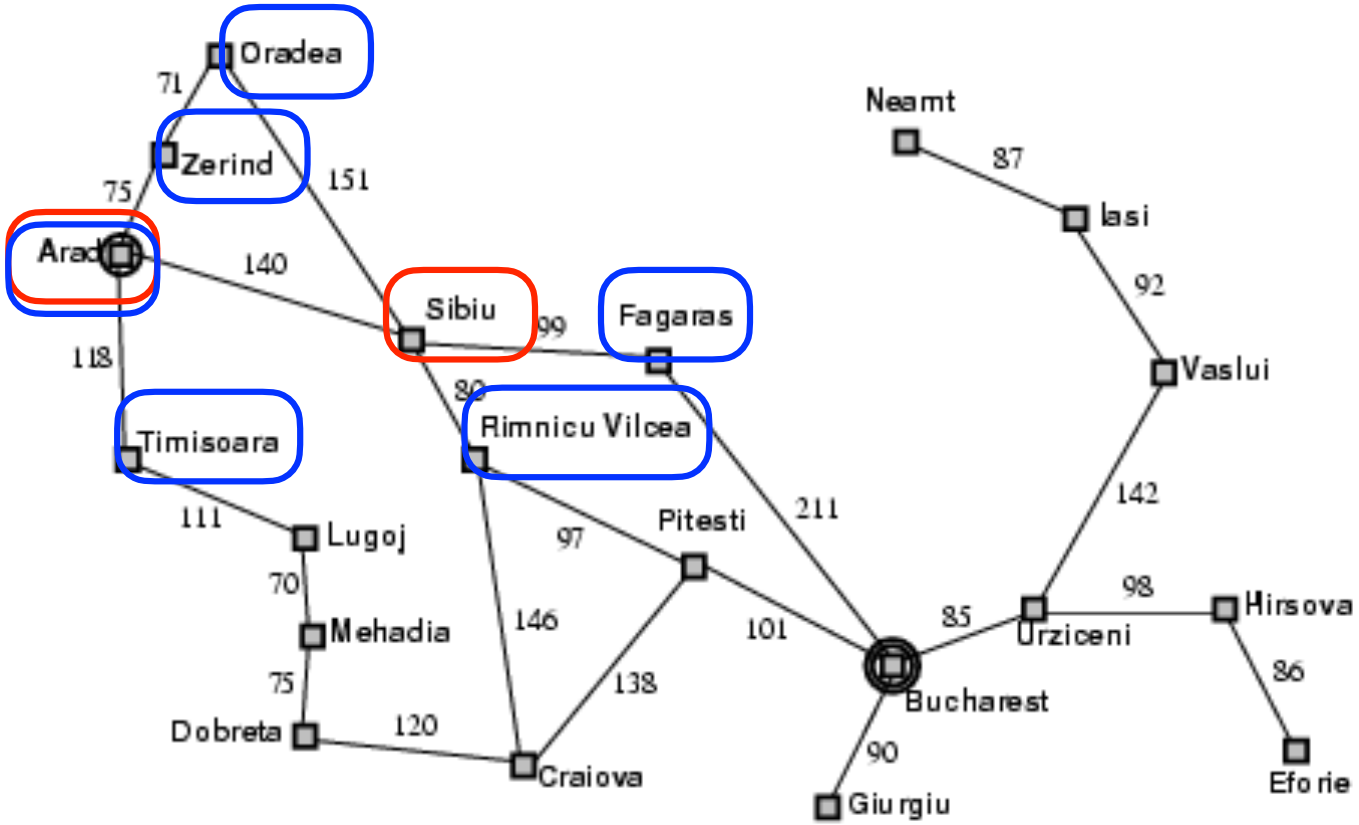
Tree Search Example




explored
(b) After expanding Arad
frontier
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Bucharest
90
Giurgiu
Neamt
87
Iasi
92
Vaslui
142
85
98
Urziceni
Hirsova
86
Eforie


(c) After expanding Sibiu
explored
explored
frontier
frontier
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

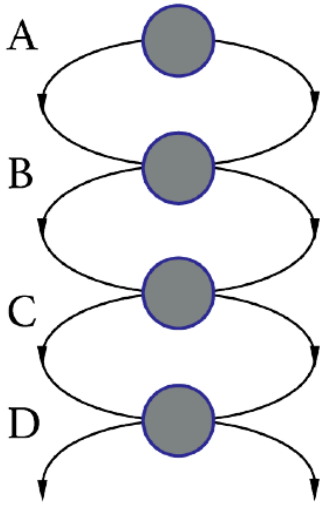

Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Timisoara
Rimnicu Vilcea
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Bucharest
90
Giurgiu
Neamt
87
Iasi
92
Vaslui
142
98
85
Urziceni
Hirsova
86
Eforie

Repeated States in Tree Search

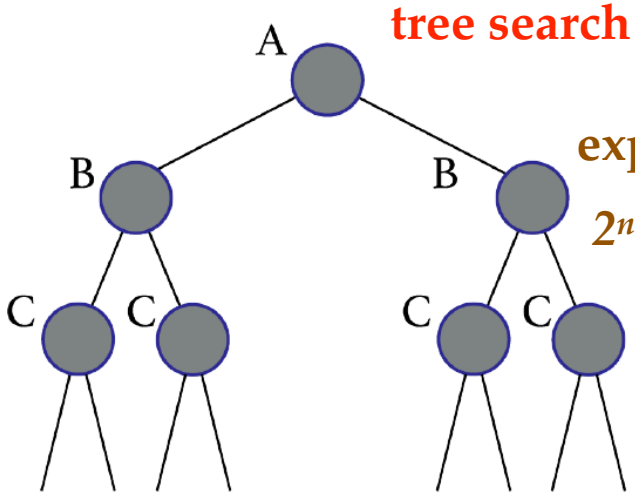
- **Failure to detect repeated states** can turn a **linear** problem into an **exponential** problem
- Use a **queue** to record **explored** states
- For **fast detection** of repeated states, **hashing** techniques are usually adopted (**hash table**)

linear problem

***n* states**



tree search

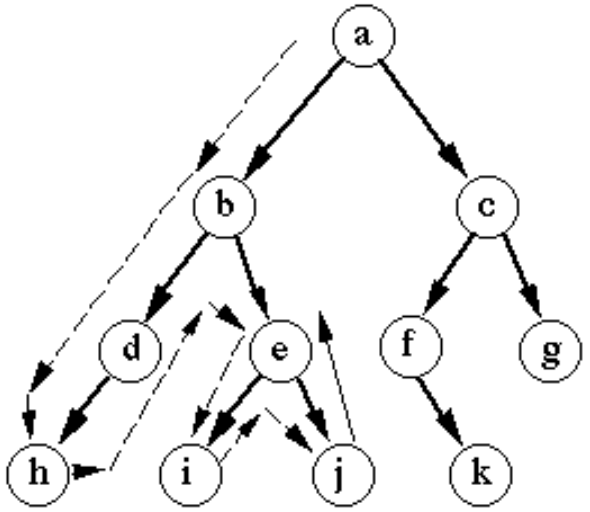


exponential problem

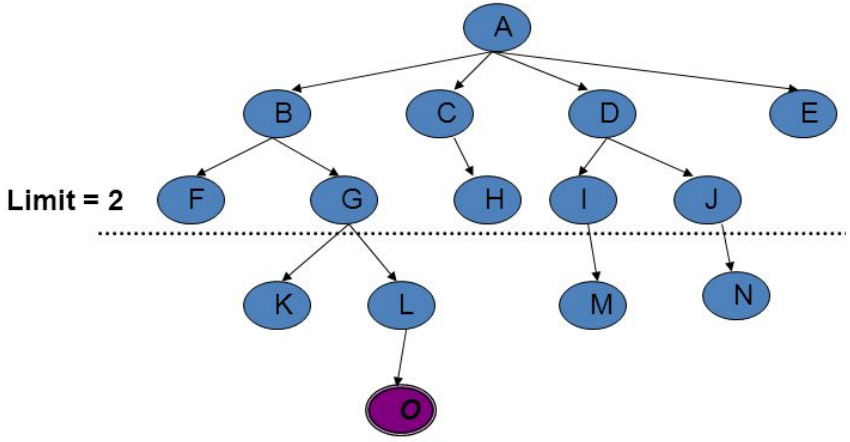
2^n states (repeated)

Solutions to Repeated States

- **Graph search**
  - **Faster (efficient)**, but **memory inefficient**
  - Never generate a state generated before
    - Must keep track of **all possible states** (use a lot of memory)
      - e.g., 8-puzzle problem has 9! = 362,880 states
    - Approximation for **DFS** (Depth First Search): only **avoid repeated states** in its **(limited) memory**
    - Approximation for **DLS** (Depth Limited Search): **avoid infinite loops** by checking path back to root

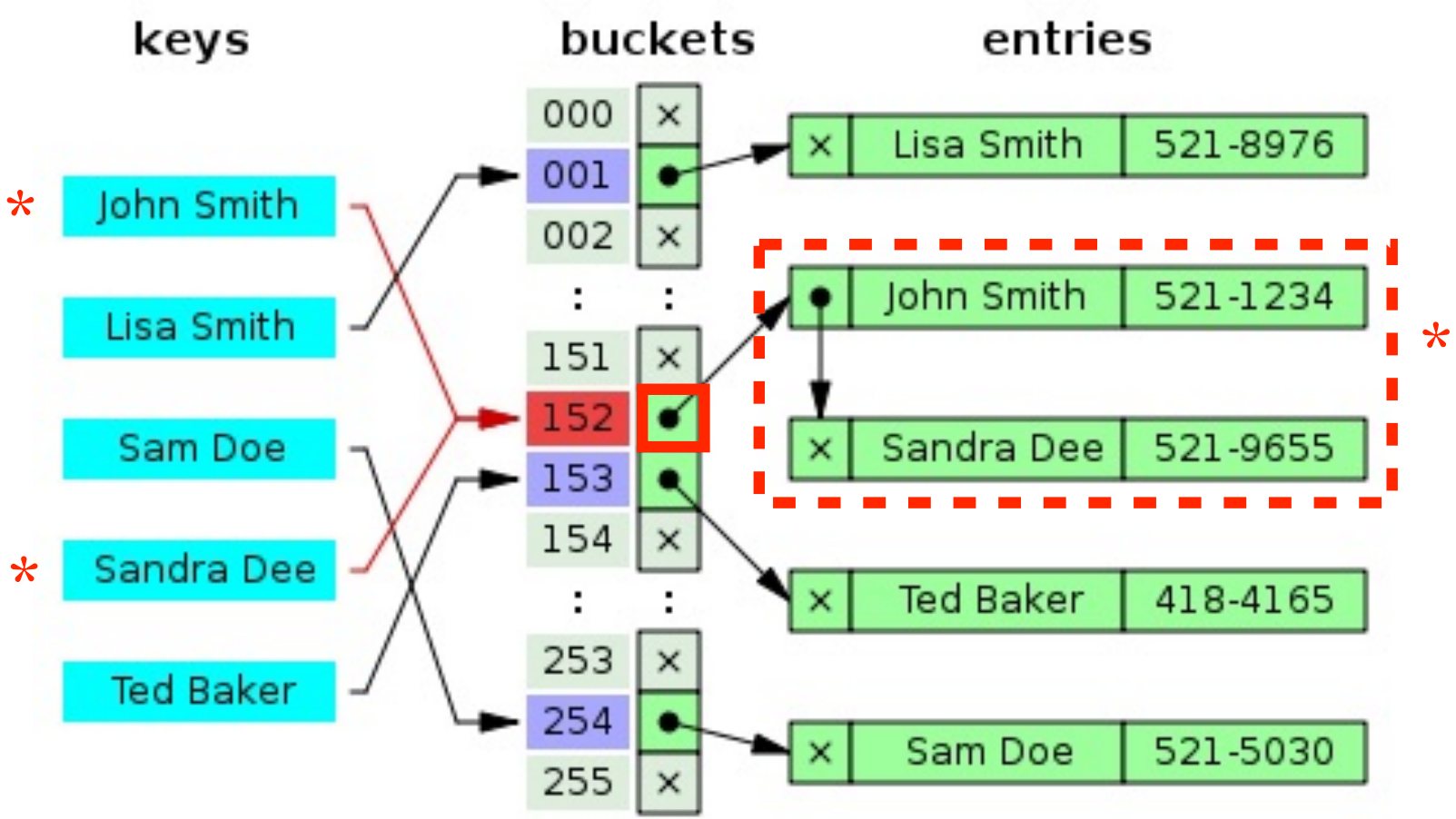


Depth-first search



Depth-limited search

- "**Visited (repeated)**?" test usually implemented as a **hash table**



"John Smith" and "Sandra Dee" are different but have the same hash value

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

initialize the frontier using the initial state of *problem*

initialize the explored set to be empty

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

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

add the node to the explored set

expand the chosen node

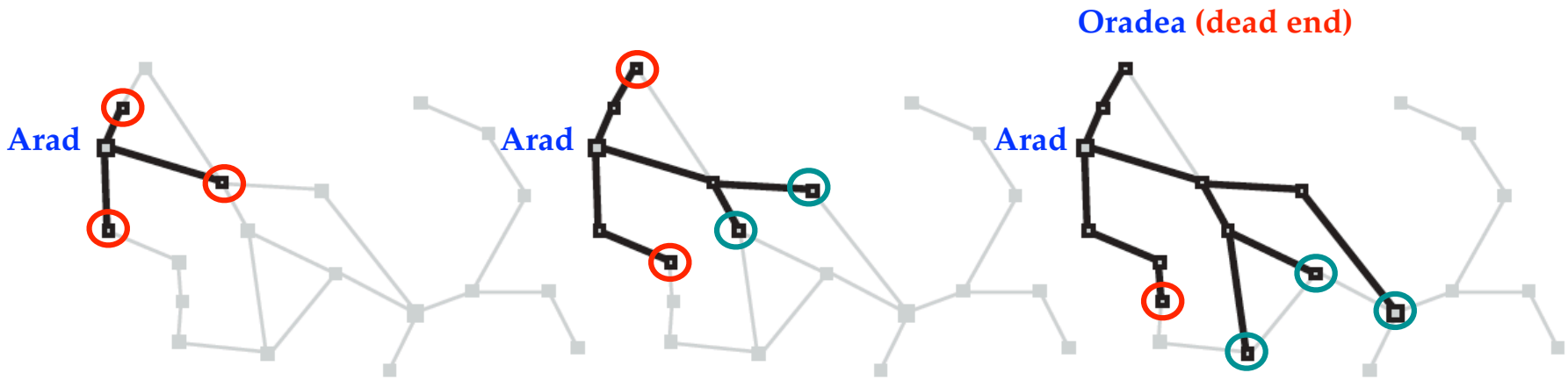
if not in the frontier or explored set

add the resulting nodes to the frontier

(**explored set** is used to check if there is repeated state)

Search Tree

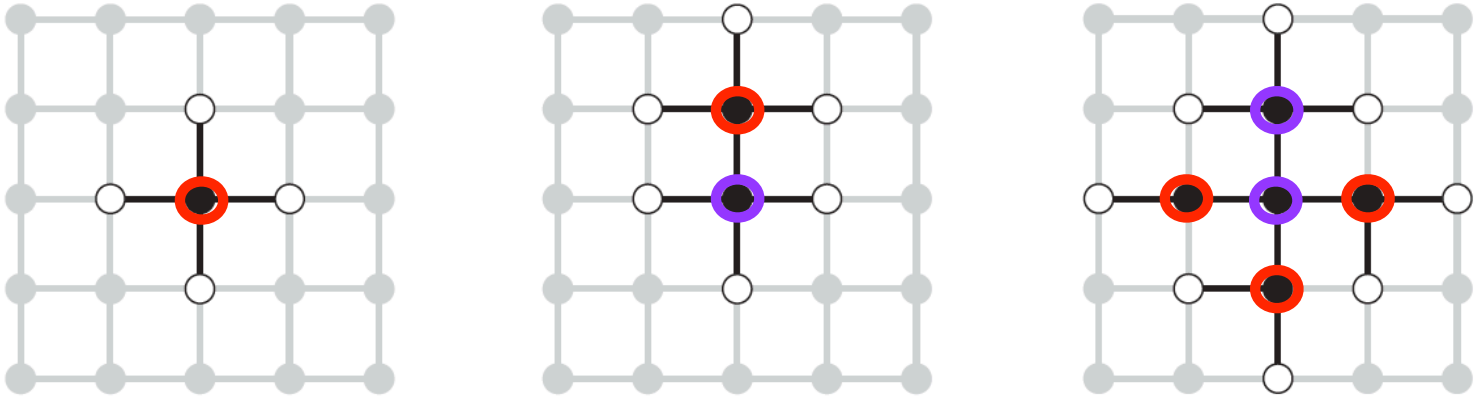
- A sequence of **search trees** generated by a graph search on the Romania problem
- At each stage, we **extend** each path by one step
- At the **third** stage, the northernmost city (**Oradea**) has become a **dead end**



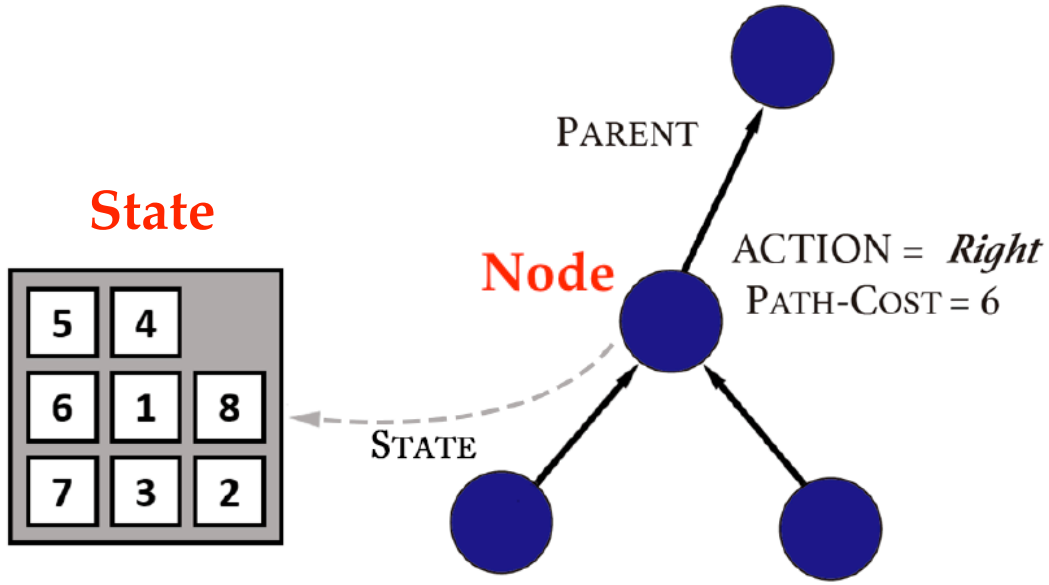
Graph Search & Frontier Separation

- The frontier **separates** the state space into **explored** and **unexplored** regions (**loop invariant proof**)

frontier
explored
unexplored



- **State** vs. **node** (map **state** to **node**)
  - **State**: a representation of a **physical configuration**
  - **Node**: a **data structure** constituting part of a search tree contains <u>much more</u> info. such as
    - **state, parent node, action, depth, path cost g(x)**



function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node

  return a node with

    s′ STATE = *problem*.RESULT(*parent*.STATE, *action*),

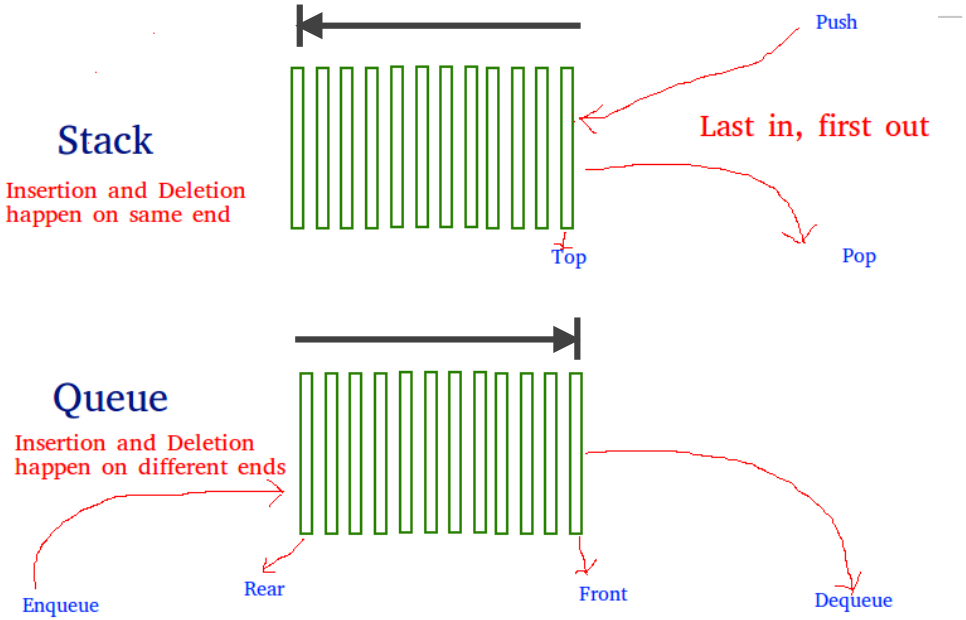
      PARENT = *parent*, ACTION = *action*,

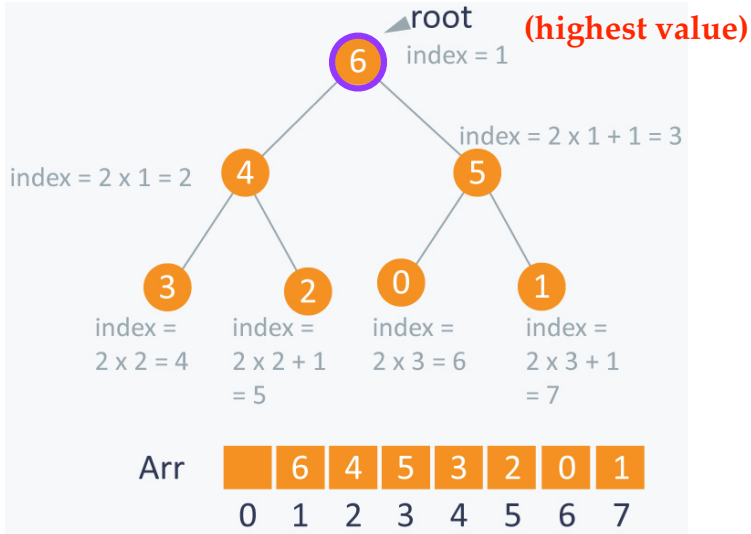
      PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)

s *a*

Infrastructure for Search Algorithms

- The appropriate data structure to maintain the frontier is a **queue** (memory buffer)
- Can be
  - **LIFO** (a.k.a. **stack**)
  - **FIFO** (a.k.a. **queue**)
  - **Priority queue** (**heap**)



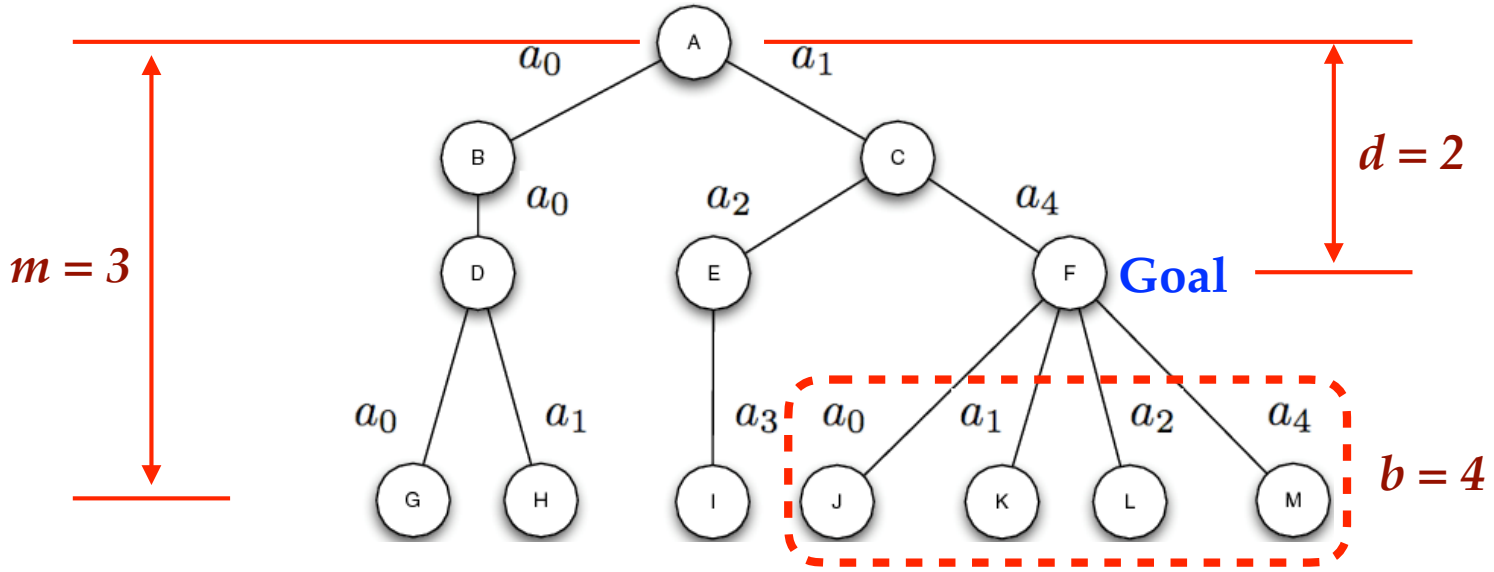


Tree Search Algorithms

- A **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** (given multiple goals)?
  - **Time complexity** - **number of nodes** generated/expanded
  - **Space complexity** - maximum number of nodes in **memory**

AI algorithms usually should concern more space complexity than time complexity.

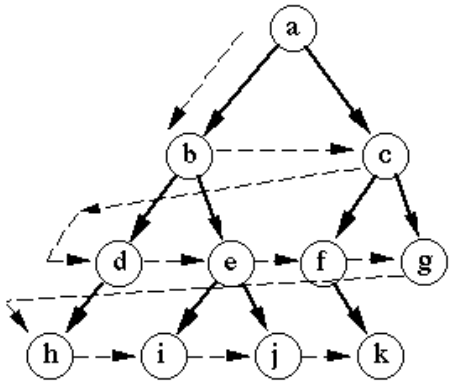
- Time and space complexity are measured in terms of
  - ***b*** - **max branching factor** of the search tree
  - ***d*** - **depth** of the **least-cost** solution
  - ***m*** - **max depth** of the state space (may be ∞)



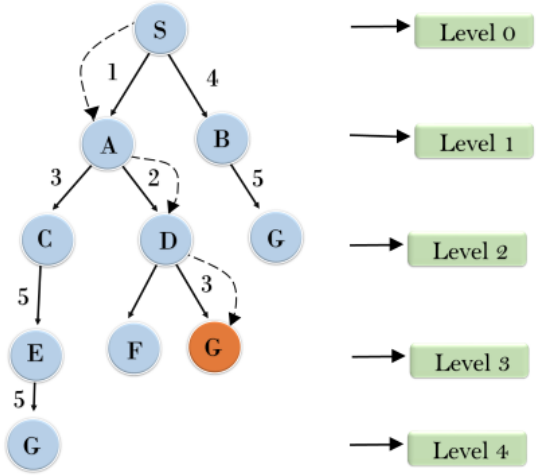
3.4 Uninformed Search Strategies (Blind Search)

- **Uninformed (blind)**
  - No clue whether one non-goal state (current exploration) is better than any other. Your search is **blind**
  - Uninformed strategies use only the **information available** in the **problem definition**, e.g.
    - Initial state, goal state, transition model, goal test, path cost, step cost

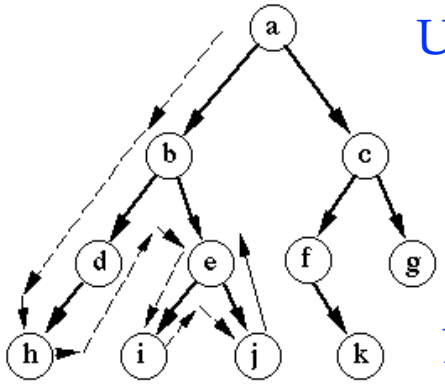
- **Blind searches**
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search



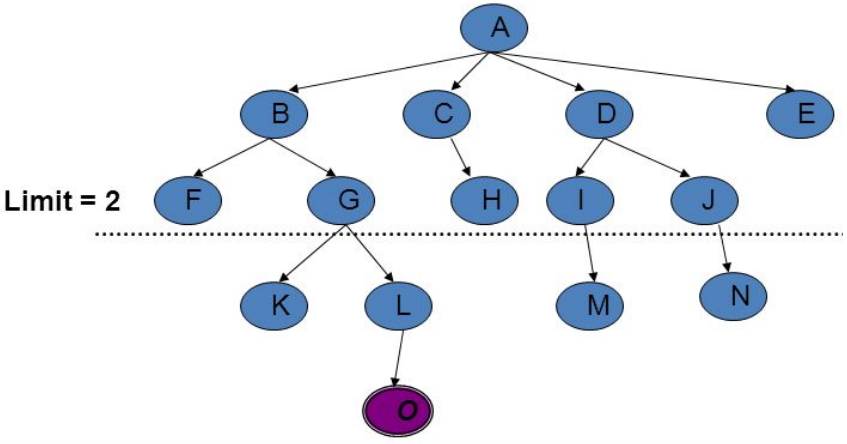
Breadth-first search



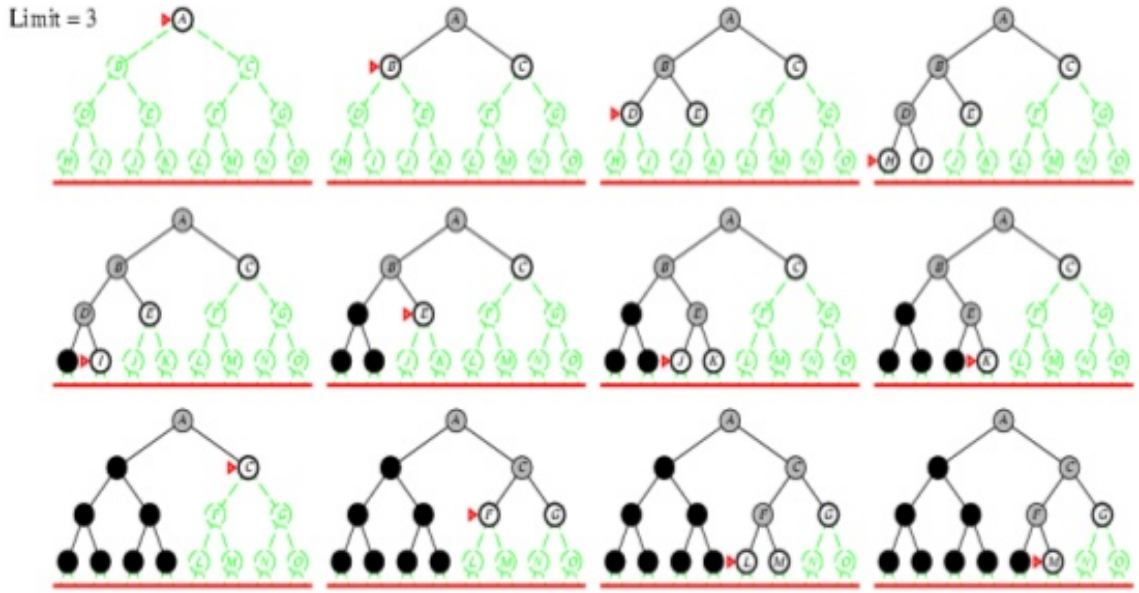
Uniform-cost search



Depth-first search



Depth-limited search



Iterative deepening search (I=3)

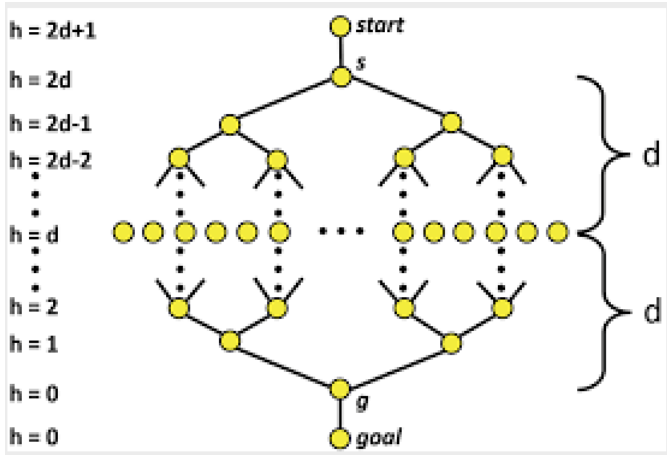
Uninformed Search Design Choices

- **Queue for frontier**
  - FIFO? LIFO? priority queue?
- **Goal-test**
  - Do goal-test when node **inserted** into frontier?
  - Do goal-test when node **removed** from frontier?
- **Tree search** or **graph search**
  - **Forget** expanded nodes? [tree search]
  - **Remember** expanded nodes? [graph search]

Queue for Frontier

- **FIFO (First In, First Out)**
  - Breadth-First Search (BFS)
- **LIFO (Last In, First Out)**
  - Depth-First Search (DFS)
- **Priority queue** sorted by path cost so far
  - Uniform Cost Search (UCS)

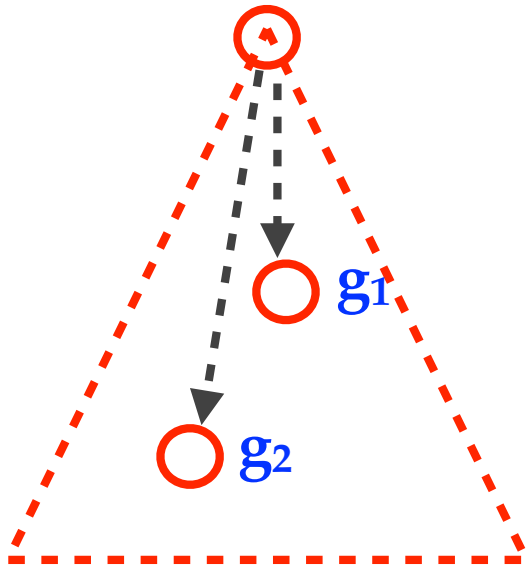
Bidirectional search



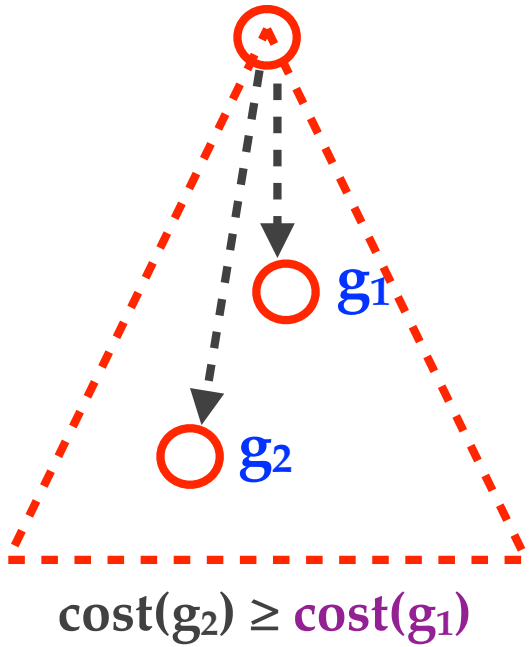
- Iterative Deepening Search (IDS) uses Depth-First Search
- Bidirectional Search can use either Breadth-First or Uniform Cost Search

When to do Goal-Test? When Generated? When Popped?

- Do Goal-Test when node is **popped from queue**
  IF you care about finding the **optimal path** AND your search space may have both **short expensive** and **long cheap** paths to a goal
  - Guard **against** a **short expensive** goal
  - e.g.,
    - Uniform Cost Search with **variable step costs**



- Otherwise, do Goal-Test when **node inserted**
  - e.g.,
    - Breadth First Search (BFS)
    - Depth First Search (DFS)
    - Uniform Cost Search when **cost** is a **non-decreasing function of depth only**
    - Depth-Limited Search (DLS)
    - Iterative-Deepening Search (IDS)



Goal Test for 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 ← INSERT ALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes (successors)
    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
```

frontier

Goal test after pop

Goal Test for General Graph Search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set (closed: set of repeated states)
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do                                                    Goal test after pop
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed   (set of repeated states)
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Goal Test for Breadth-First Graph Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Goal test before push

Goal Test for Uniform Cost Search (Sort by g)

- A^* is identical but uses $f = g + h$

Greedy best-first (GBFS) is identical but uses h

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

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

Goal test after popped

DLS: Depth-Limited Search

IDS: Iterative-Deepening Search

Goal Test for DLS & IDS

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

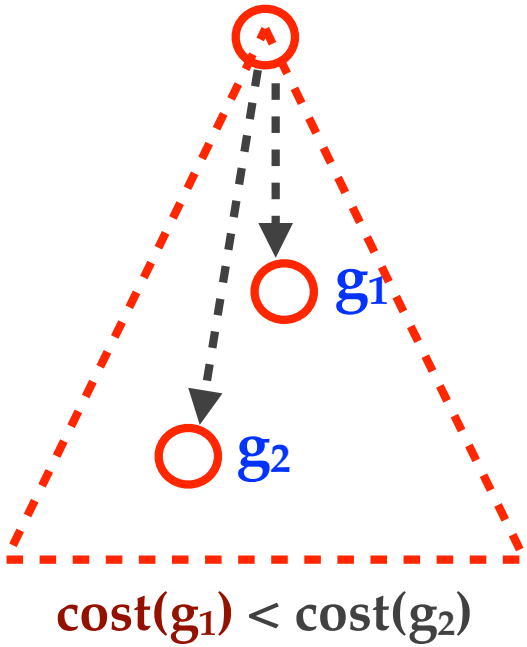
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Goal test before push

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Summary: When to do Goal-Test?

- For **BFS**, **DFS**, **DLS**, and **IDS**, the goal test is done when the **child node is generated**
  - These are **not optimal searches** in the general case
  - BFS and IDS are **optimal if cost is a (increasing) function of depth only**; then, **optimal goals are also shallowest goals** and so will be found first



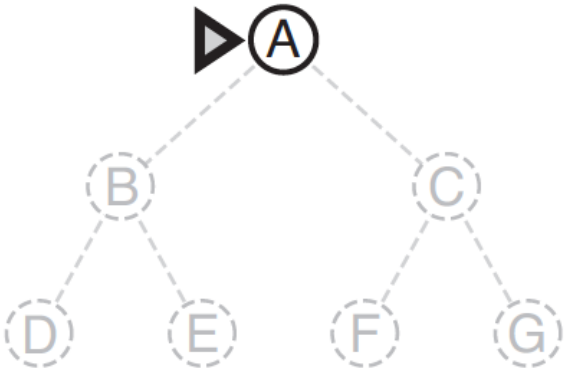
- For **UCS** (Uniform Cost Search) and **A*** the goal test is done when the node is **removed** from the queue
  - This precaution **avoids** finding a short expensive path before a long cheap path

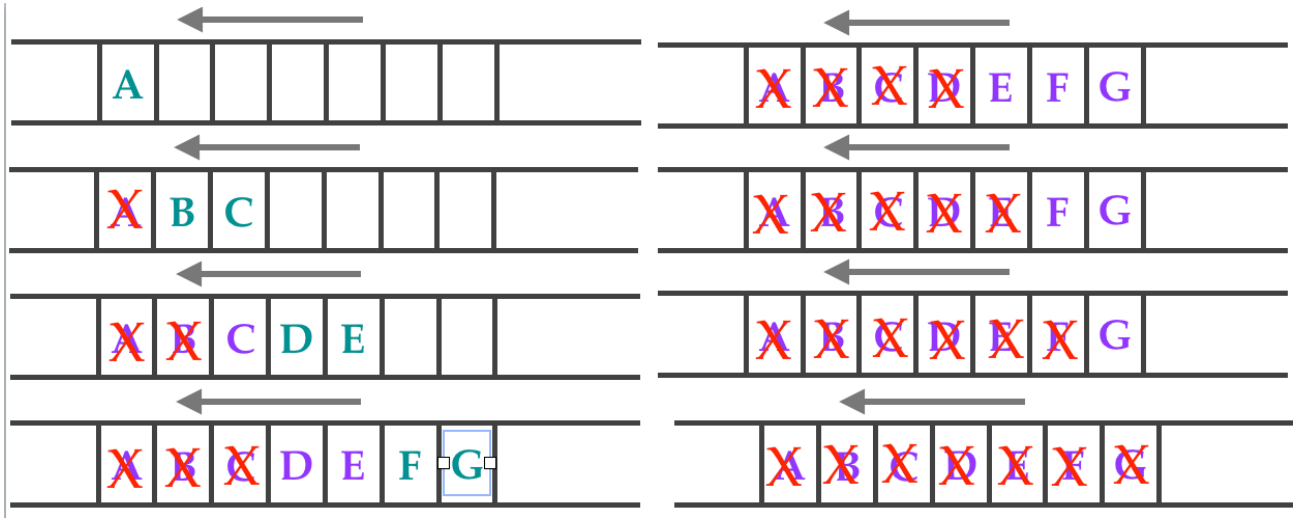
Breadth-First Search (BFS)

- Expand **shallowest unexpanded** node
- **Frontier** (or fringe): nodes in queue to be explored
- Frontier is a **first-in-first-out (FIFO)** queue, i.e., new successors go at end of the queue
- **Goal-Test when inserted**

Initial state = A
Is A a goal state?

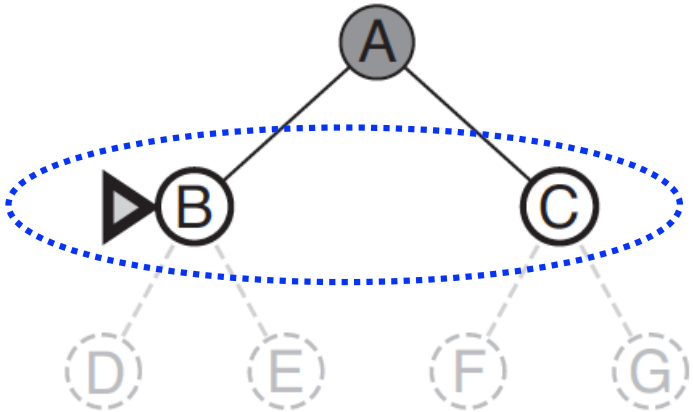
Put A at end of queue.
frontier = [A]





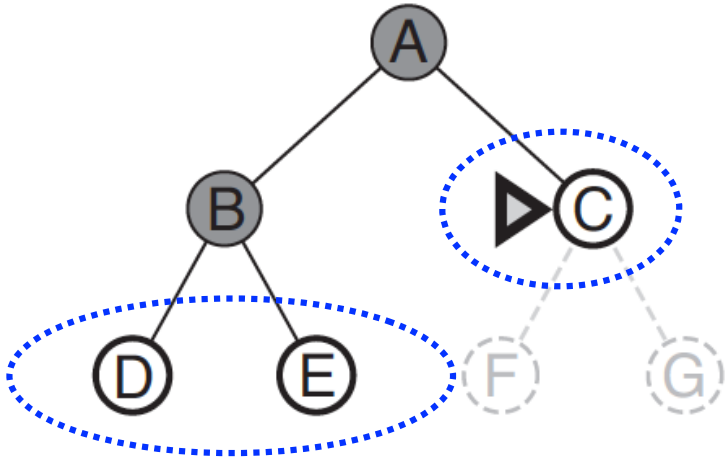
Expand A to B, C.
Is B or C a goal state?

Put B, C at end of queue.
frontier = [B,C]



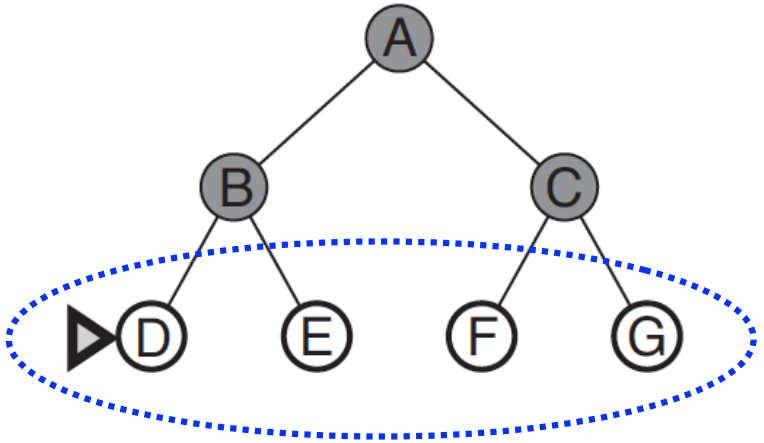
Expand B to D, E
Is D or E a goal state?

Put D, E at end of queue
frontier=[C,D,E]



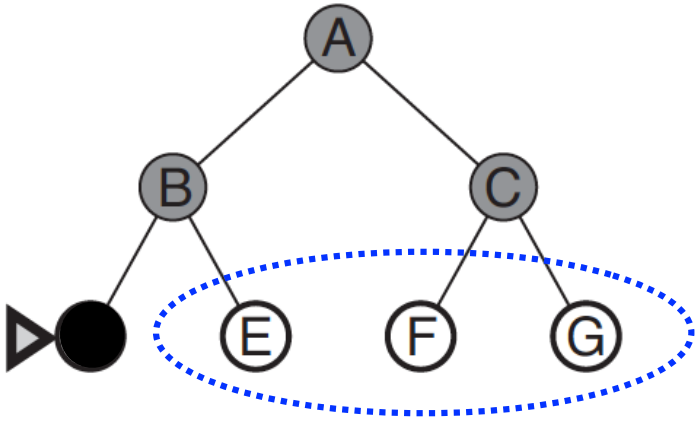
Expand C to F, G.
Is F or G a goal state?

Put F, G at end of queue.
frontier = [D,E,F,G]



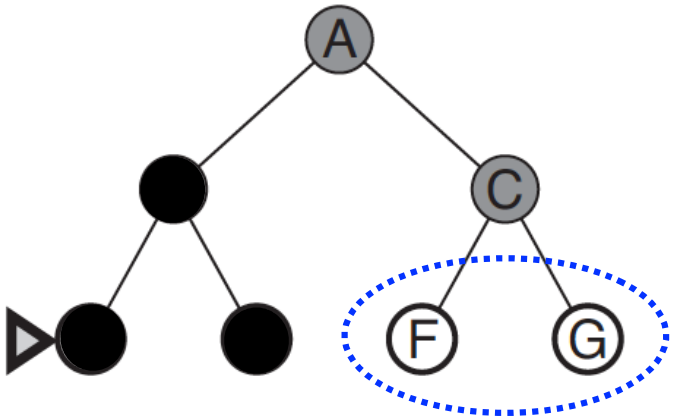
Expand D to no children.
Forget D.

frontier = [E,F,G]

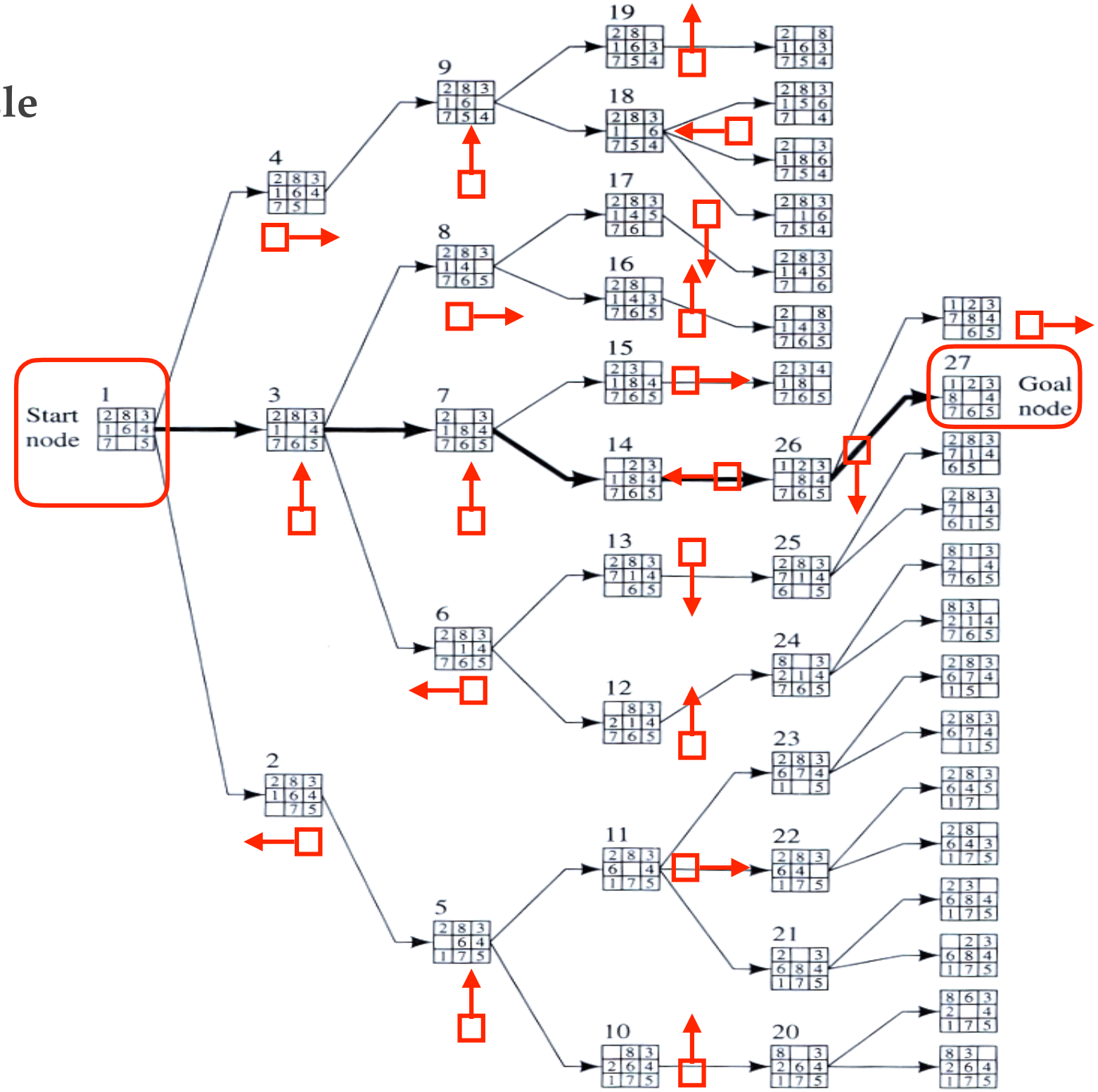


Expand E to no children.
Forget B,E.

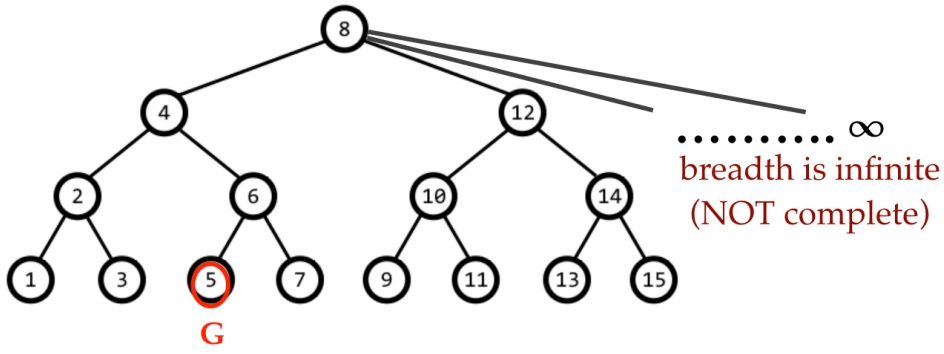
frontier = [F,G]

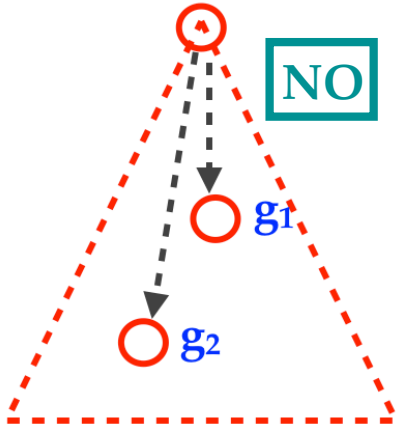


BFS for 8-puzzle

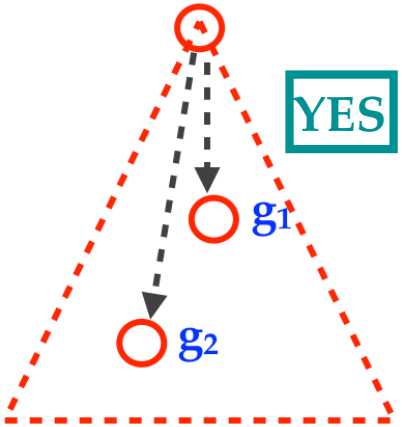


- **Properties of BFS**
  - **Completeness**: **Yes**, it always reaches a goal (if *b* is **finite**)
  - **Optimality**
    - **No,** for general cost functions
    - **Yes** only if the **path cost** is a **non-decreasing function of depth** $f(d) \geq f(d\text{-}1)$, e.g., step-cost = constant, as in the 8-puzzle
      - All optimal goal nodes occur on the **same level**
      - **Optimal goals** are always **shallower** than non-optimal goals
      - An optimal goal will be **found before** any non-optimal goal





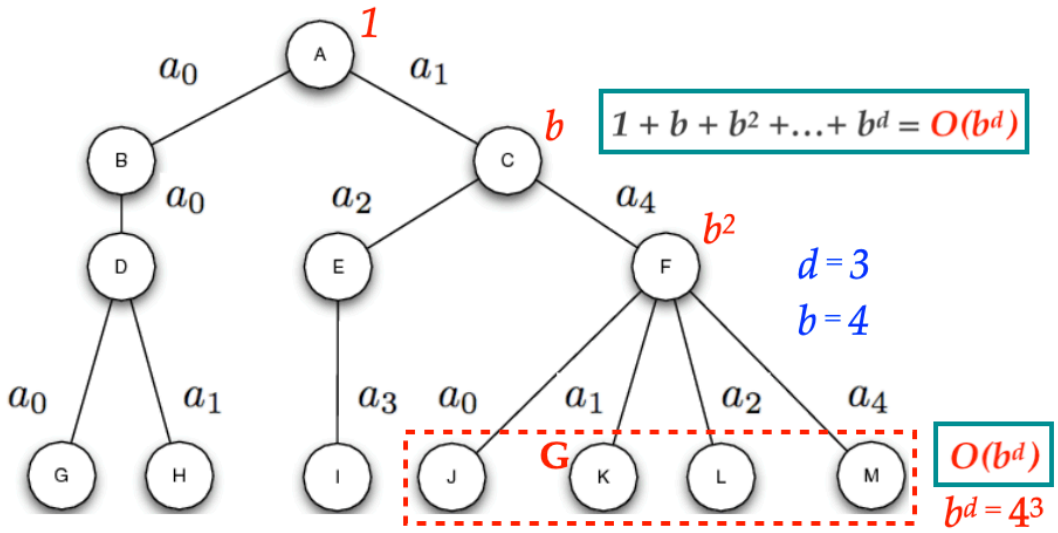
Can't guarantee
$cost(g_1) < cost(g_2)$



$cost(g_1) \leqq cost(g_2)$

d - **depth** of the **least-cost** solution

- **Time complexity**: $1 + b + b^2 + \ldots + b^d = O(b^d)$
  - Or $O(b^{d+1})$ if **goal test** is applied **after expansion** (this is the number of nodes we generate)



- **Space complexity**: $O(b^d)$
  - Keep each **node in frontier** in memory (i.e., it is dominated by the size of the frontier)
  - e.g. assume branching factor $b = 7$, if 10^6 nodes/sec, 10^3 bytes/node, $d = 7$ then 1GB/sec and 24hrs = 86.4TB

Space is the big problem!

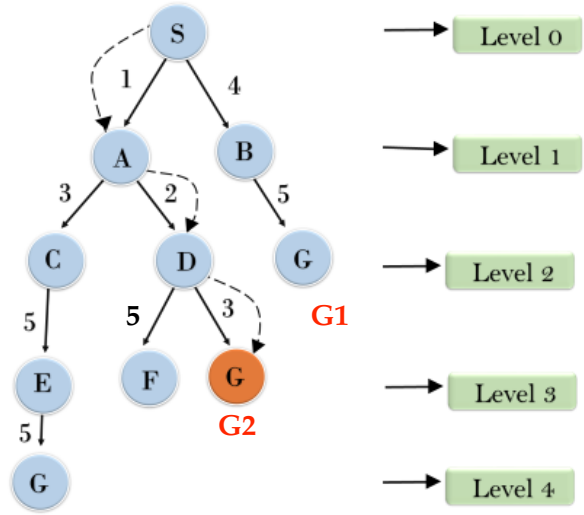
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6 *d=7*	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

10GB

- Time and memory requirements for breadth-first search
- The numbers shown assume branching factor **b = 7; 10^6 nodes/sec, 10^3 bytes/node**

Uninform-Cost Search

- Expand the unexpanded node with the **lowest path cost** $g(n)$
- **Equivalent** to BFS if **step costs are all equal**
- Frontier is a **priority queue**, i.e., new successors are merged into the queue sorted by $g(n)$
  - Can **remove successors** already on queue w/ **higher** $g(n)$
    - Saves memory, costs time
- Goal-test when node is **popped off** queue



Uniform-cost search

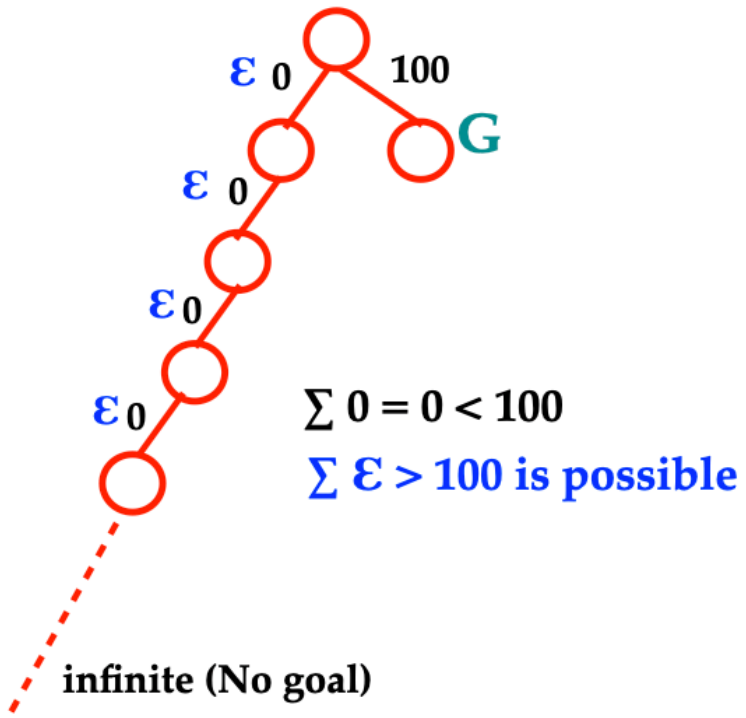
Explored		Frontier
S	A:1, B:4	A, B
S, A	B:4, C:4, D:3	B, C, D
S, A, D	B:4, C:4, F:8, G2:6	B, C, F, G2
S, A, D, B	C:4, F:8, G2:6, G1:9	C, F, G2, G1
S, A, D, B, C	G2:6, G1:9, E:9	F, G2, G1, E
S, A, D, B, C, G2		G2, G1, E

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

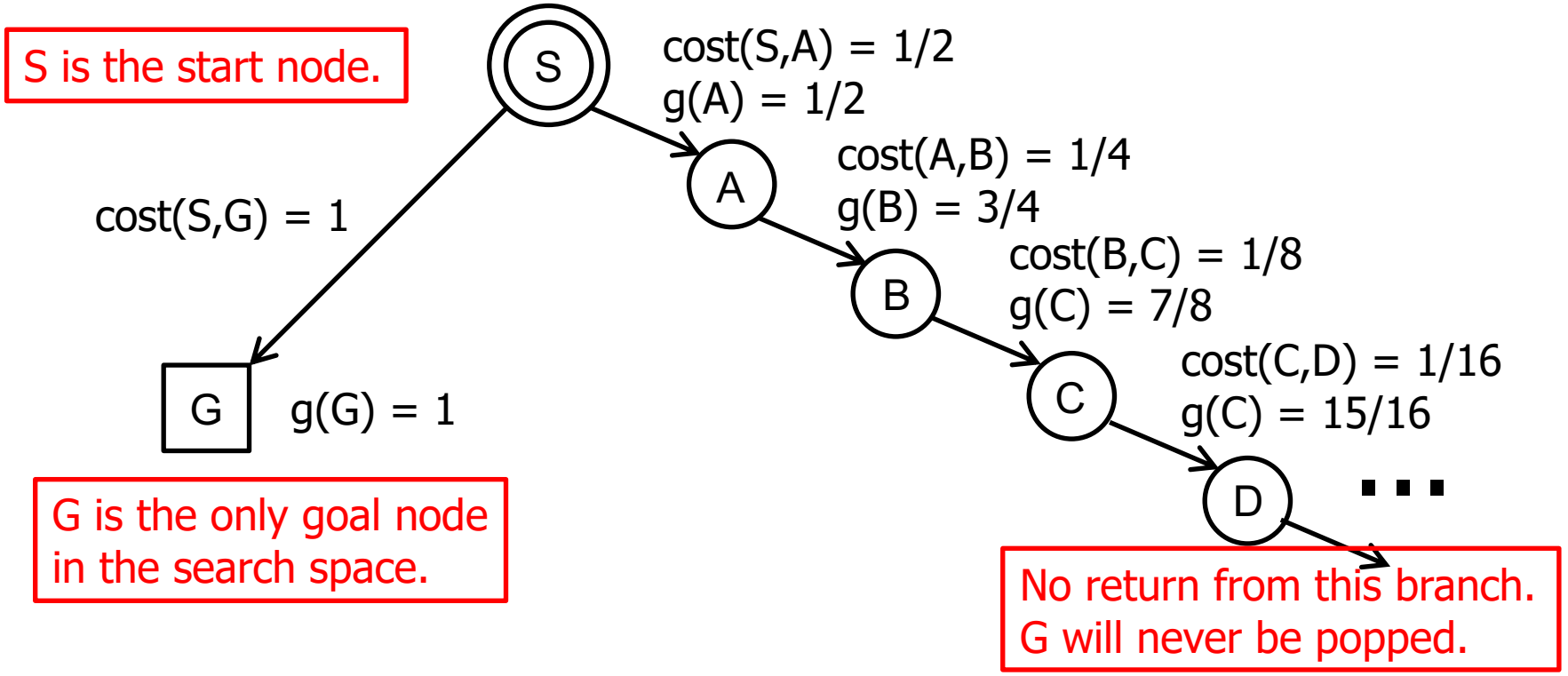
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
 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

Goal test after pop

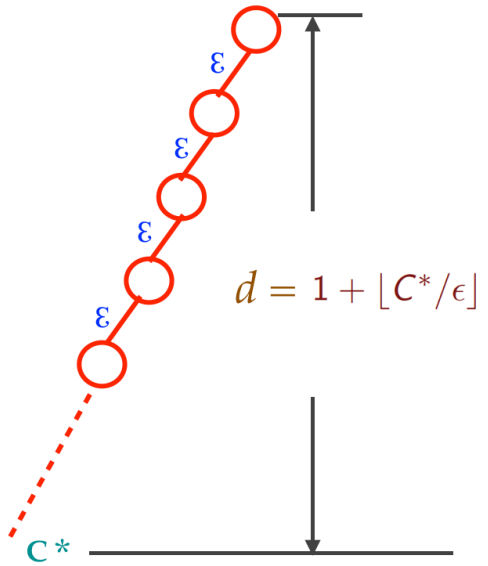
- **Implementation**: Frontier = queue ordered by path cost (priority queue)
- **Properties of Uniform-Cost Search**
  - **Completeness**: **Yes**, if *b* is **finite** and **step cost** $\geq \varepsilon > 0$ (otherwise it can get stuck in infinite loops, e.g, $\sum 0 = 0 < 100$)
  - **Optimality**: **Yes** - nodes expanded in **increasing order** of ***g*(*n*)** (for any step cost $\geq \varepsilon > 0$)



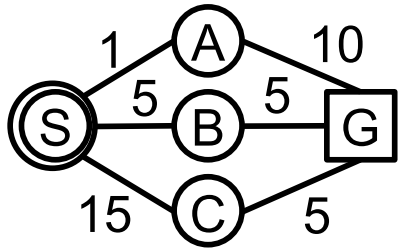
- Why require **step cost ≥ ε > 0**?
  - Otherwise, an **infinite regress** is possible
  - [**Step cost = 0**] e.g, $\sum 0 = 0 < 100$
  - [**Step cost has no lower bound ≧ ε > 0**]
    e.g. $\sum_{n=1\ to\ \infty} 2^{-n} = 1.\ (1, ½, ¼, \ldots)$



- **Time complexity**: # of nodes with **g ≦ cost of optimal solution**
  - **Maximum depth** is given by $1 + \lfloor C^*/\epsilon \rfloor$, where C^* is the cost of the optimal solution
  - Time complexity is $1 + b + b^2 + \ldots + b^d \approx O(b^{1+d})$ $= O(b^{1+\lfloor C^*/\epsilon \rfloor})$ where $d = \lfloor C^*/\epsilon \rfloor$
- **Space complexity**: # of nodes with **g ≦ cost of optimal solution**, $O(b^{1+\lfloor C^*/\epsilon \rfloor}) \approx O(b^{1+d})$



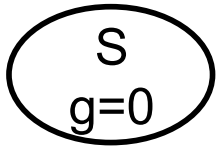
Example hand-simulated search: **Search tree** method

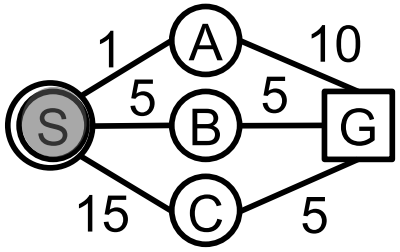


Route finding problem.
Steps labeled w/cost.

Order of node expansion: _____

Path found: _____ Cost of path found: _____



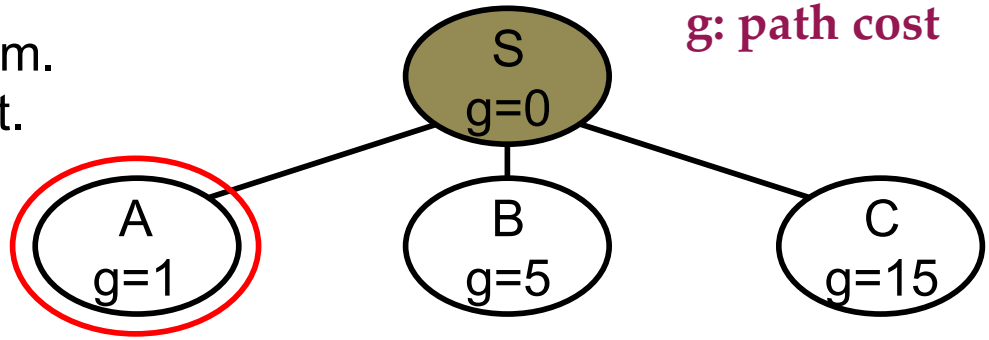


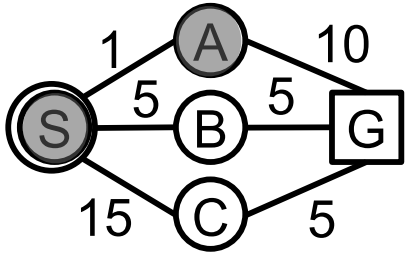
Route finding problem.
Steps labeled w/cost.

Order of node expansion: S _____

Path found: _____ Cost of path found: _____

g: path cost



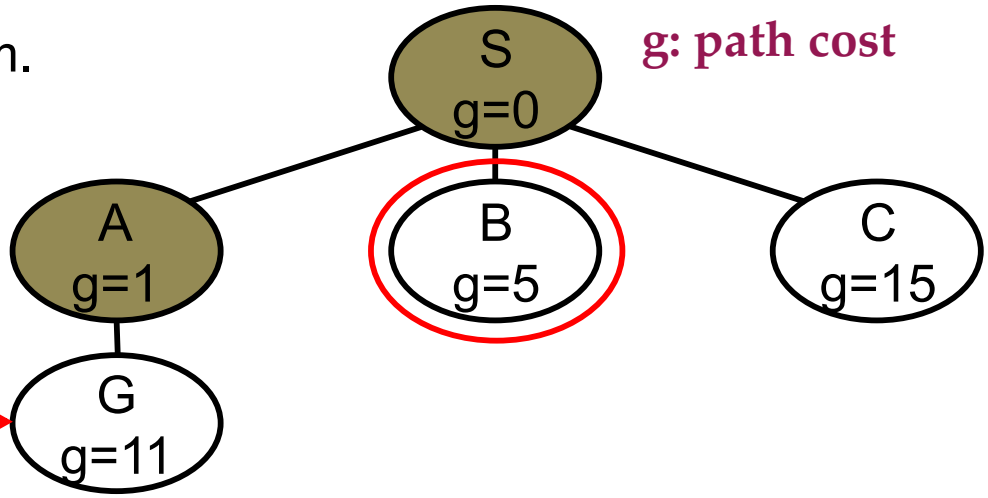


Order of node expansion: S A

Path found: _____ Cost of path found: _____

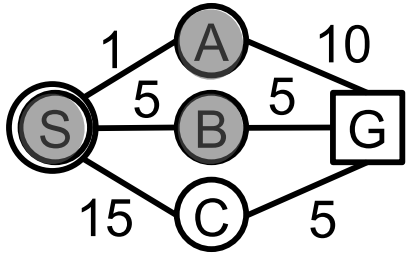
Route finding problem.
Steps labeled w/cost.

This early expensive goal node will go back onto the queue until after the later cheaper goal is found.



g: path cost

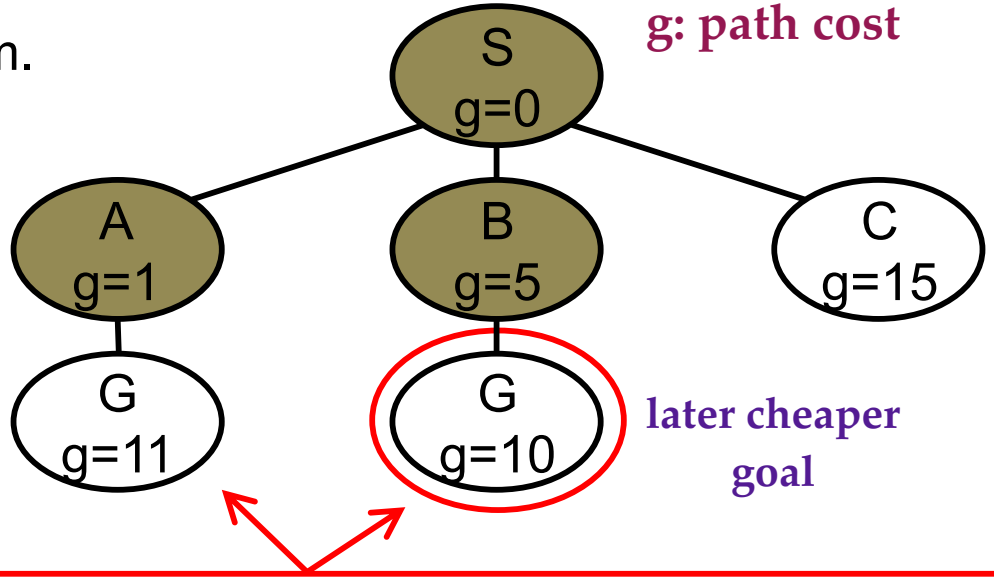
early expensive goal



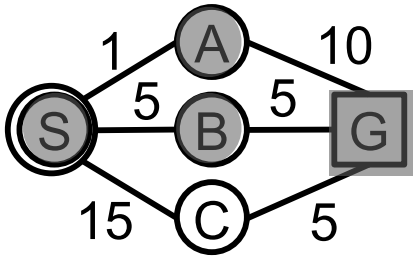
Order of node expansion: S A B _____

Path found: _____ Cost of path found: _____

Route finding problem.
Steps labeled w/cost.



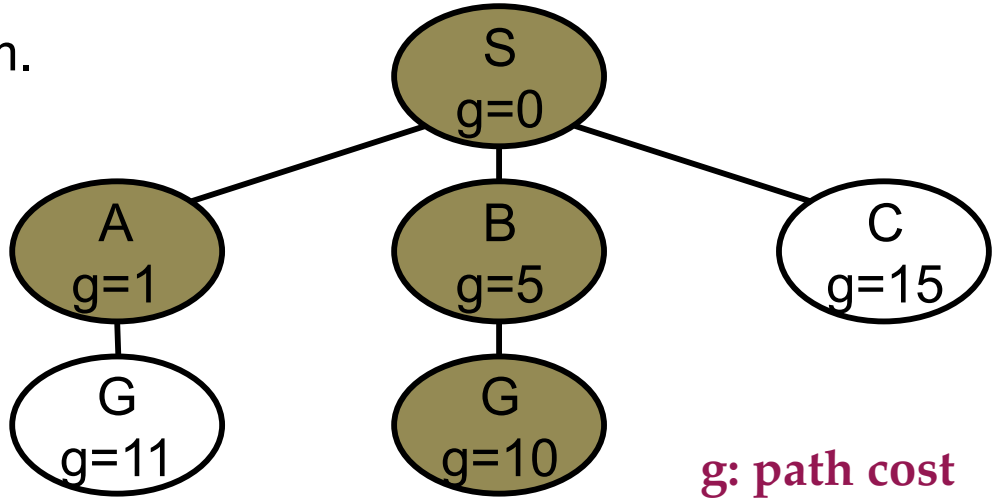
If we were doing graph search we would remove the higher-cost of identical nodes and save memory. However, UCS is optimal even with tree search, since lower-cost nodes sort to the front.



Route finding problem.
Steps labeled w/cost.

Order of node expansion: S A B G

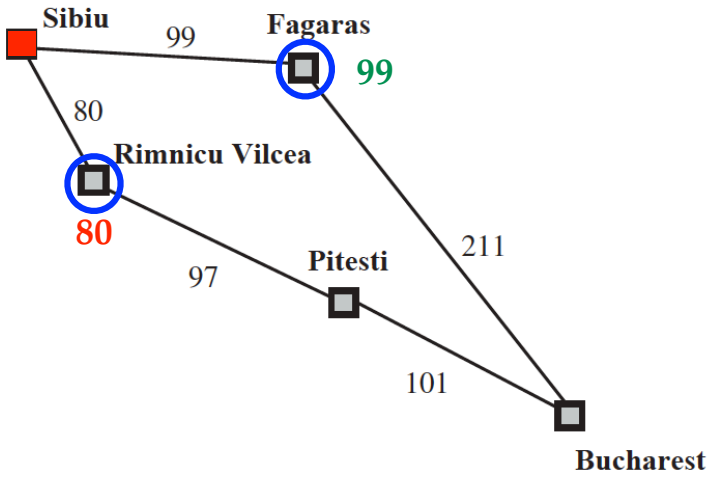
Path found: S B G Cost of path found: 10



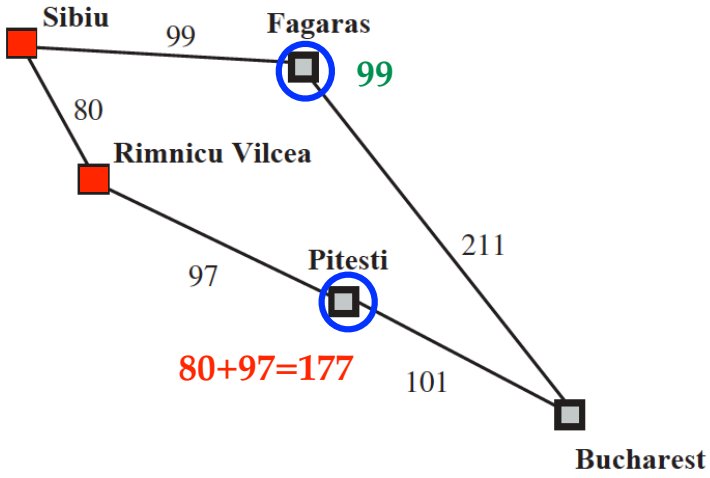
Technically, the goal node is not really expanded, because we do not generate the children of a goal node. It is listed in "Order of node expansion" only for your convenience, to see explicitly where it was found.

Romania example

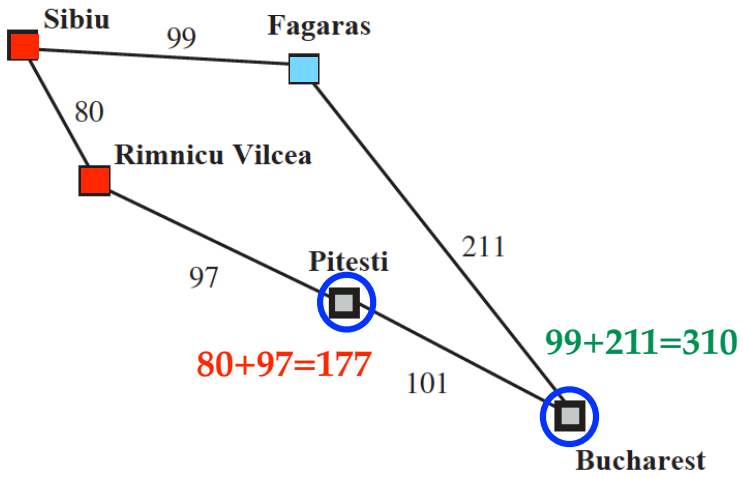
- The **successors** of Sibiu are Rimnicu Vilcea (80) and Fagaras (99)



- The **least-cost node**, Rimnicu Vilcea (80), is expanded next, adding Pitesti (97) with cost 80 + 97=177



- The **least-cost node** is now Fagaras (99), so it is expanded, adding Bucharest (211) with cost 99+211=310

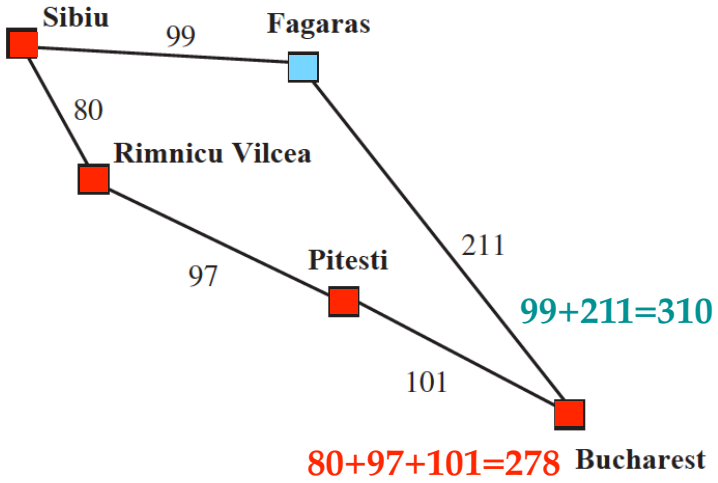


Goal-test when node is **popped** off queue

- Now **a goal node has been generated**, but uniform-cost search **keeps going**, choosing Pietist for expansion and adding a second path to Bucharest (101) with cost 80+97+101= 278

- Now the algorithm **checks** to see **if this new path is better than the old one**; it is, so the old one is discarded

- Bucharest, now with g-cost **278**, is selected for expansion and the solution is returned



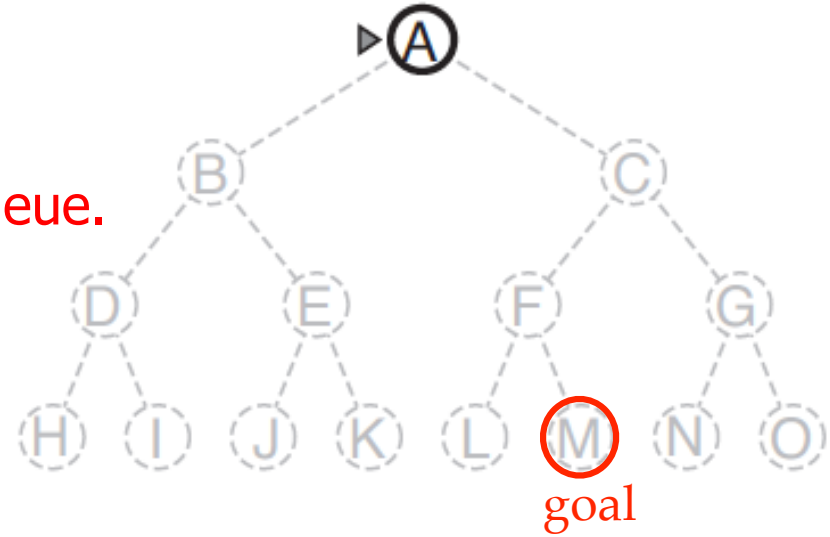
Depth-First Search

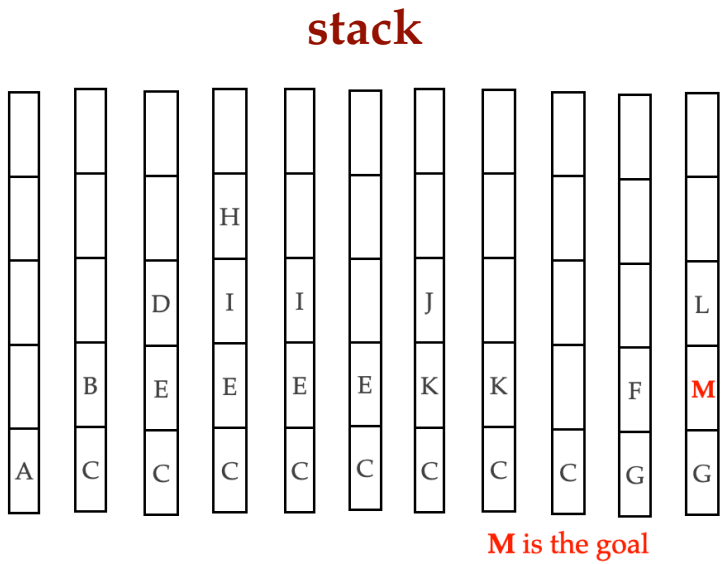
- Expand **deepest unexpanded** node
- **Frontier = Last In First Out (LIFO)** queue, i.e., new successors go at the **front** of the queue
- **Goal-Test when inserted**

Usually use "recursive" or "stack" to implement.

Initial state = A
Is A a goal state?

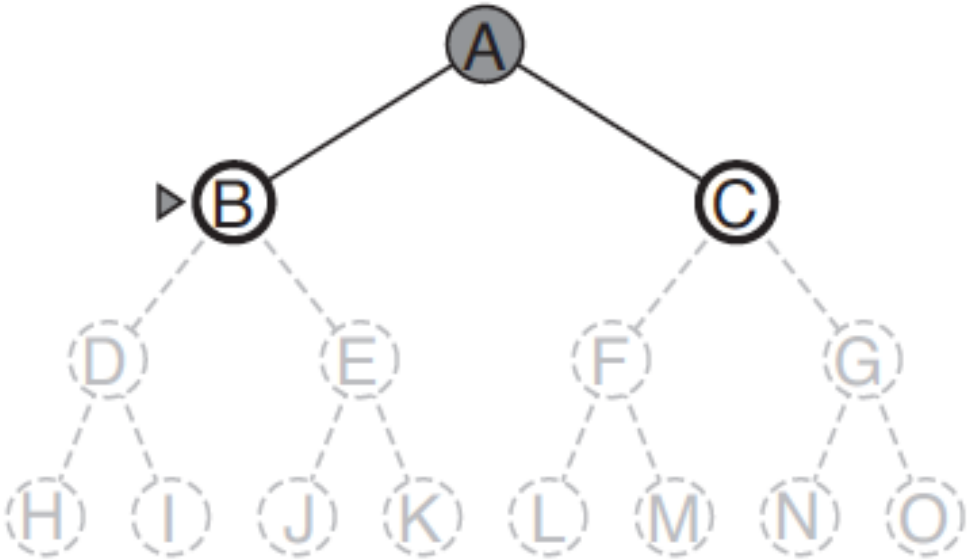
Put A at front of queue.
frontier = [A]





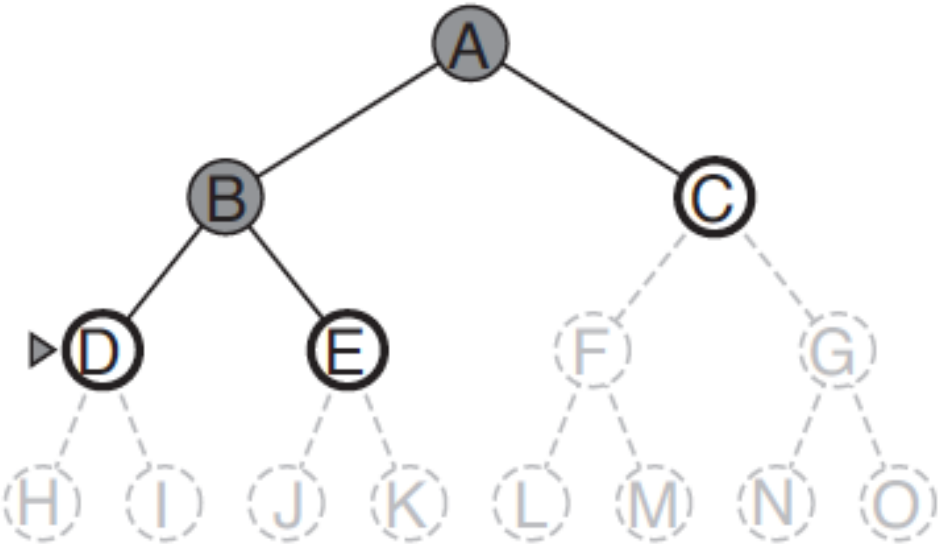
Expand A to B, C.
Is B or C a goal state?

Put B, C at front of queue.
frontier = [B,C]



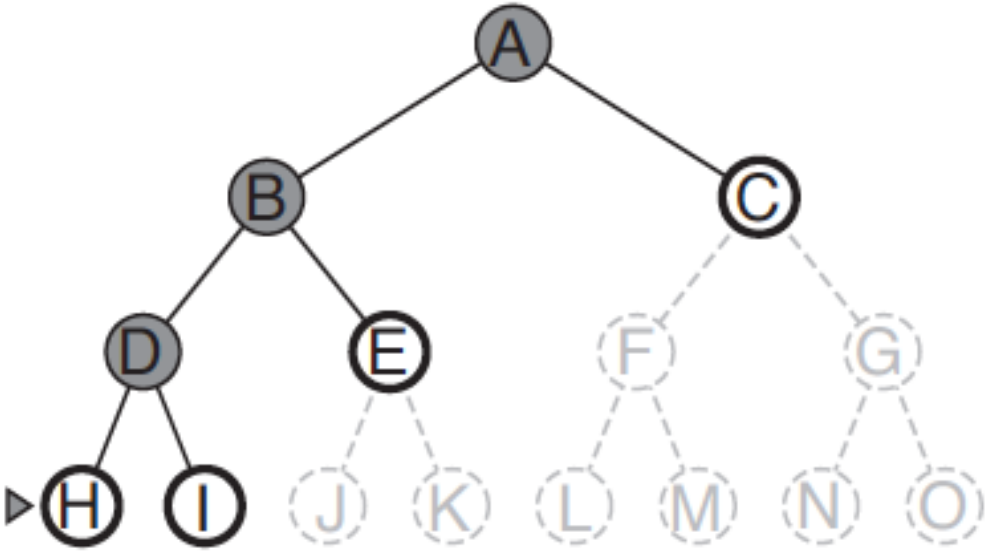
Expand B to D, E.
Is D or E a goal state?

Put D, E at front of queue.
frontier = [D,E,C]



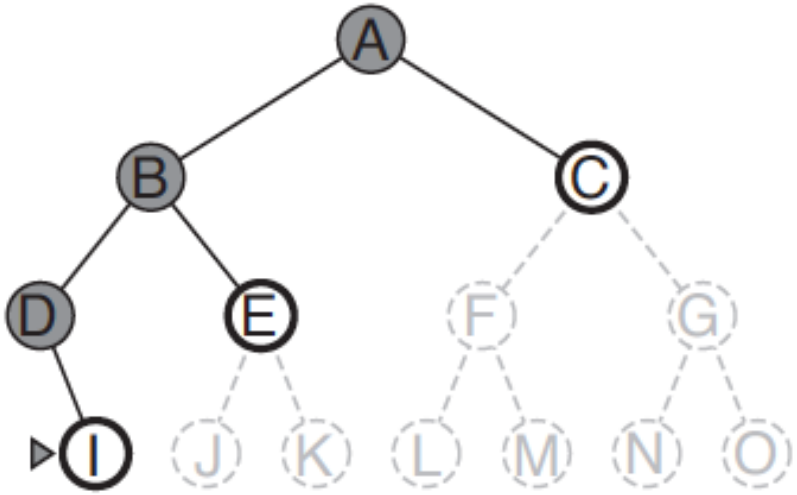
Expand D to H, I.
Is H or I a goal state?

Put H, I at front of queue.
frontier = [H,I,E,C]



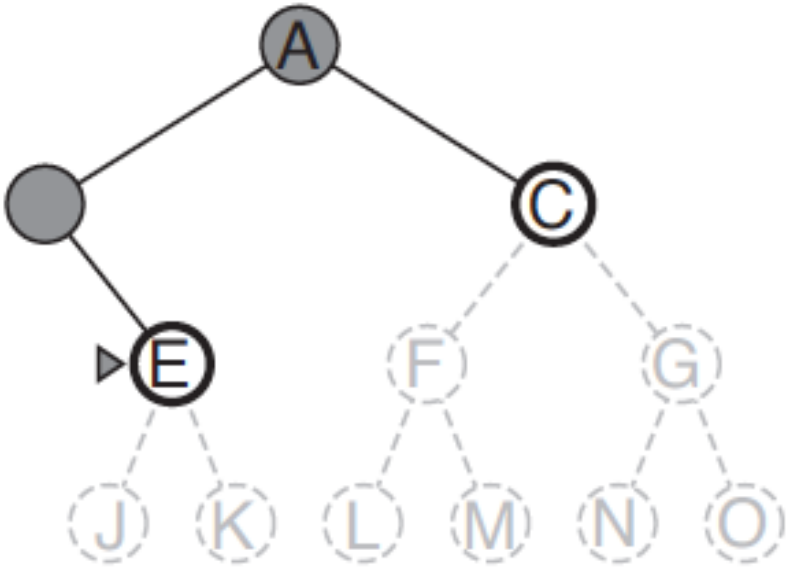
Expand H to no children.
Forget H.

frontier = [I,E,C]



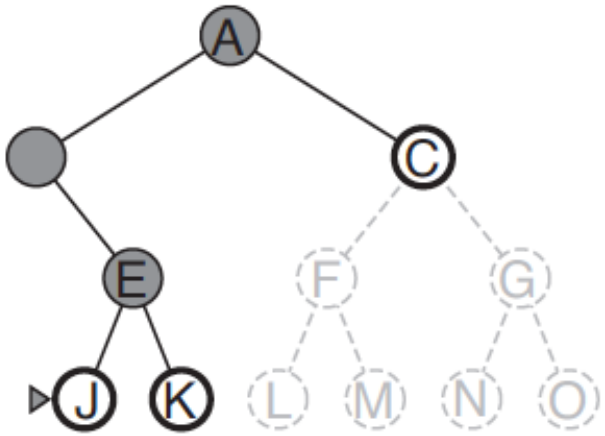
Expand I to no children.
Forget D, I.

frontier = [E,C]



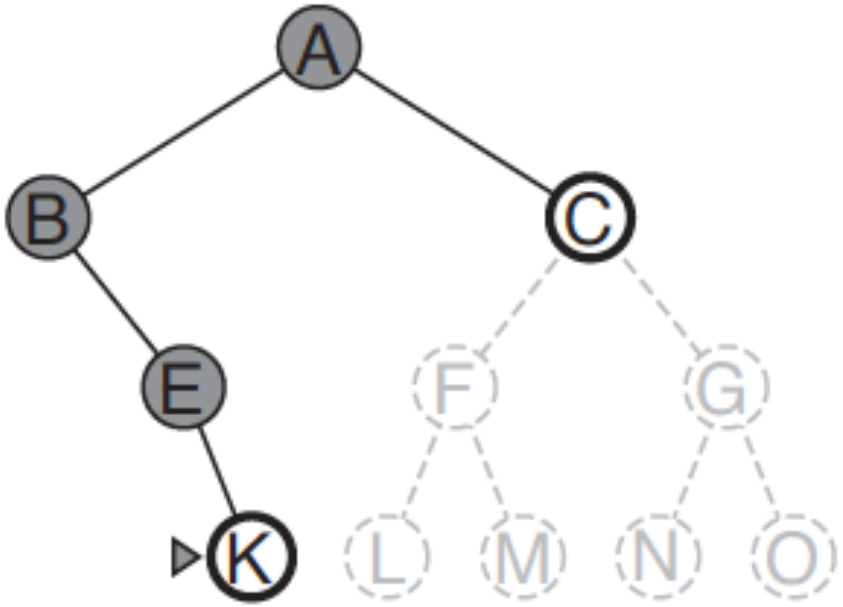
Expand E to J, K.
Is J or K a goal state?

Put J, K at front of queue.
frontier = [J,K,C]



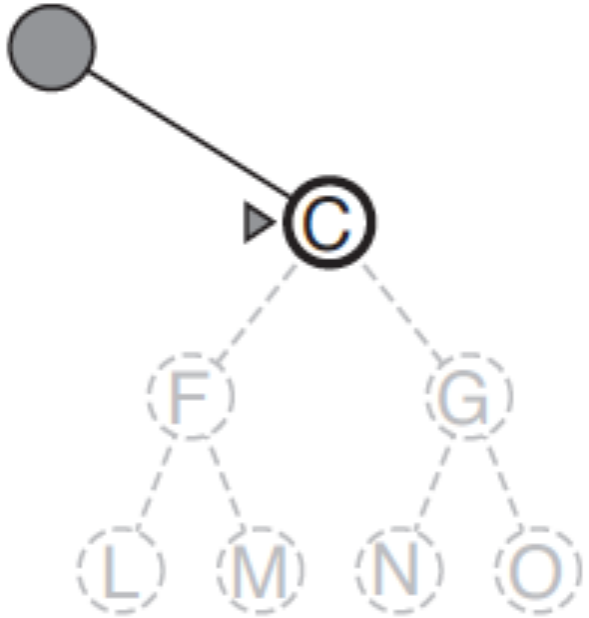
Expand J to no children.
Forget J.

frontier = [K,C]



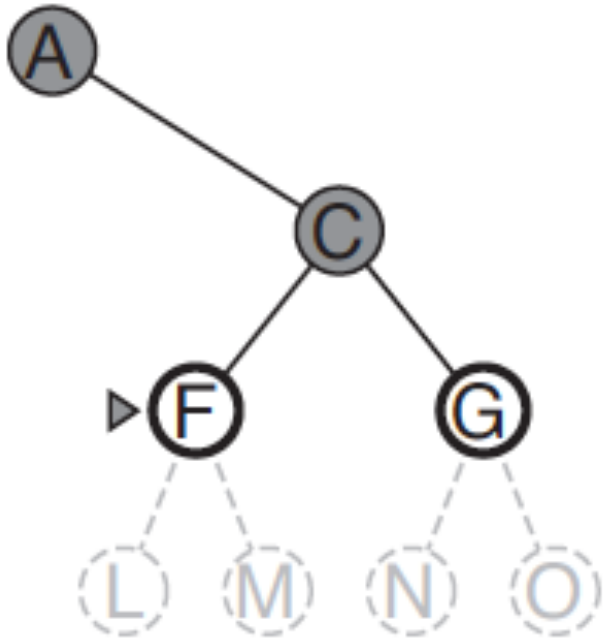
Expand K to no children.
Forget B, E, K.

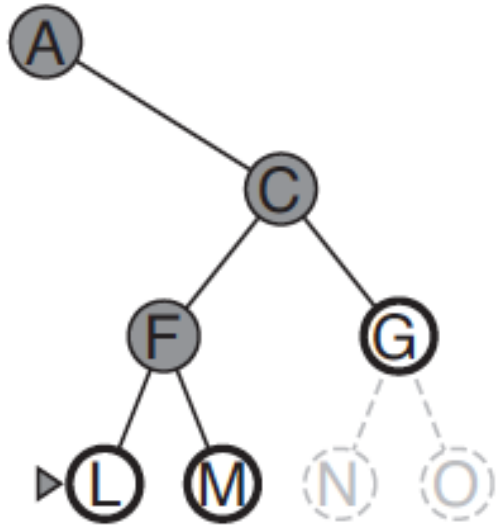
frontier = [C]

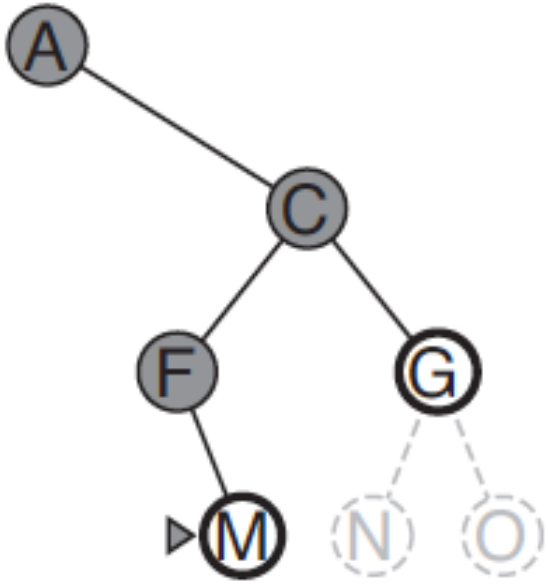


Expand C to F, G.
Is F or G a goal state?

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

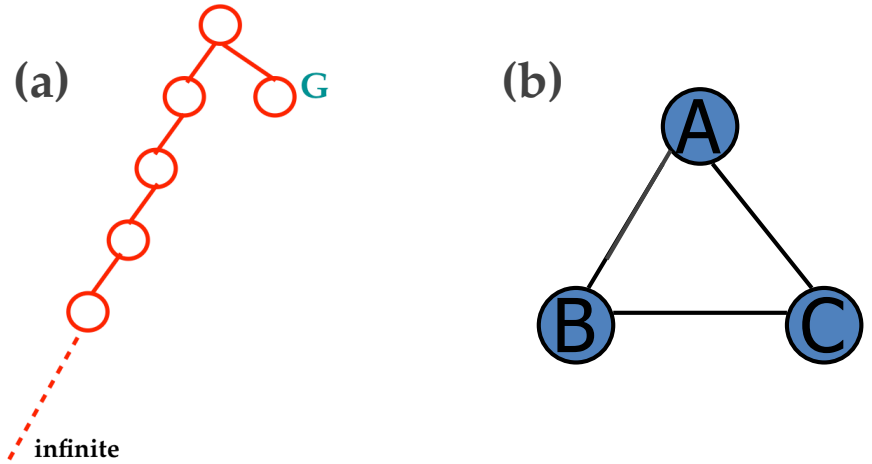






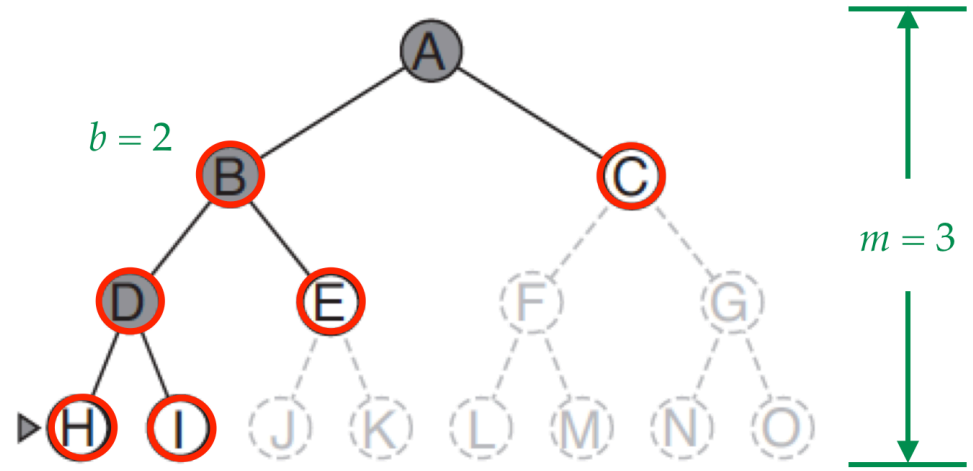
M is the only goal node.

- **Properties of DFS**
  - **Completeness**
    - **No**, fails in
      - (a) **infinite-depth** spaces (may miss goal entirely)
      - (b) spaces with **loops**
    - (b) can be solved by **modifying to avoid loops / repeated states** along path
      - Check **if current nodes occurred** before on path to root
      - Can use **graph search** (remember all nodes ever seen)
        - **Problem with graph search**: space is exponential, not linear



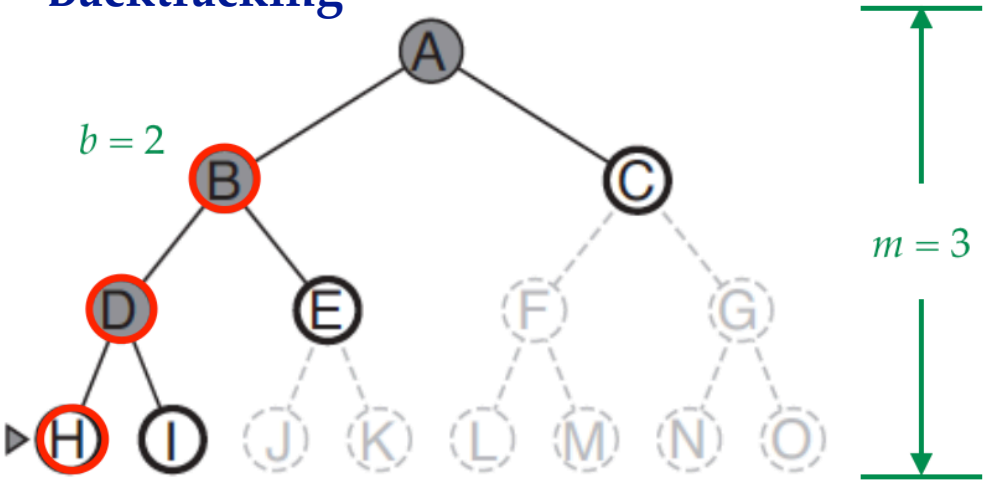
- **Optimality: No**. It may find a non-optimal goal first
- **Time complexity**
  - $O(b^m)$ with m = max depth of space
    - **Terrible** if m is **much larger** than d (m may be infinite)
    - If solutions are **dense**, may be **much faster** than BFS

- **Space complexity**
  - ***O(bm)*, linear space!** [only have to remember the nodes in frontier]
  - **Backtracking** technique only generate **one successor** instead of all successors
    → space complexity is reduced to ***O(m)***
    (remember only a **single path + expanded unexplored nodes**)
    → Backtracking should trade off **higher time complexity**



frontier = {C, E, H, I} $bm = 6$

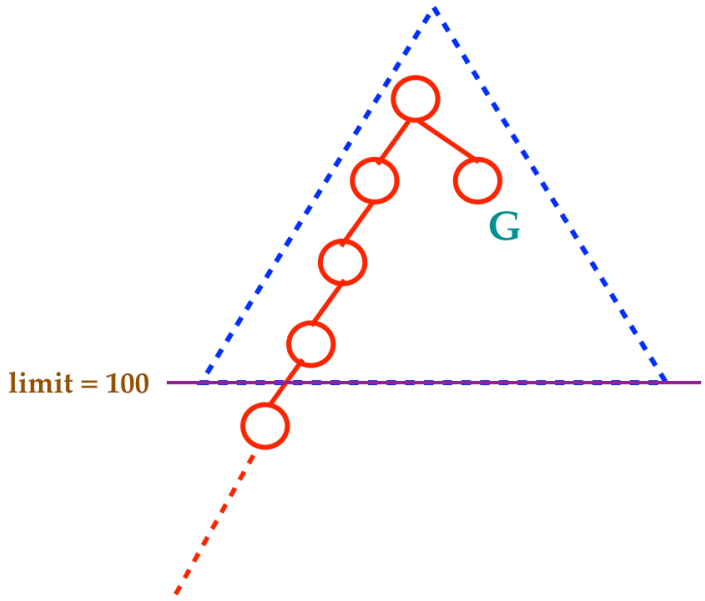
Backtracking



frontier = {B, D, H} $m = 3$

Depth-Limited Search (DLS)

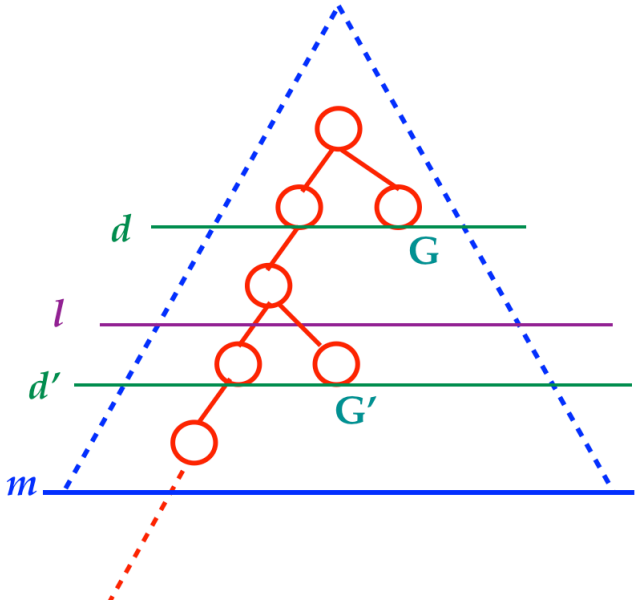
- DLS combines the advantages of BFS (better time complexity) and DFS (better space complexity)
- DFS never terminates if m $\rightarrow \infty$.
- DLS = DFS with **depth limit *l***
- Nodes at depth *l* have no successors
- **Recursive** implementation



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

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_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
```

- **Properties of DLS**
  - **Completeness**:
    - **Not complete** if $l < d$ (G′)
    - **Complete** otherwise (G)



  - **Optimality**: **Not optimal** in general (even if $l > d$)
  - **Time complexity**: $O(b^l)$
  - **Space complexity**: $O(bl)$, **linear** space
  - Terminate with two kinds of failure
    - **Failure**: no solution
    - **Cutoff**: no solution within the depth limit

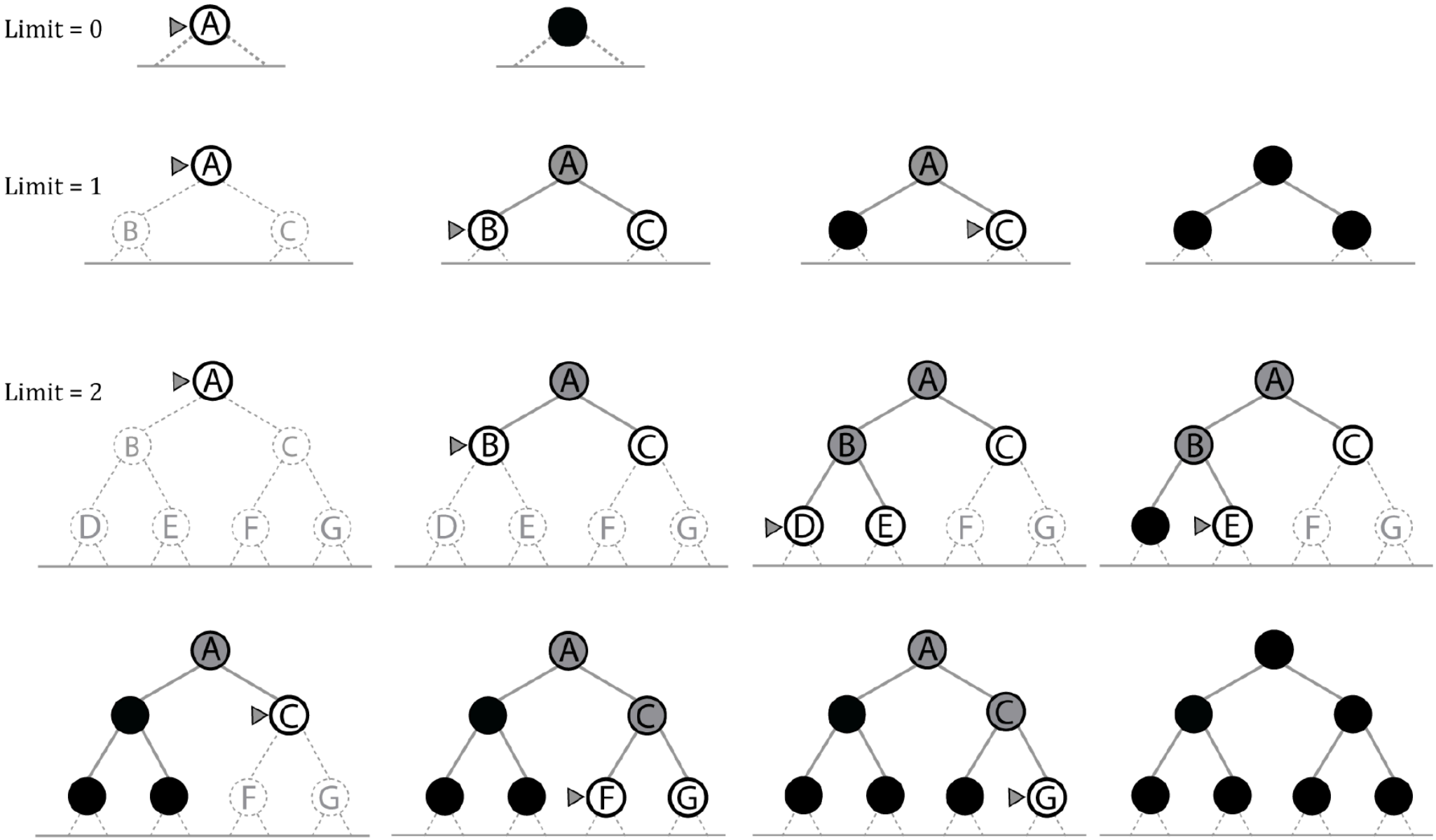
- DFS
  - Time complexity: $O(b^m)$
  - Space complexity: $O(bm)$

Iterative-Deepening Search (IDS)

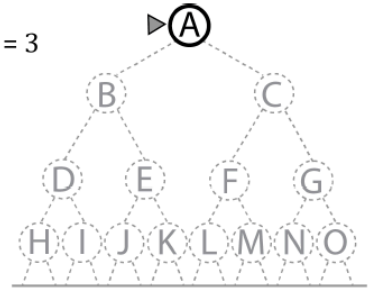
- IDS is an improvement of DLS
- Combine the benefits of BFS and DFS
- Call DLS **iteratively** with **increasing depth limit**
- Use DFS as a subroutine
  - Check the root
  - Do a DFS searching for a path of length 1
  - If there is no path of length 1, do a DFS searching for a path of length 2
  - If there is no path of length 2, do a DFS searching for a path of length 3…

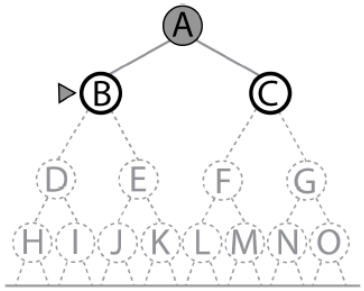
- IDS terminates when a **solution is found** or if the **depth limited search returns failure**, meaning that **no solution** exists

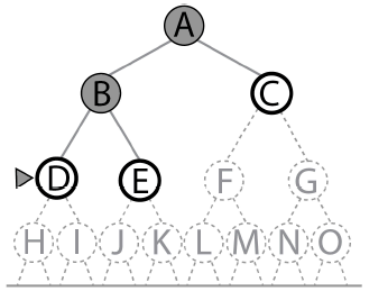
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

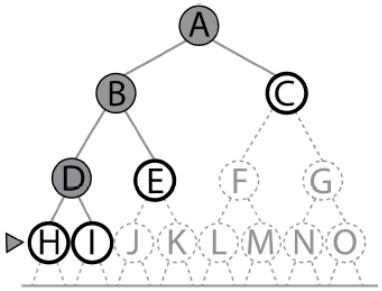

Limit = 0
A
Limit = 1
A
B
C
A
B
C
A
C
Limit = 2
A
B
C
D
E
F
G
A
B
C
D
E
F
G
A
B
C
D
E
F
G
A
B
C
E
F
G
A
C
F
G
A
C
F
G
A
C
G

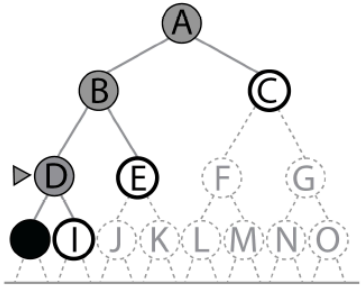
Limit = 3

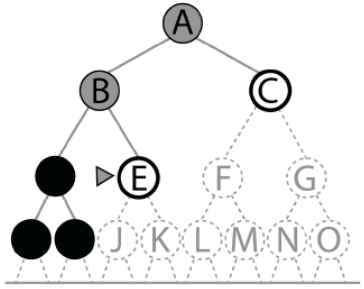

A
B C
D E F G
H I J K L M N O

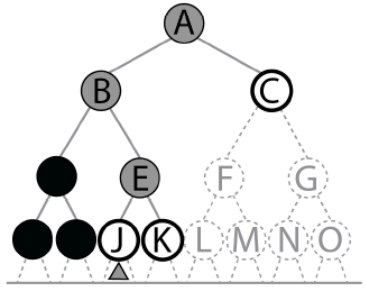

A
B C
D E F G
H I J K L M N O

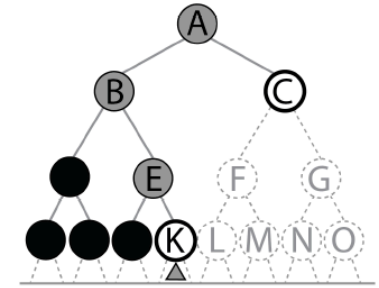

A
B C
D E F G
H I J K L M N O

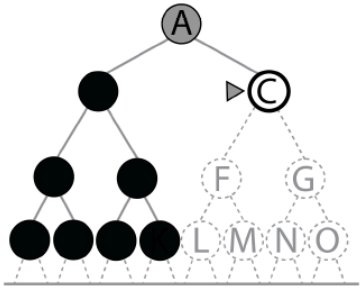

A
B C
D E F G
H I J K L M N O

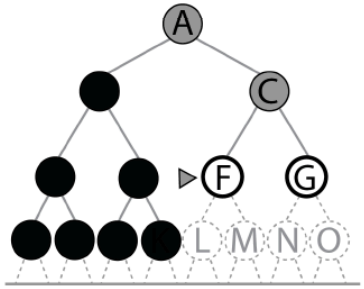

A
B C
D E F G
I J K L M N O

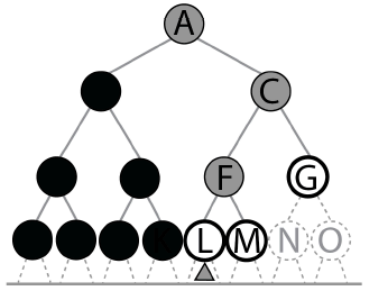

A
B C
E F G
J K L M N O

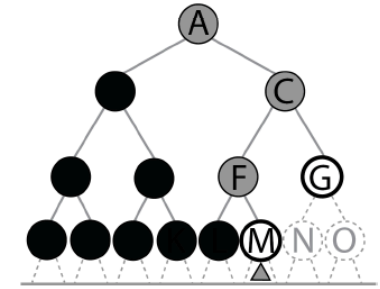

A
B C
E F G
J K L M N O


A
B C
E F G
K L M N O


A
C
F G
L M N O


A
C
F G
L M N O


A
C
F G
L M N O


A
C
F G
M N O

- **Properties of Iterative Deepening Search**
  - **Completeness**: **Yes** $[l = 0 \text{ to } \infty]$
  - **Optimality**
    - **No**, for **general cost** functions
    - **Yes**, only if **step cost = 1**
      - Can be modified to explore **uniform-cost tree** (optimal)
      - e.g. Firstly search the nodes with path cost $\leqq 10$

- **Time complexity**
  - The nodes on the **bottom level** (**depth** d) are generated **once** — $(1)b^d$
  - Those on the **next-to-bottom level** are generated **twice**, and so on — $(2)b^{(d-1)}$
  - The children of the **root**, which are generated d times. — $(d)b^1$
  - Total: $O(b^d)$ *is* asymptotically the same as BFS

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

- **Space complexity**: $O(bd)$

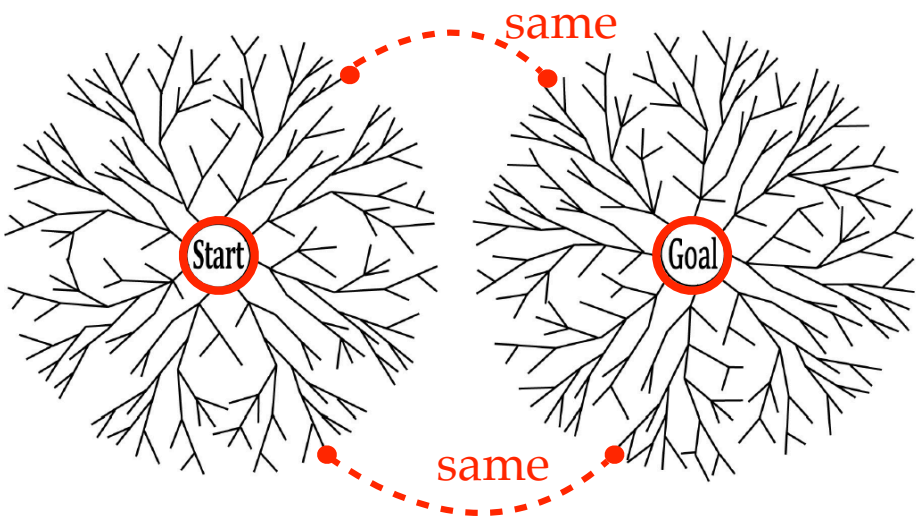
- Numerical comparison (time complexity) for $b = 10$, $d = 5$, solution at **far right leaf**
  - $N(IDS) = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 =$ **123,450** $(d)b^1 + (d-1)b^2 + \ldots + (2)b^{(d-1)} + (1)b^d = O(b^d)$
  - $N(BFS) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 =$ **111,100** $b + b^2 + \ldots + b^d = O(b^d)$

$$\frac{N(IDS)}{N(BFS)} \approx \frac{b+1}{b}$$

- Repeated search in IDS is **not severe**
- **IDS is preferred when search space is large and depth is unknown**
- More advantages of IDS in **adversarial search**

Bidirectional Search

- **Idea**
  - Simultaneously search **forward** from *S* and **backward** from *G*
  - **Stop** when both "**meet in the middle**"
  - Need to keep track of the **intersection** of **2 open sets** of nodes



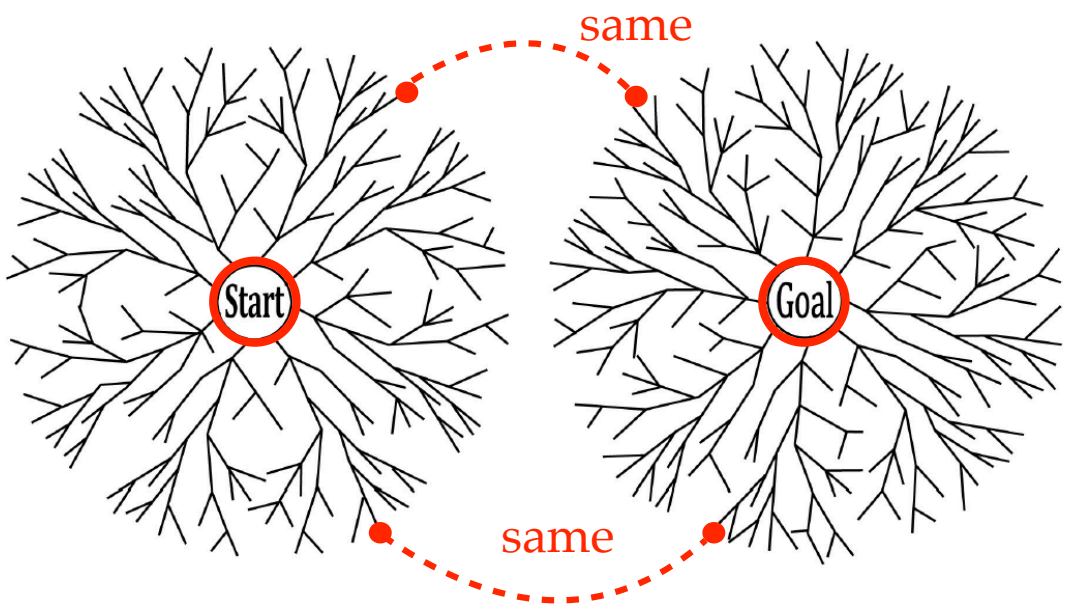
$2 * b^{d/2}$

- **Time complexity**: reduce from $O(b^d)$ to $O(b^{d/2})$
  - Though the reduction is attractive, **how to search backward**?
  - Need a way to specify the **predecessors** of G
    - This can be difficult
      - e.g., predecessors of checkmate in chess?
  - Which to take if there are **multiple goal states**?
    - 8-puzzle：only one goal
    - 8-queen puzzle：many goals (Which are the predecessors?)

Problem:

e.g. Robot falls down and has no way to climb up

- **Space complexity**: increase from $O(bd)$ to $O(b^{d/2})$ [the space spent in saving the **matching** of the states from two ends - space complexity] as well, can be problematic



Succeed when a branch from the start node **meets** a branch from the goal node.

Comparison of Uninformed Search Strategies

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

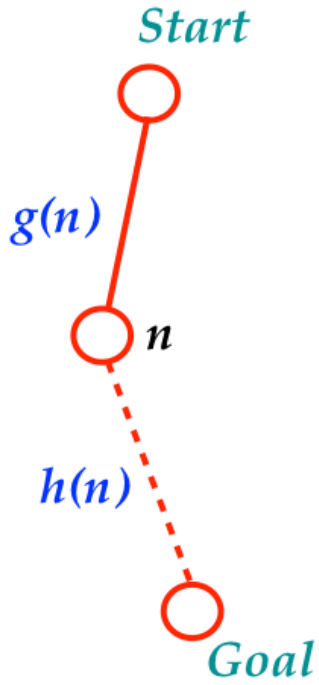
- ***b***: the branching factor
- ***d***: the depth of the shallowest solution
- ***m***: the max depth of the search tree
- ***l***: the depth limit

- ***Superscript***
  - *a* : complete if ***b*** is finite
  - *b* : complete if step costs ≥ ε for positive ε
  - *c* : optimal if step costs are all identical
  - *d* : if both directions use BFS

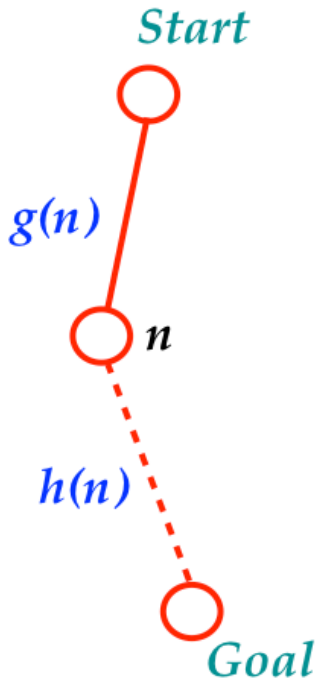
3.5 Informed (Heuristic) Search Strategies

- **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
- **Heuristic**
  - **Definition**: a **rule of thumb** (or set of rules, e.g. straight-line distance) intended to increase the probability of solving some problem

- **Heuristic function**
  - ***h*(*n*)**: estimates the (optimal) **remaining cost** of reaching goal from node ***n***
  - Defined using only the **state** of node ***n***
  - ***h*(*n*) = *0*** if ***n*** is a **goal** node



- **Best-First Search**
  - Greedy search ($h(n)$)
  - A* search ($g(n) + h(n)$)
  - Optimality of A*
- **Memory Bounded Search**
  - Iterative deepening A*
  - Recursive best-first search
  - Simplified memory-bounded A*
- **Heuristic**
  - Performance
  - Generating heuristics



Best-First Search

- **Informed** search, a.k.a. **heuristic** search
- **Idea**: Use an **evaluation function** for each node to estimate the **desirability**
  - Expand the most desirable unexpanded node
- The **evaluation function** is called **heuristic**, denoted as $h(n)$
  - It estimates the cost from node n to the **closest (best) goal**

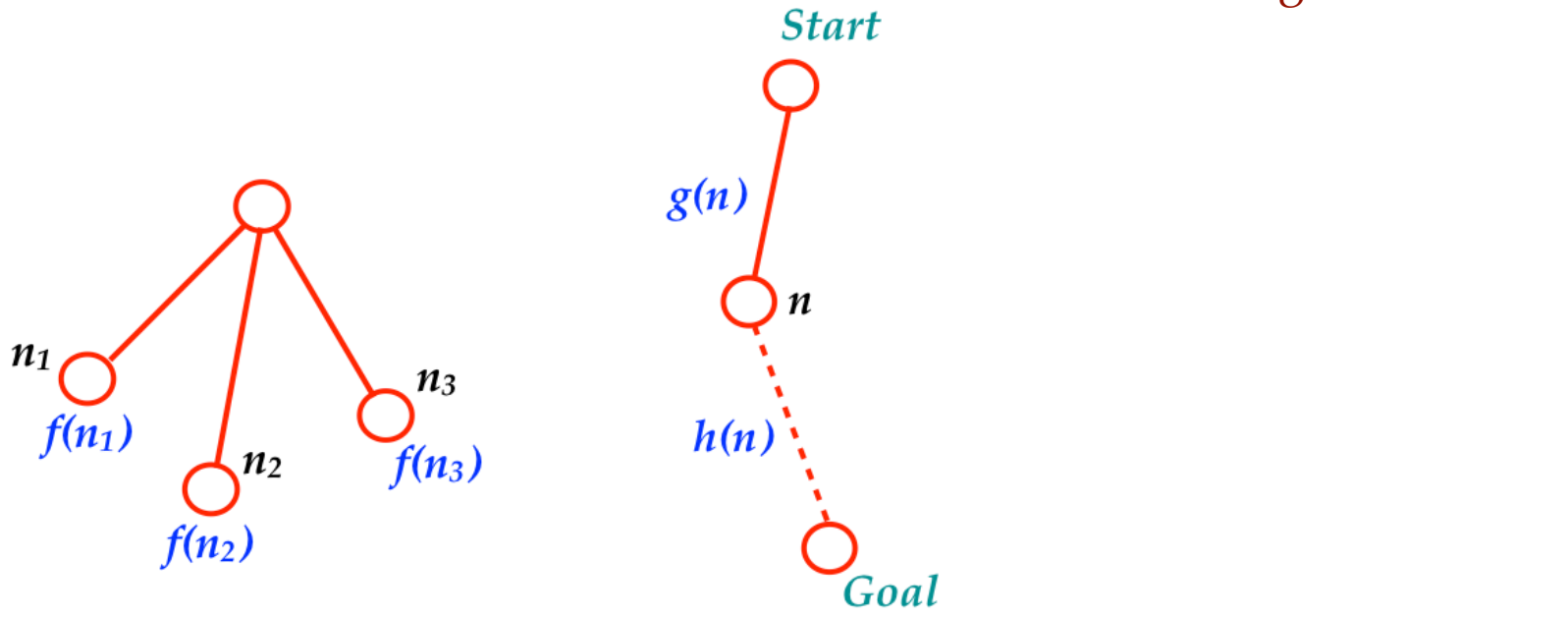
- Special cases
  - Greedy search, $f(n) = h(n)$
  - Uniform Cost Search, $f(n) = g(n)$
  - A* search, $f(n) = g(n) + h(n)$

path cost

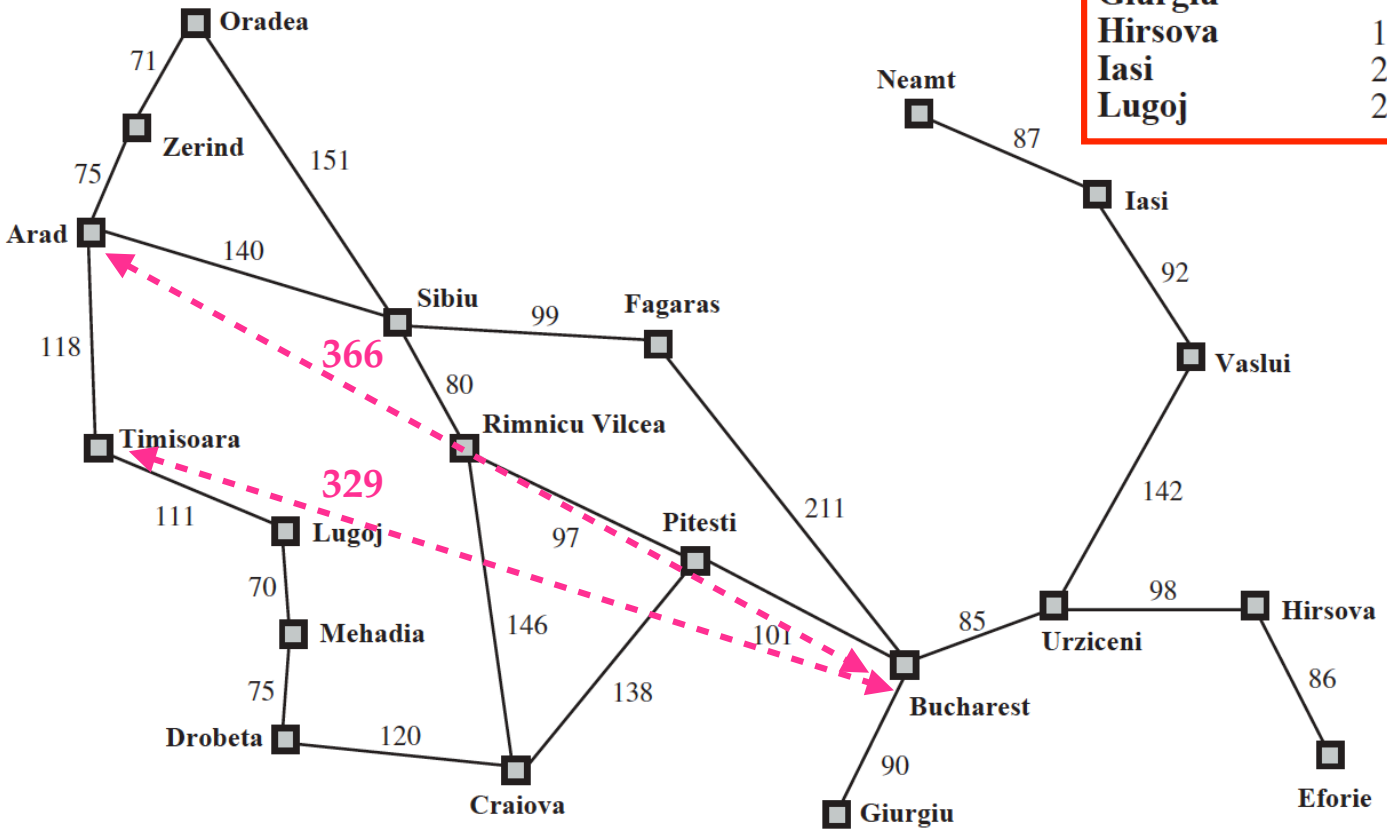
A* Search ≈ Unifrom Cost Search [$g(n)$] + Greedy Search [$h(n)$]

may be optimal

well-designed $h(n)$ may be sooner to reach the goal



Heuristic ($h(n)$) for the Romania problem



Straight-line distance (h_{SLD}) to Bucharest

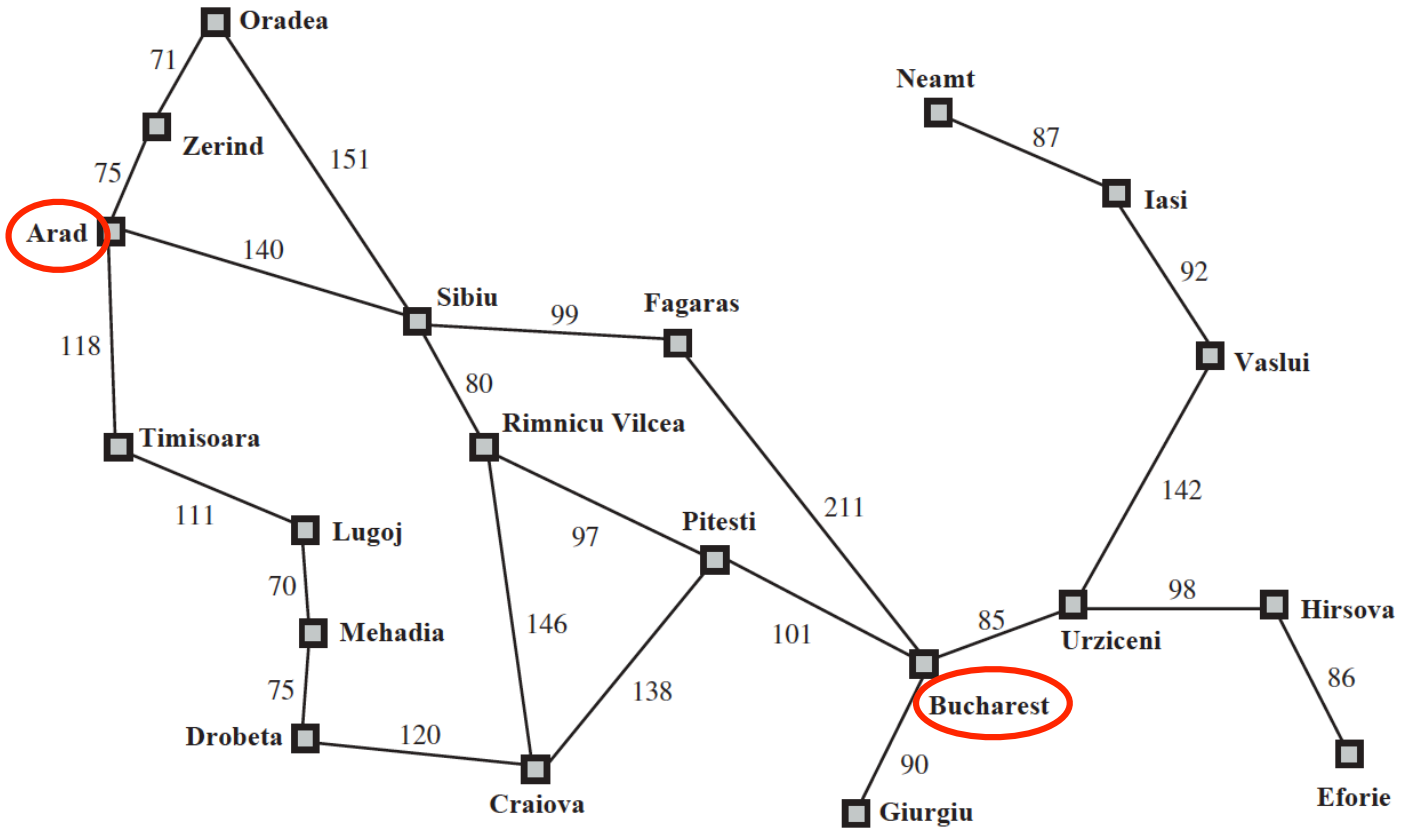
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

$f(n) = h(n)$

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

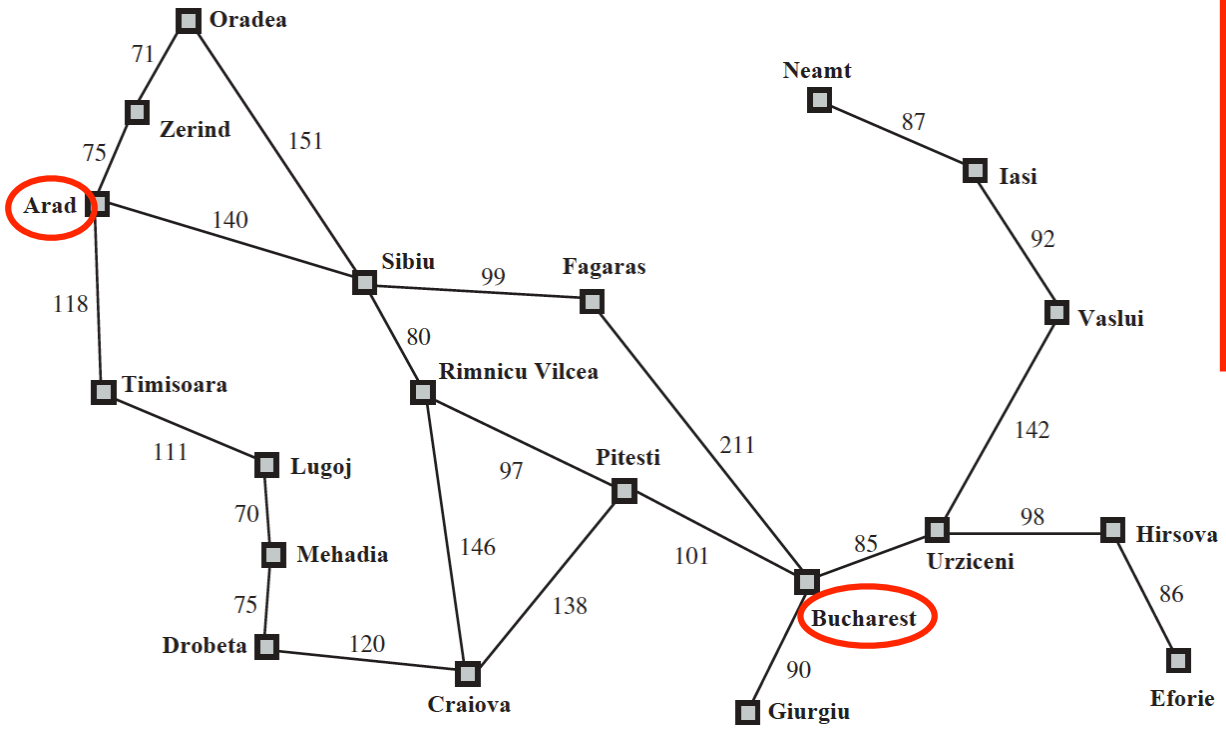
Greedy Best-First Search

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



- Stages in a greedy best-first tree search for Bucharest with the **straight-line distance** heuristic h_{SLD}
- Nodes are labeled with their h-values

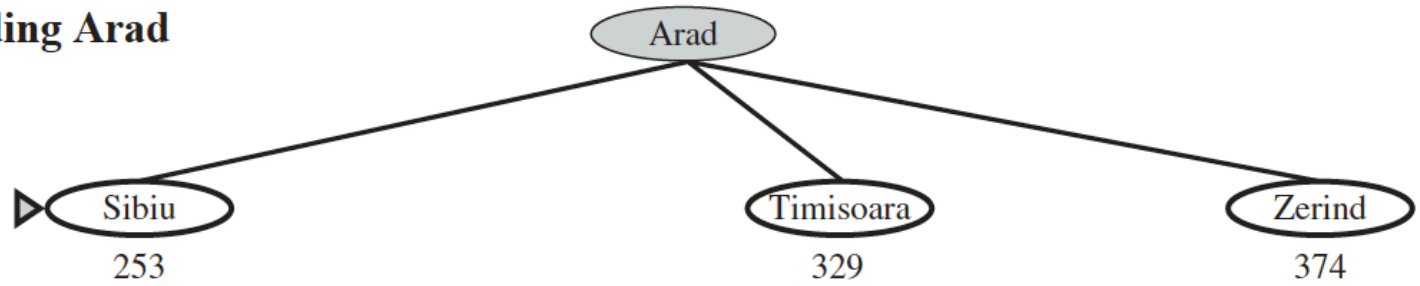
(a) The initial state



Straight-line distance (h_{SLD}) to Bucharest

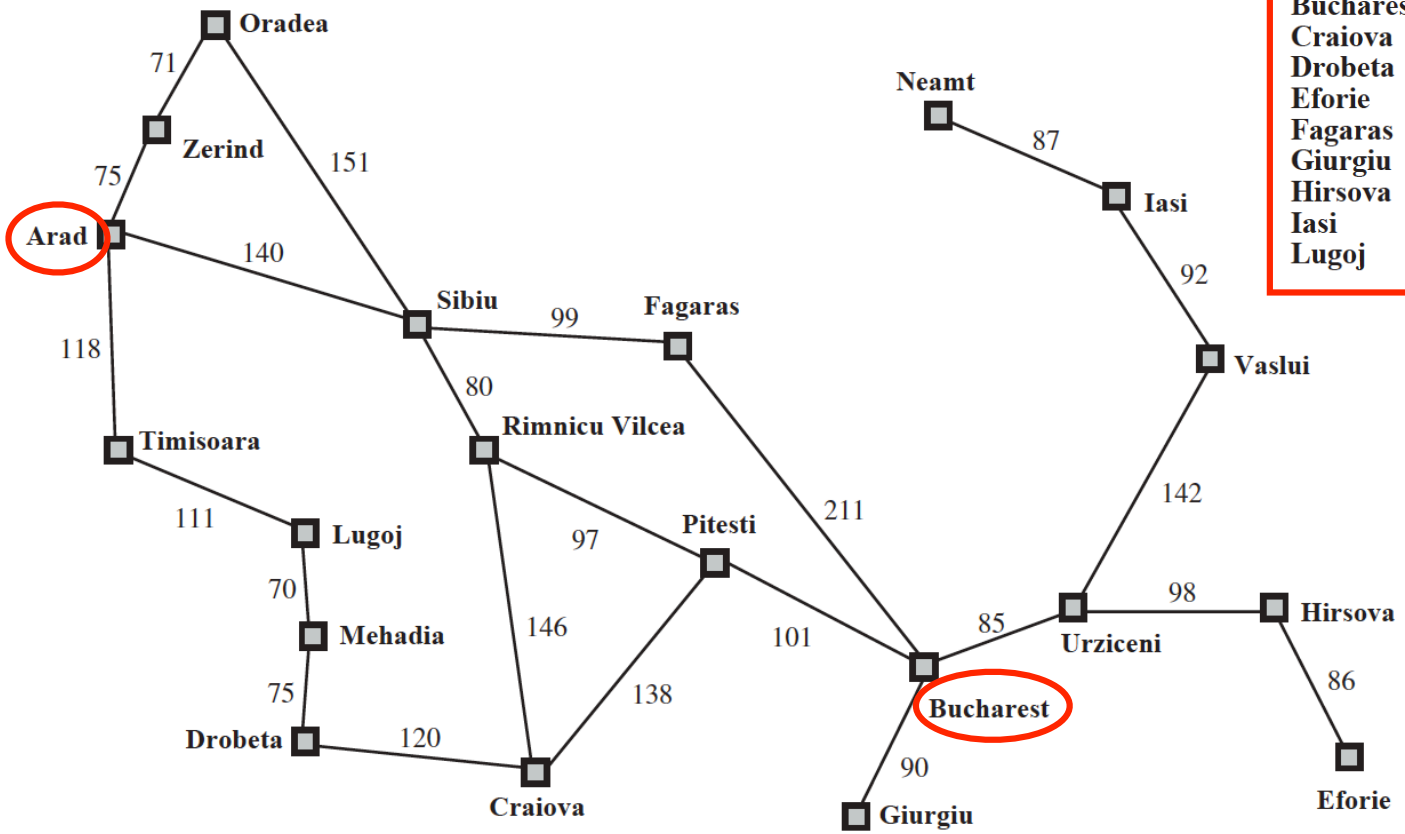
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

(b) After expanding Arad

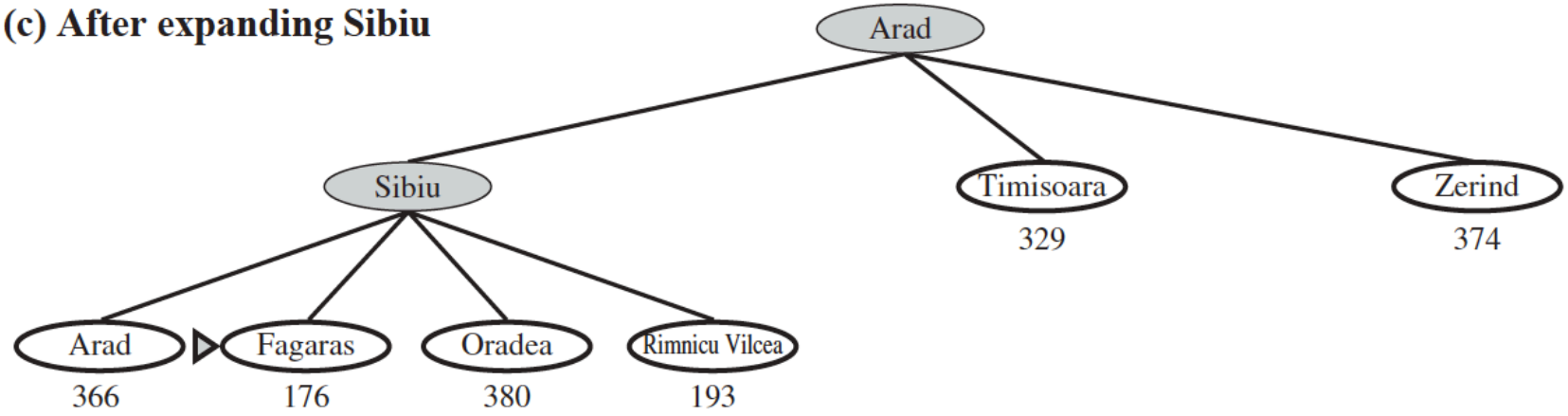


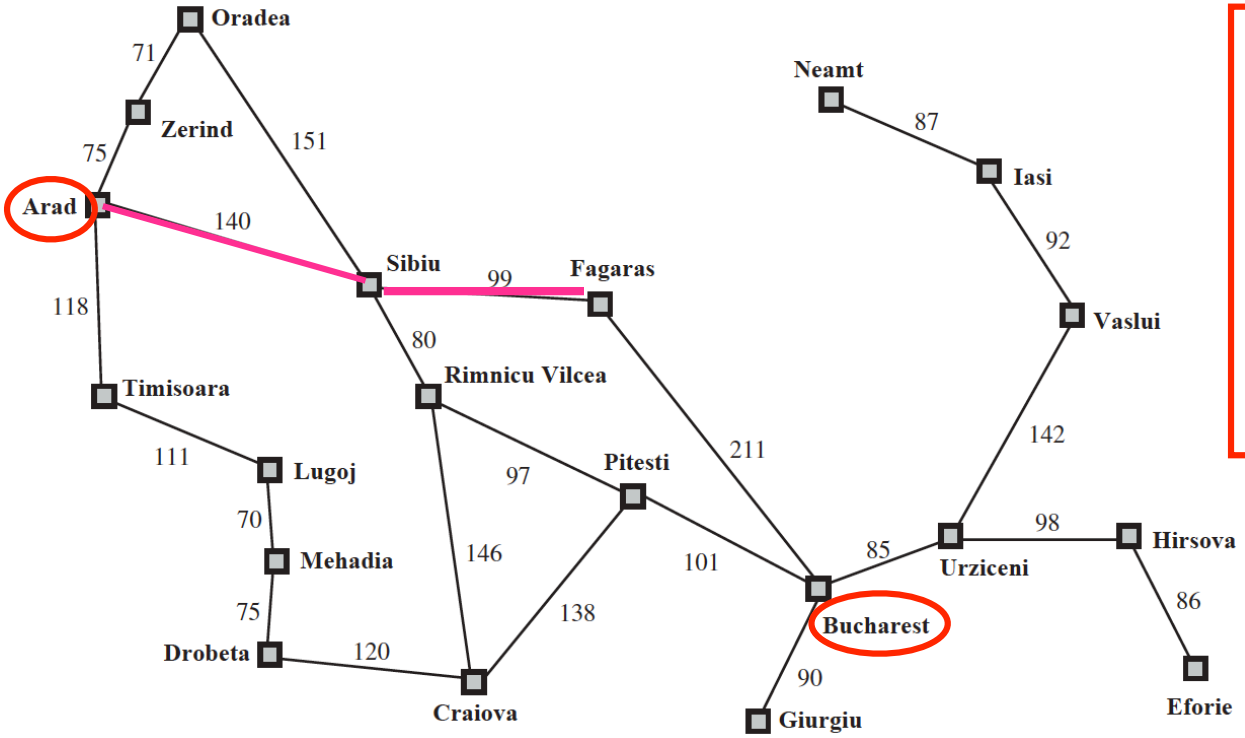
Straight-line distance (h_{SLD}) to Bucharest

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



(c) After expanding Sibiu

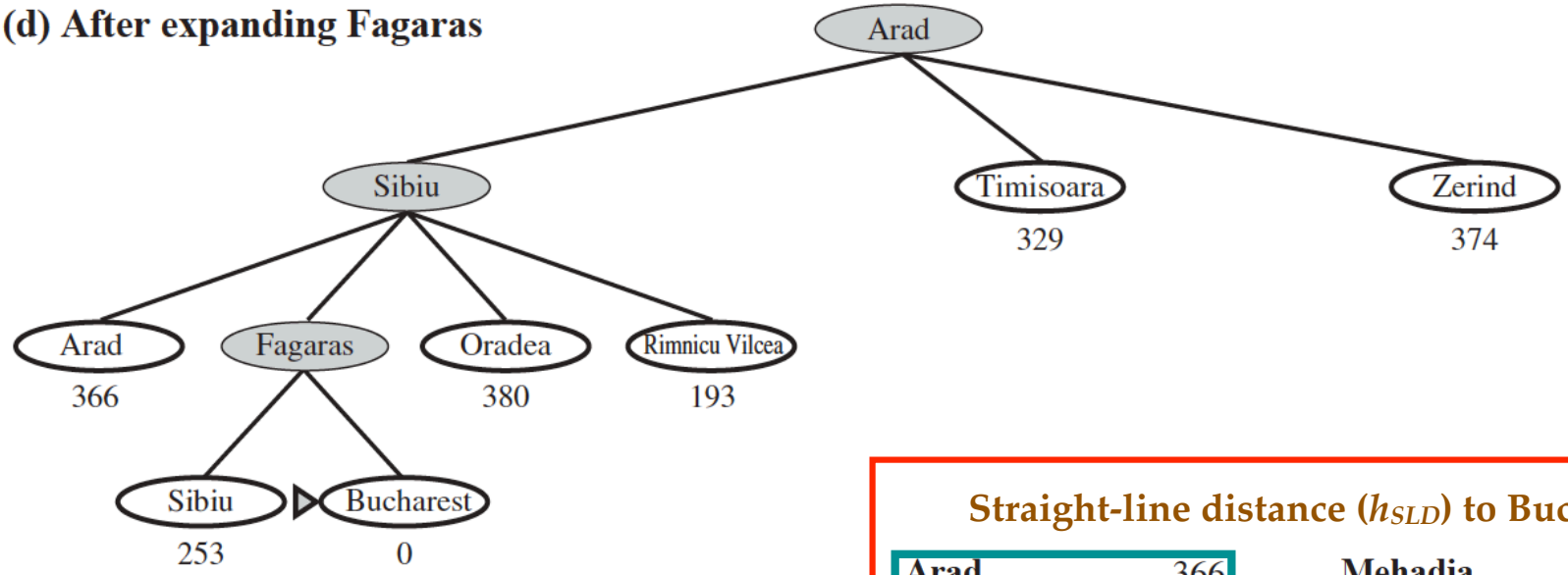




Straight-line distance (h_{SLD}) to Bucharest

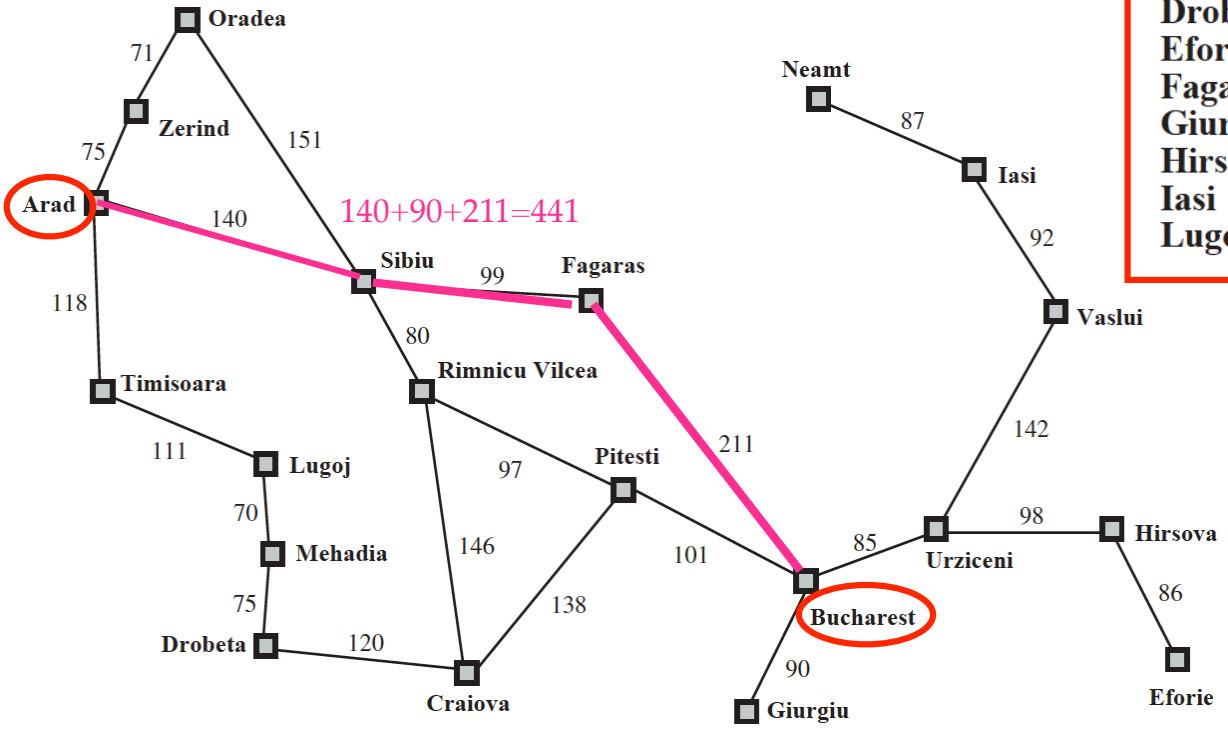
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(d) After expanding Fagaras



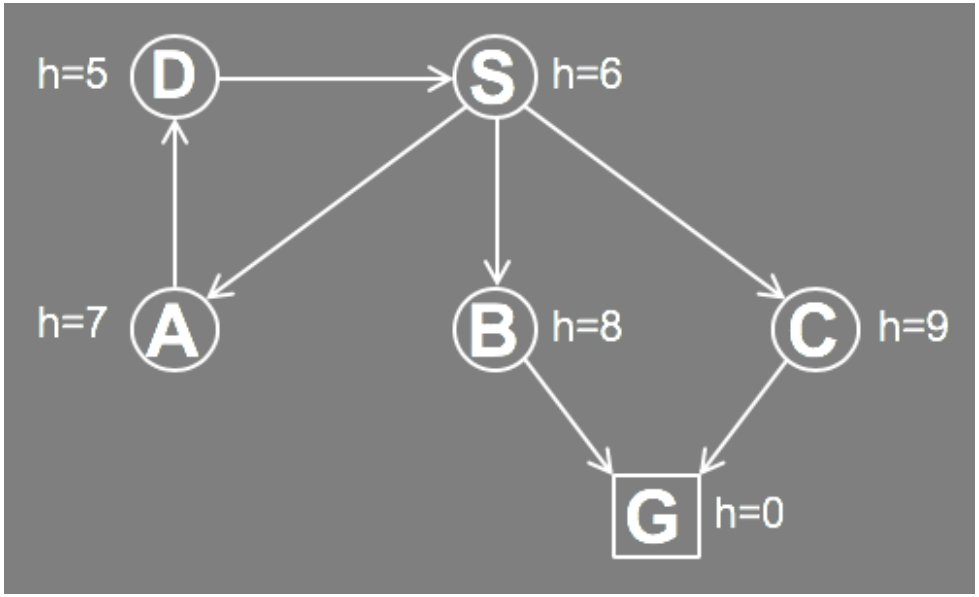
Straight-line distance (h_{SLD}) to Bucharest

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



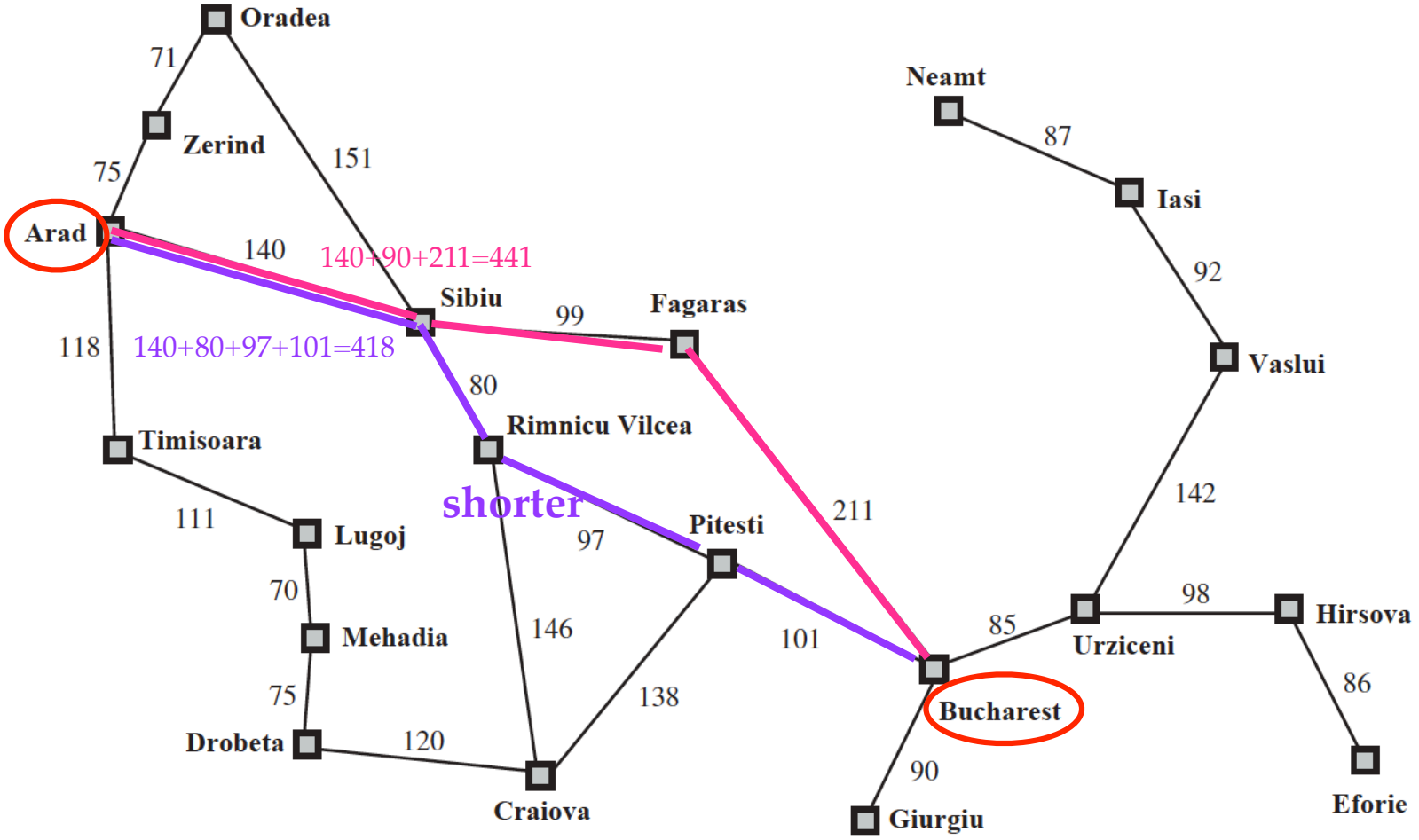
- **Properties of Greedy Search**
  - **Completeness**: No
    - **Tree-Search** (no repeated states check) may **get stuck in loops** and never reach any goal even infinite state spaces

Order of node expansion: S A D S A D S A D. . . .
Path found: none Cost of path found: none



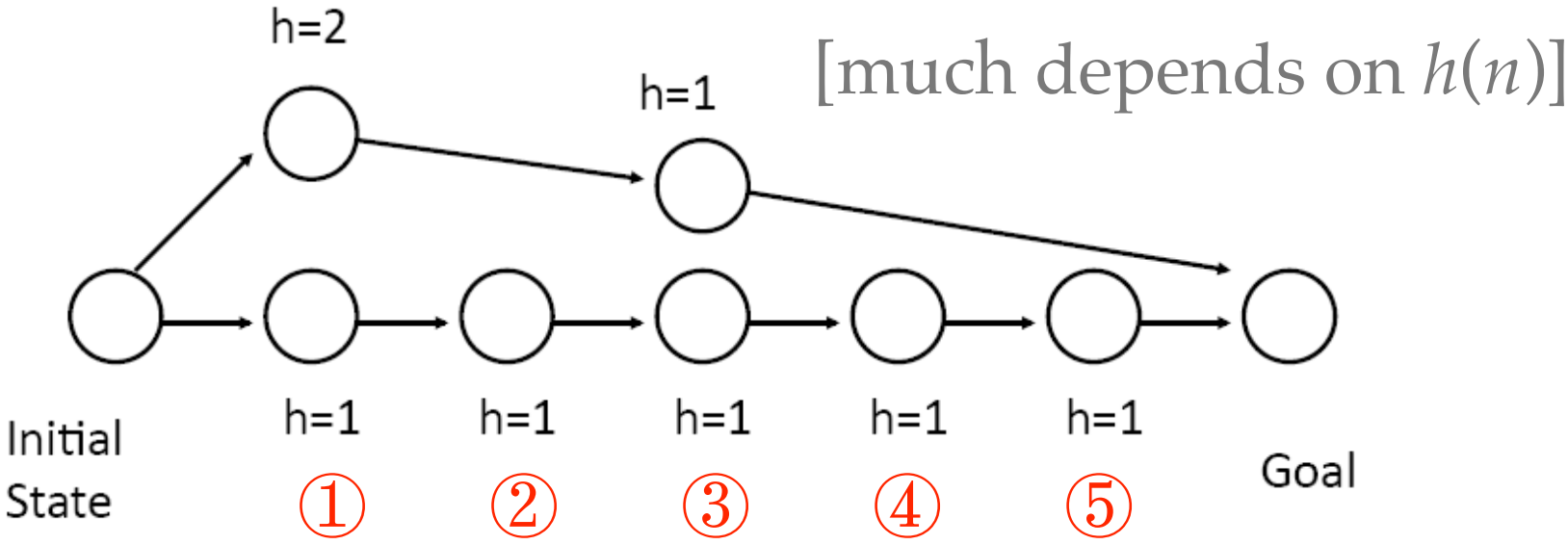
    - **Graph-Search** (check repeated states) is **complete** in **finite spaces**, but **not complete** in **infinite ones**

- **Optimality**: No
  - E.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest is shorter!



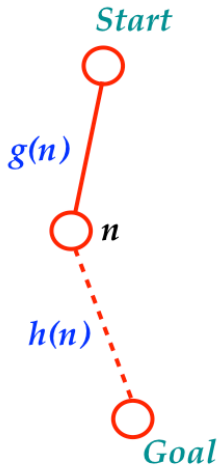
- **Time complexity** [much depends on $h(n)$]
  - Worst case: $O(b^m)$
  - Best case: $O(bd)$ – If $h(n)$ is 100% accurate (cheating?)
  - i.e., time complexity highly depends on the $h(n)$ function
- **Space complexity**: $O(b^m)$, since it keeps all nodes in memory

- How can we fix the greedy problem?



→ A* search helps to fix the greedy problem.

A* Search



- **Objective**: minimizing the **total** estimated solution cost
- **Idea**: Avoid expanding paths that are **already expensive**
- **Generally, A* is the preferred simple heuristic search**
- Evaluation function: $f(n) = g(n) + h(n)$
  - [Uniform Cost] $g(n)$: cost so far to reach n (path cost)
  - [Greedy Best First] $h(n)$: estimated (optimal) cost to goal from node n (heuristic)
  - [A*] $f(n)$: estimated total cost from the starting node to goal through n

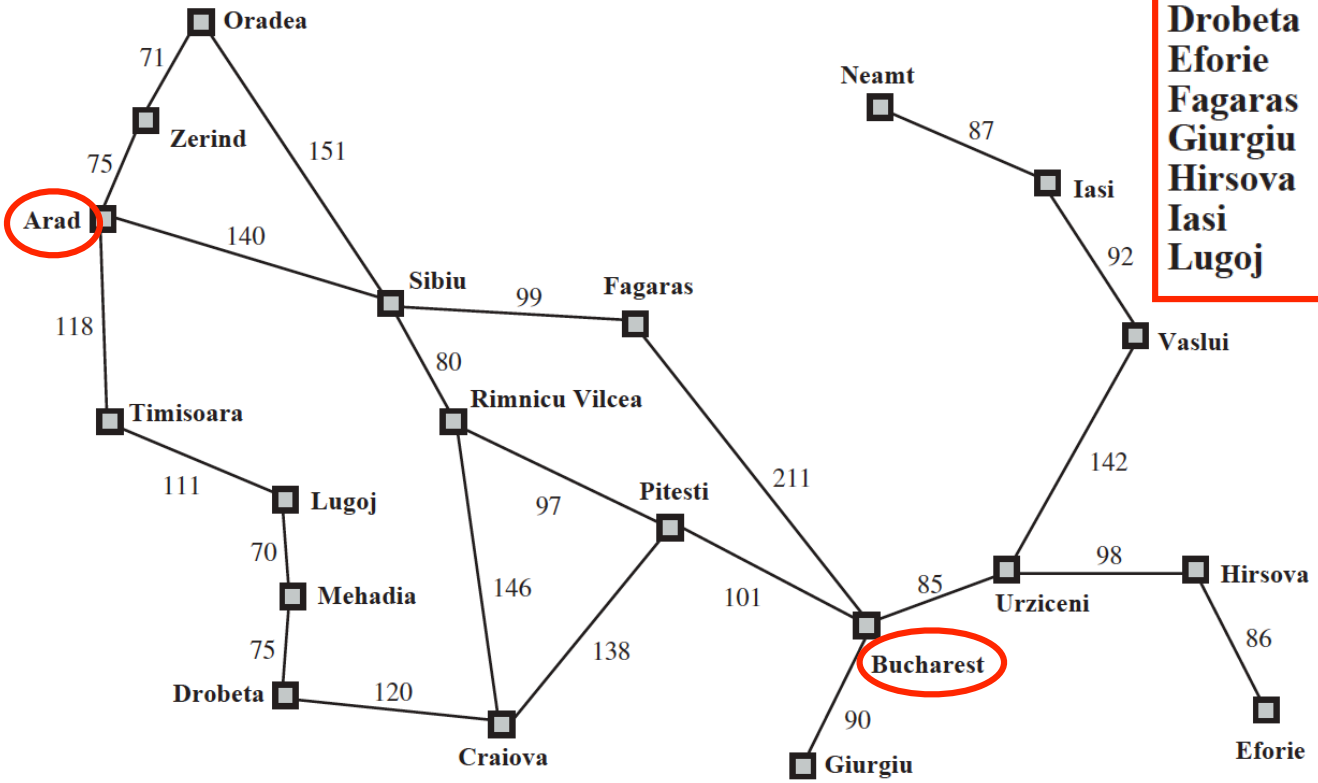
- **A* search** combines the advantages of the **uniform-cost** search and the **greedy** search

  A* Search ≈ Uniform Cost Search [$g(n)$] + Greedy Search [$h(n)$]

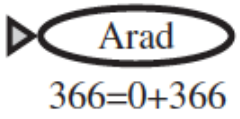
- **Memory-efficient** versions of A* are available
  - Recursive Best-First Search (RBFS)
  - Simplified Memory-Bounded A* (SMA*)

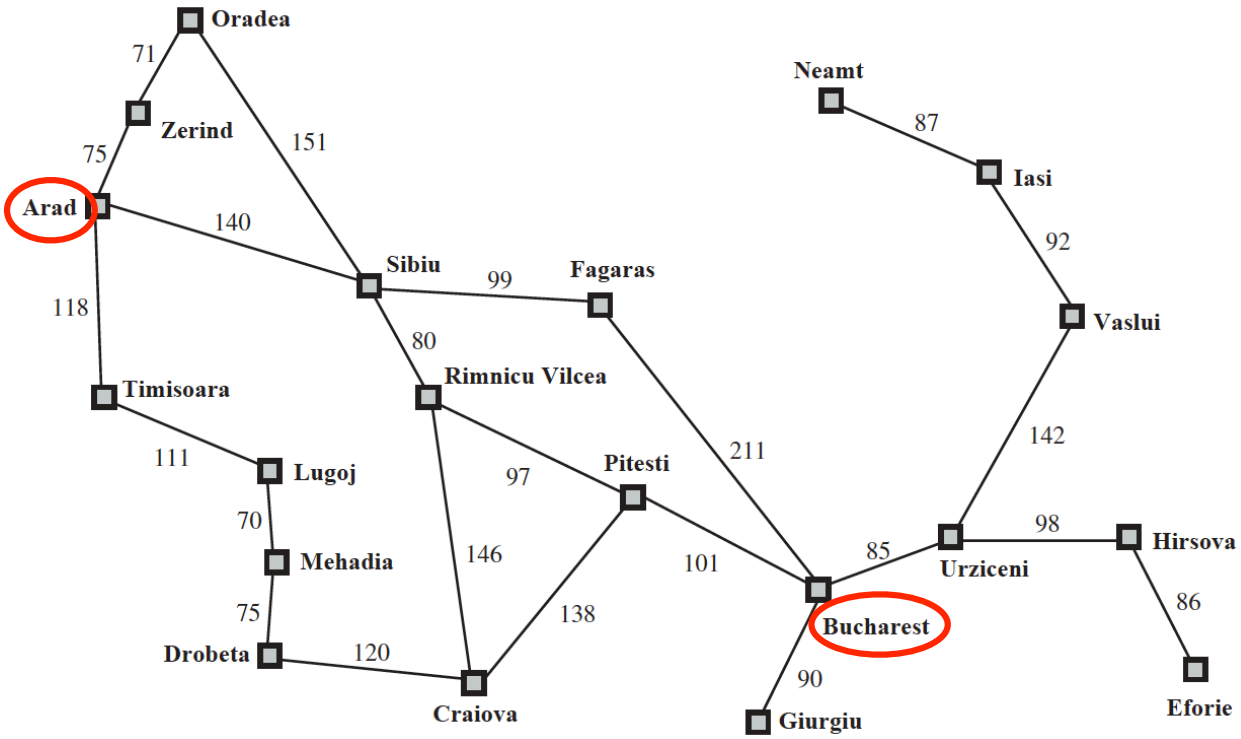
- **Example**
  - Stages in an A* search for Bucharest
  - Nodes are labeled with $f = g + h$
  - The h values are the straight-line distances to Bucharest

Straight-line distance (h_{SLD})			
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) The initial state

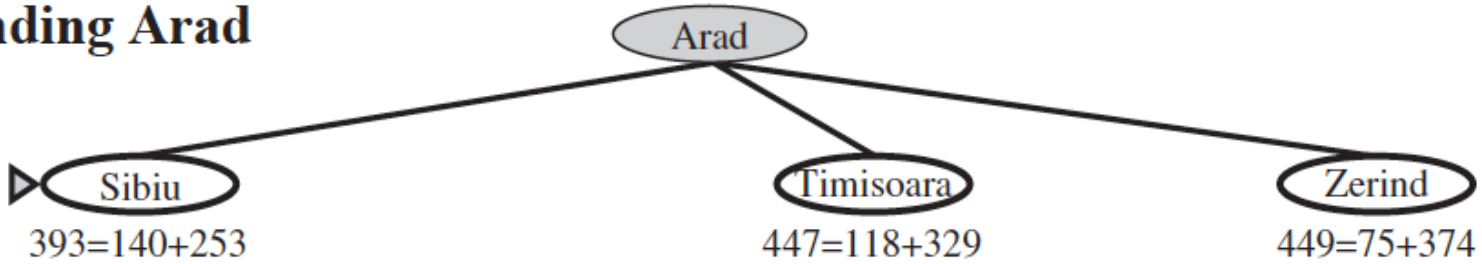


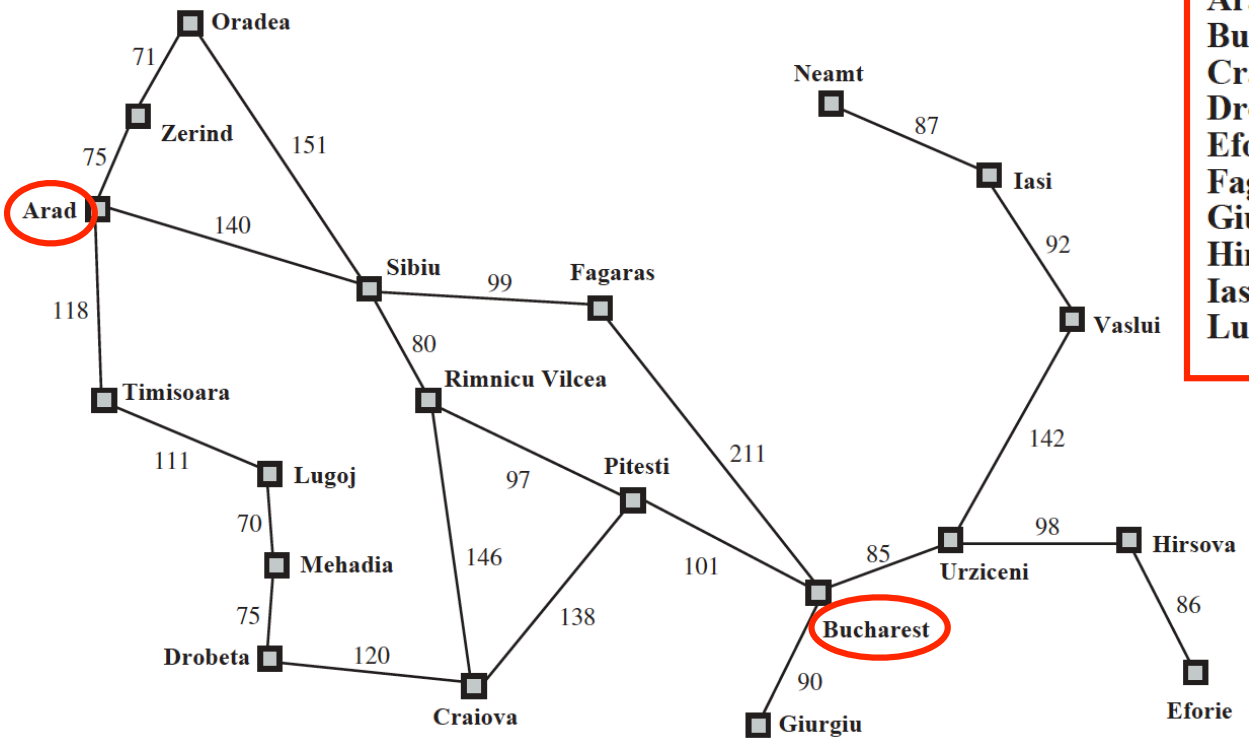


Straight-line distance (h_{SLD})

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

(b) After expanding Arad

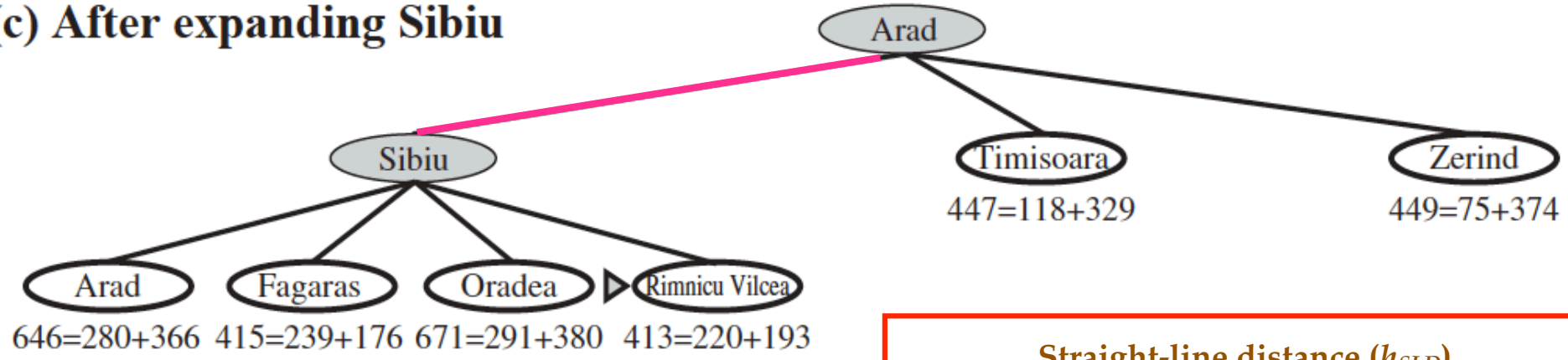


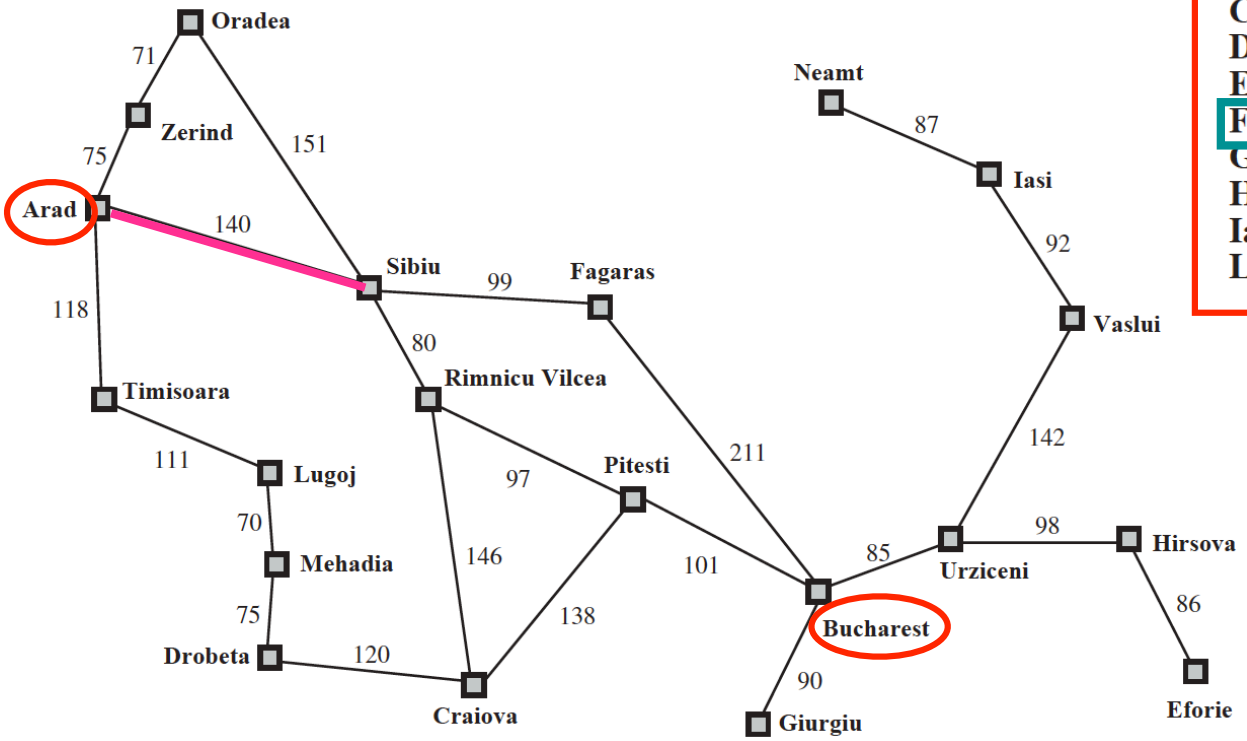


Straight-line distance (h_{SLD})

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

(c) After expanding Sibiu

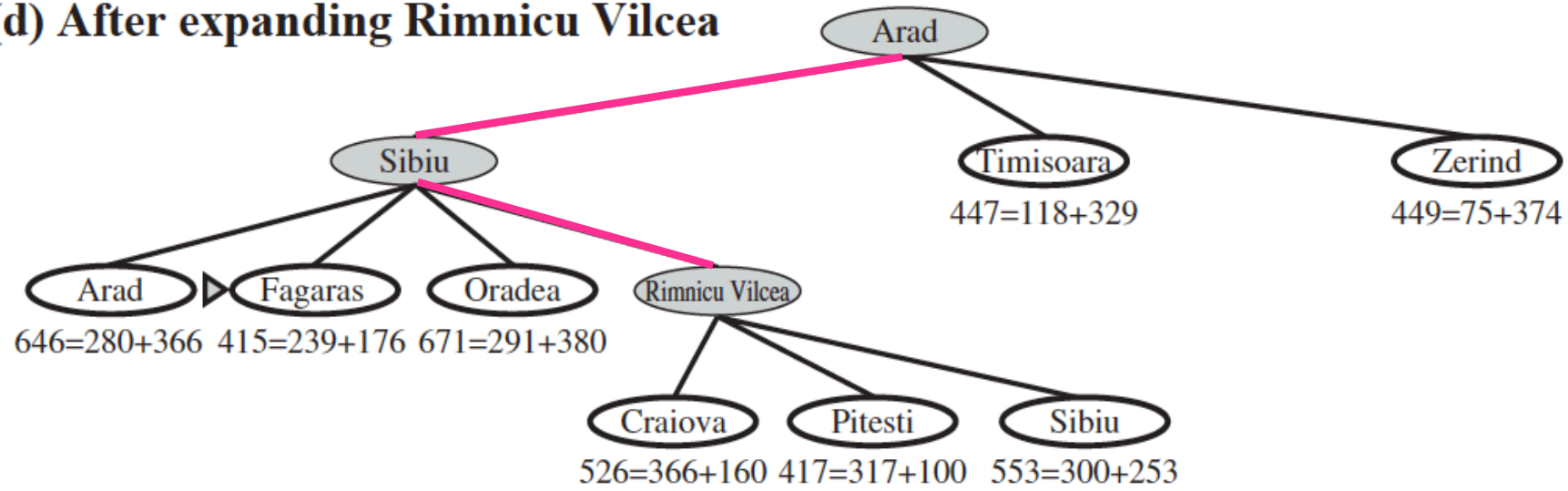


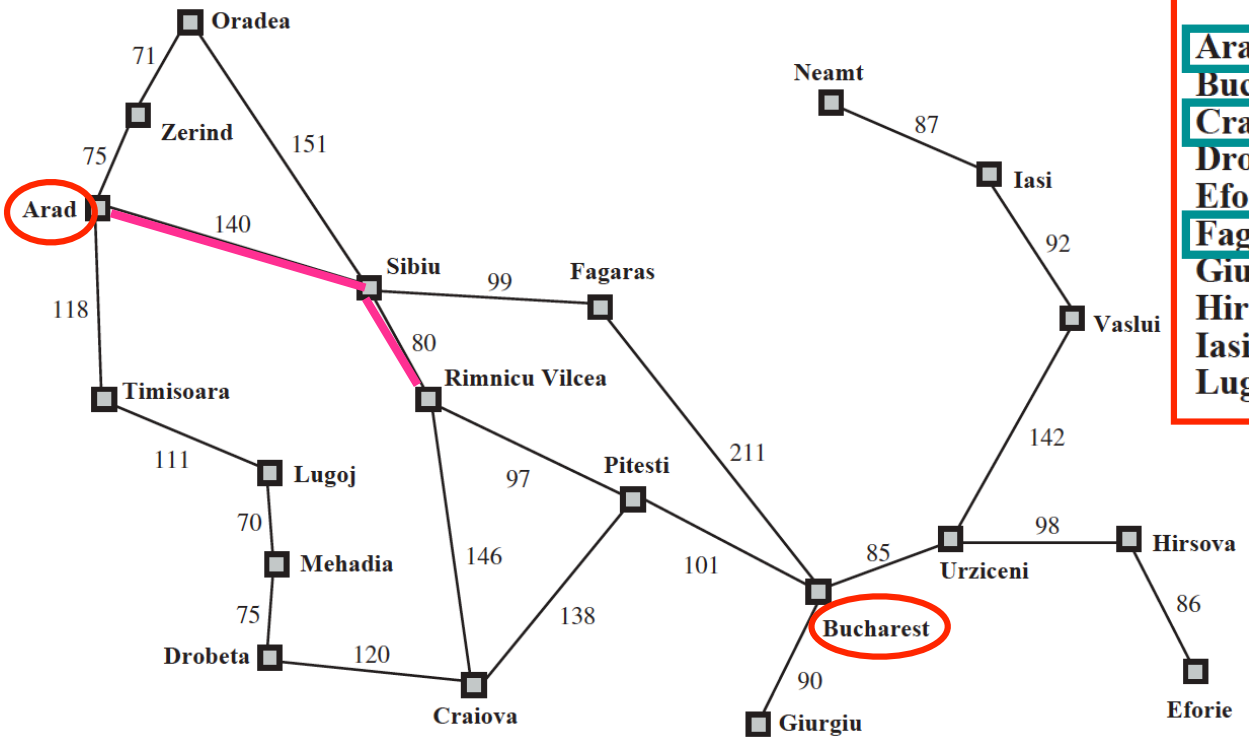


Straight-line distance (h_{SLD})

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

(d) After expanding Rimnicu Vilcea

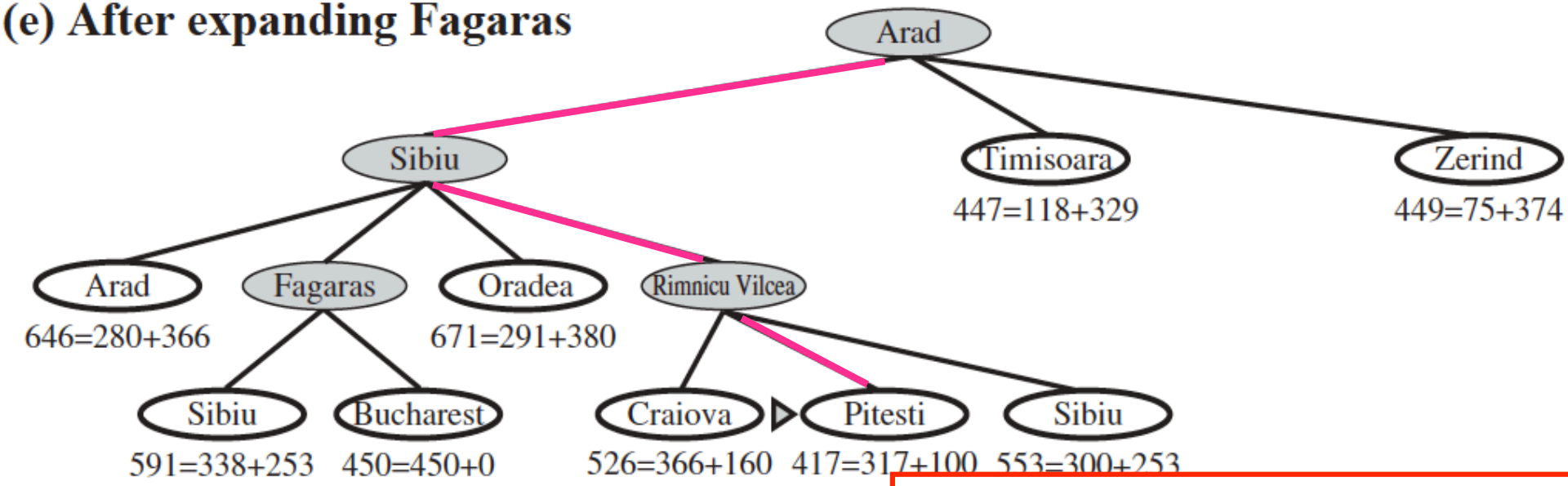


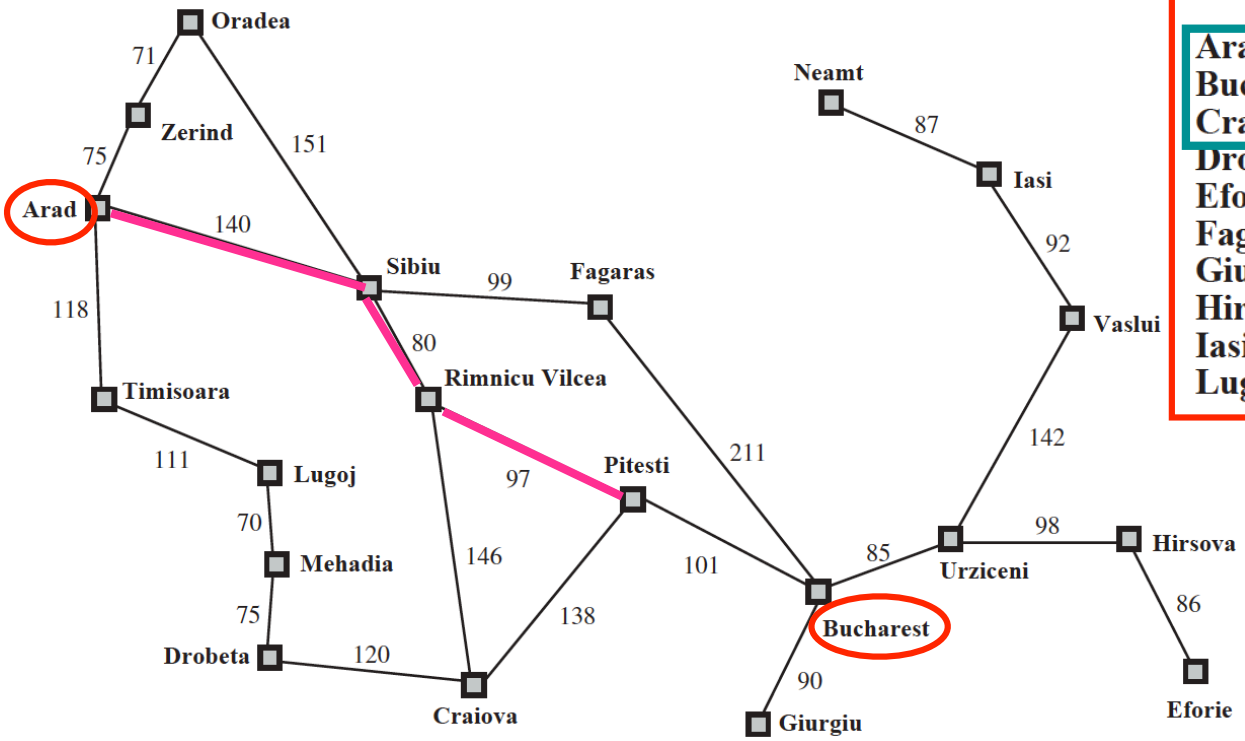


Straight-line distance (h_{SLD})

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

(e) After expanding Fagaras

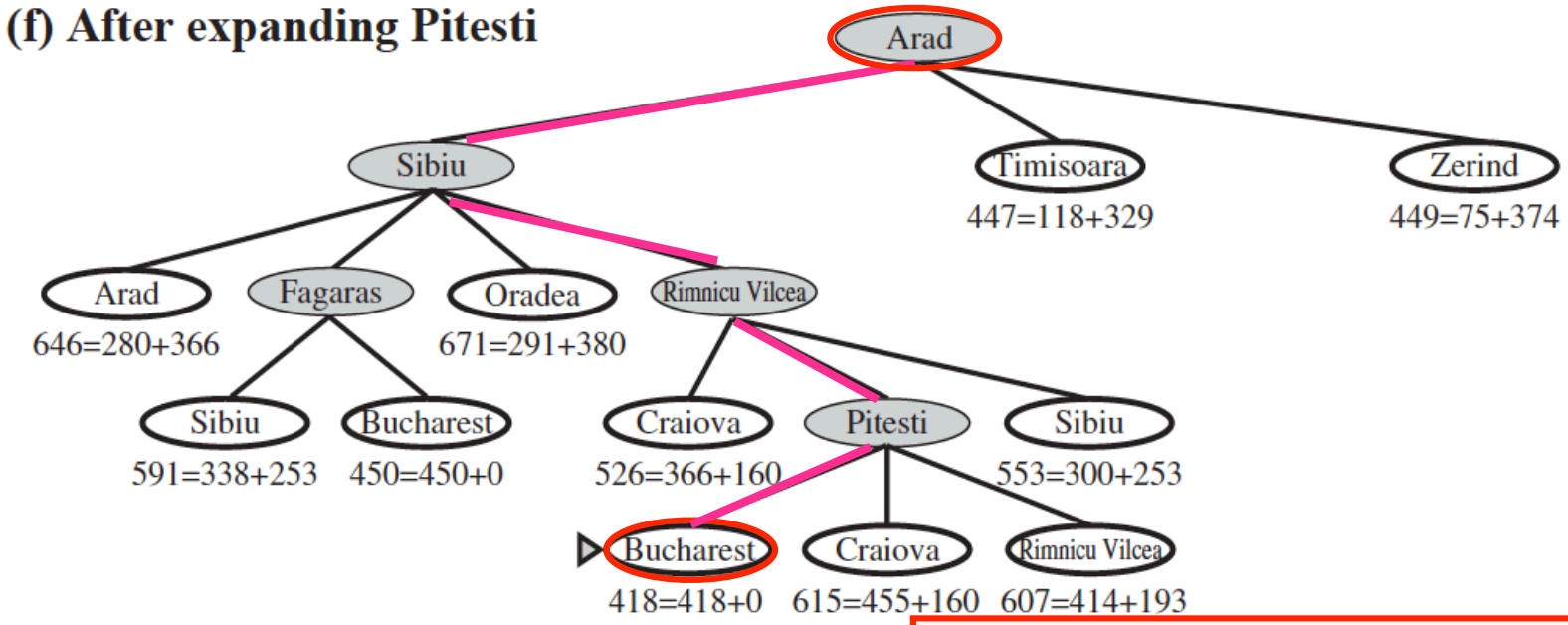


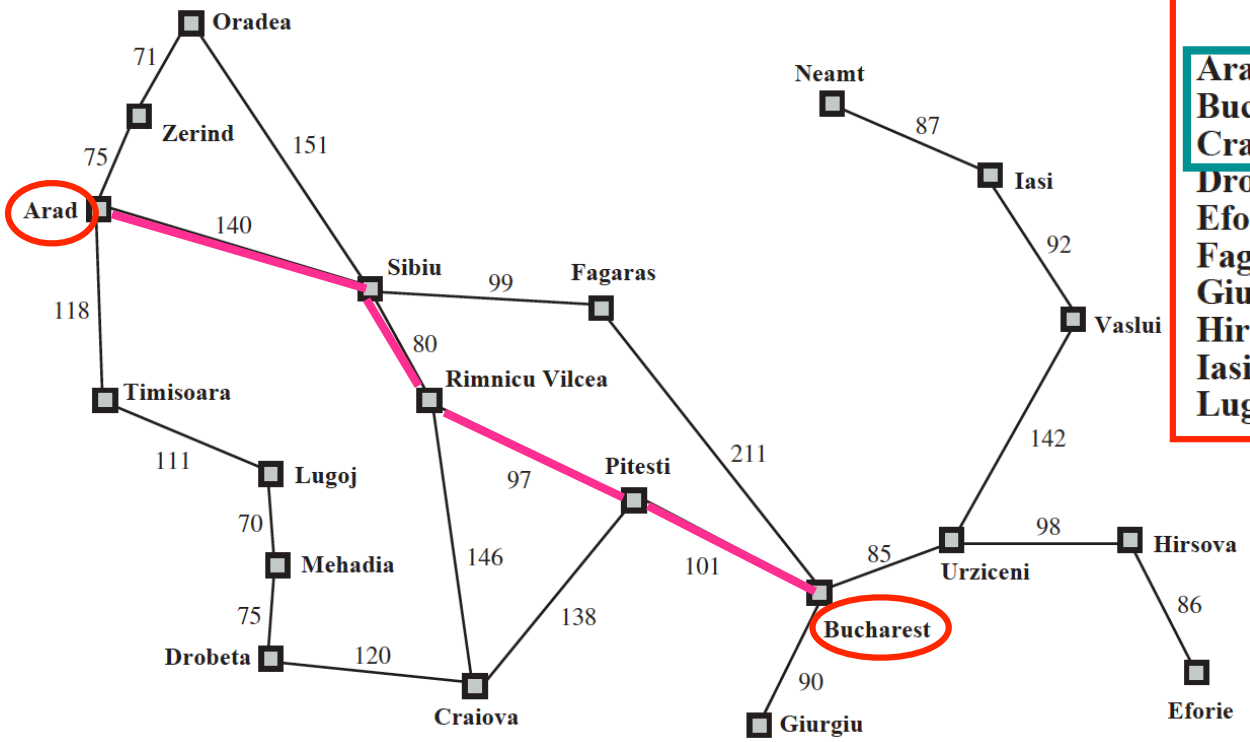


Straight-line distance (h_{SLD})

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

(f) After expanding Pitesti



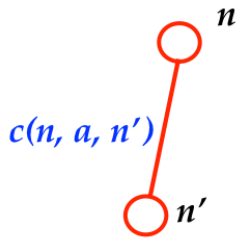


Straight-line distance (h_{SLD})

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

- **Properties of A* Search**
  - **Completeness**
    - **Yes**. Unless **infinite nodes with** $f \leqq f(goal)$

$$f_k, f_{k+1}, \ldots, f_\infty \leqq f(goal)$$



[most time $c(n, a, n') \geq \epsilon > 0$]

  - **Optimality**

$$= g(goal) + h(goal)$$

0

    - Depends on whether h is **admissible** (tree search)/**consistent** (graph search)
    - Optimal on **trees** if h is **admissible** (tree has at most single path to goal)
    - Optimal on **graphs** if h is **admissible** and **consistent** (graph may have more than one path to goal)

Admissible

- Never overestimates the actual cost.
- $\forall n, h(n) \leq h^*(n)$, where h^* is the actual cost.
- e.g., $h_{SLD}(n) \leq h^*(n)$.

Consistent

- A.k.a. monotonicity. [nondecreasing]
- $\forall$ successor n' of any n generated by any action a, $h(n) \leq c(n, a, n') + h(n')$, where c is the step cost.

triangle inequality

- **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
    - Example: $h_{SLD}(n) \leq h^*(n)$ (straight line distance never overestimates the actual road distance)
- **Theorem**: If $h(n)$ is **admissible**, A* using **Tree Search** is **optimal**

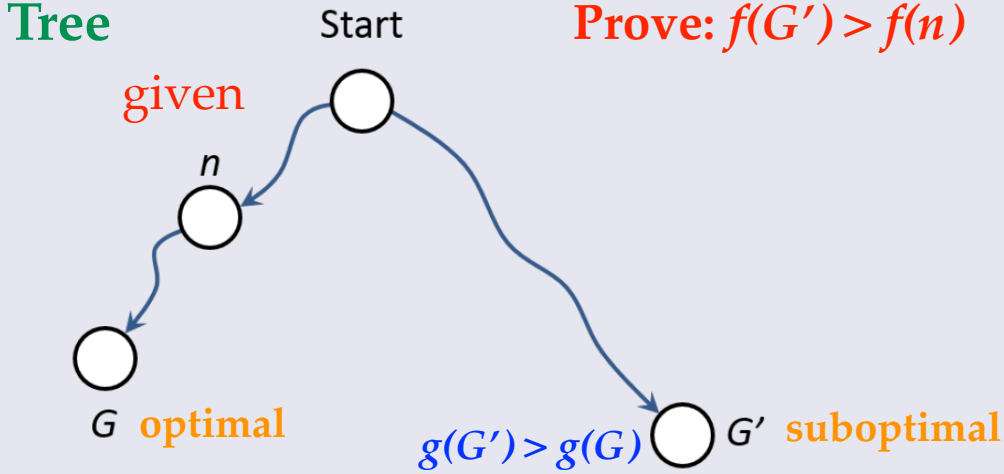
- **Time complexity**
  - $O(b^{\varepsilon d})$ for **constant** step costs, where $\varepsilon = (h^* - h) / h^*$ (relative error) and d is the solution depth
    - Time complexity much depends on the heuristic
    - e.g. $\varepsilon = 0$ *when* $h = h^*$, $\varepsilon = 1$ *when* $h = 0$
      - $O(b^0) = O(1)$ linear time
      - $O(b^{\varepsilon d}) = O(b^d)$ [same as BFS]
  - **Effective branch factor** is b^{ε}
- **Space complexity**
  - $O(b^d)$, since it keeps all nodes in memory

- **Optimality of A* on Trees** [at most one path to the goal]
  - Optimal on **trees** if h is **admissible**

Proof of A*'s optimality on trees.

Suppose some suboptimal goal G' has been generated and is in the queue. $g(G') > g(G)$

Let n be an unexpanded node on a shortest path to an optimal goal G.



Prove: $f(G') > f(n)$

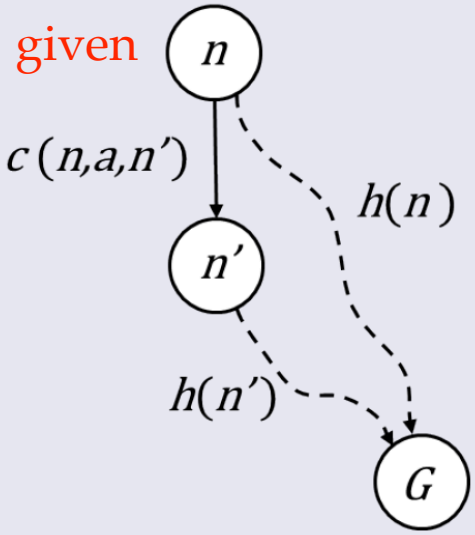
$$
\begin{aligned}
f(G') &= g(G') + h(G') && h(G) = 0,\ h(G') = 0 \\
&= g(G') \\
&> g(G) && g(G') > g(G) \\
&= g(n) + h^*(n) \\
&\geq g(n) + h(n) && h(n) \leq h^*(n) \\
&= f(n) && f(G') > f(n)
\end{aligned}
$$

$0 \leq h(G) \leq h^*(G) = 0$

Node n with lower $f(n)$ is expended earlier than G' with higher $f(G')$

- **Optimality of A* on Graphs** [may have more than on path to the goal]
  - Optimal on **graphs** if h is **admissible** and **consistent**
  - **Consistent**
    - ∀ successor n' of any n generated by any action a, $h(n) \leq c(n, a, n') + h(n')$ (**nondecreasing**), where c is the step cost
  - **Lemma**: If $h(n)$ is **consistent**, the values of f along any path in A* are **nondecreasing (Monotonic)**

Graph



given

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

Therefore,

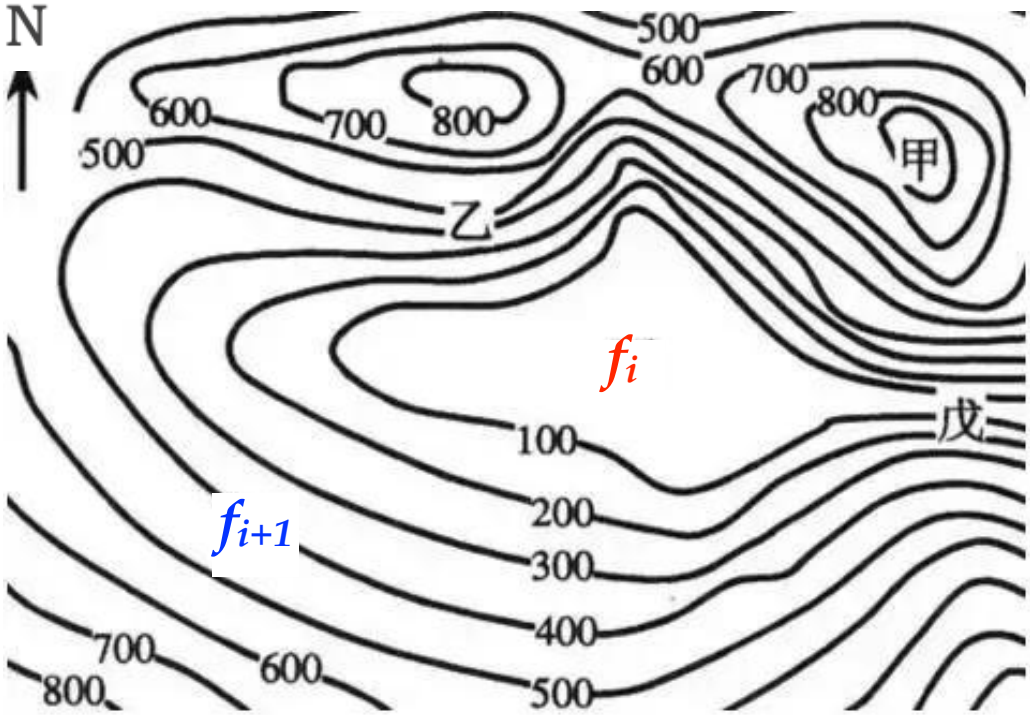
Prove: $f(n') \geq f(n)$

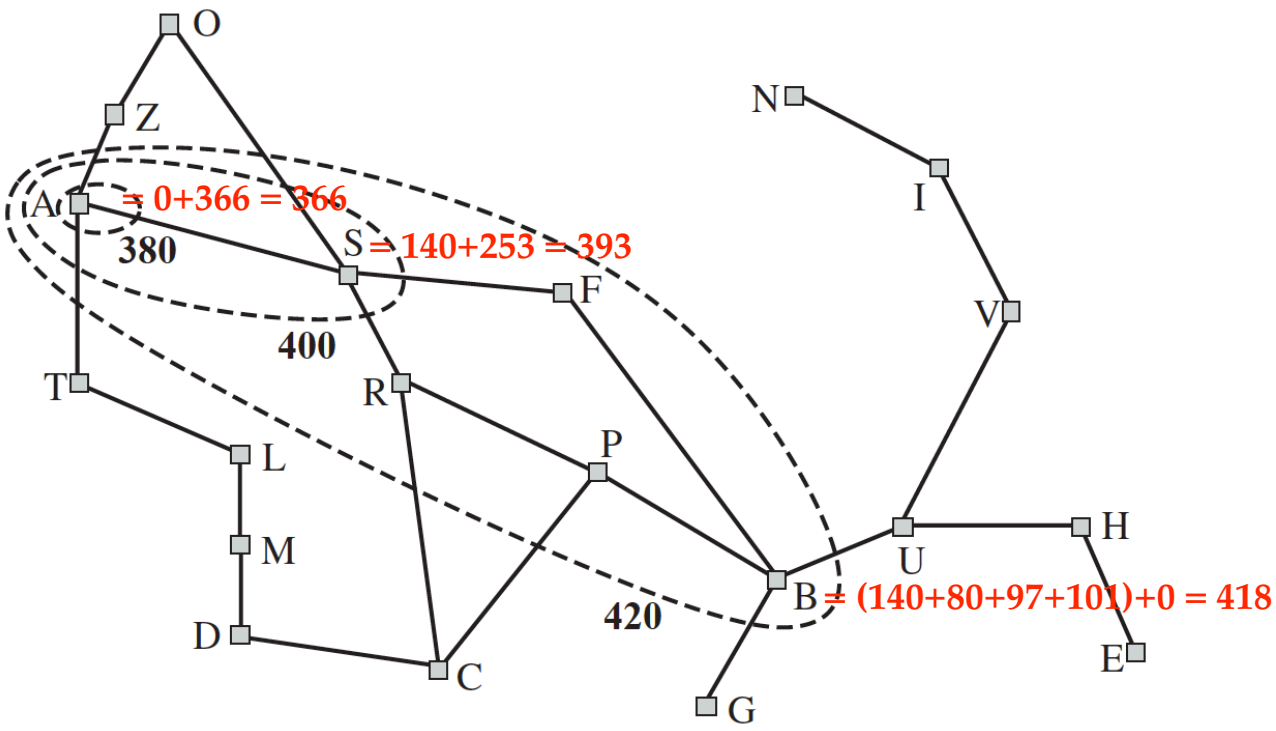
$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n, a, n') + h(n') \qquad (g(n')) \\
&\geq g(n) + h(n) \qquad \text{consistency} \\
&= f(n) \qquad f(n') \geq f(n)
\end{aligned}
$$

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

cutoff

(nondecreasing)





- Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with **Arad** as the start state
- Inside the contour labeled 400, all nodes have $f(n)$ less than or equal to 400, and so on.
- Because A* expands the frontier node of **lowest *f*-cost**, we can see that an A* search fans out from the start node, adding nodes in **concentric bands** of **increasing *f*-cost**

Memory-Bounded Heuristic Search

- Iterative Deepening A* (IDA*)
- Recursive Best-First Search (RBFS)
- Simplified Memory-Bounded A* (SMA*)

Iterative Deepening A* (IDA*)

A*

Time complexity: $O(b^{\varepsilon d})$

Space complexity: $O(b^{d})$

- Time complexity is not A*'s biggest drawback
- A* usually **runs out of memory** before it reaches goals
- Iterative deepening A* (IDA*)
  - Use $f(g + h)$ as **cutoff** instead of the **depth** **(DLS)**
  - Initial cutoff: $f(s_0) = h(s_0)$
  - Reset cutoff to **smallest** f of non-expanded nodes **(frontier)**
  - Perform **DFS** on nodes where $f(n) < cutoff$

IDA*(*problem*)

```
1   currentCutoff = f(s0)
2   repeat
3       result = f-LIMITED-SEARCH(problem, currentCutoff)
4       if result ≠ cutoff
5           return result
6       currentCutoff = smallest f-value of non-expanded nodes.
```

- **Properties of IDA***
  - **Completeness** and **Optimality** same as A*
    - **Completeness**
      - **Yes**. Unless **infinite nodes with** $f \leqq f(goal)$

        $f_k, f_{k+1}, \ldots, f_\infty \leqq f(goal)$
    - **Optimality**

      [most time $c(n, a, n') \geq \epsilon$]
      - Optimal on **trees** if h is **admissible**
      - Optimal on **graphs** if h is **admissible** and **consistent**
  - **Time complexity**: $O(b^{\varepsilon d})$
  - **Space complexity**: $O(bd)$ [DFS]

A*

Time complexity: $O(b^{\varepsilon d})$

Space complexity: $O(b^d)$

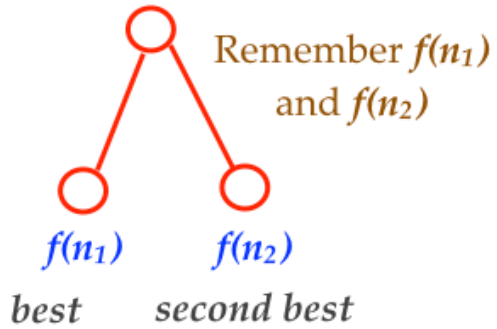
- What happens if all *f*-values are different (real-values)?
  - The **no. of iterations** can equal the **no. of nodes** whose ***f*-value** is **less** than the **cost of an optimal path**! [Since *f*-values are different, there may have many interactions causing severe repeated search]

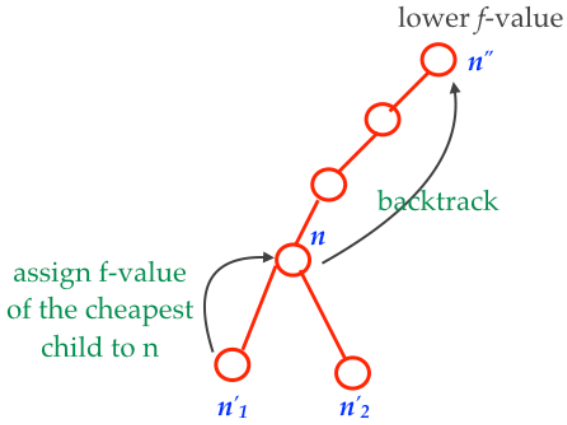
Recursive Best-First Search (RBFS)

- RBFS is a simple recursive algorithm that mimics standard **best-first search** using only **linear space**

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
    return RBFS(*problem*, MAKE-NODE(*problem*.INITIAL-STATE), ∞)

- **RBFS** uses **DFS** where each node on the current path **remembers** the **best *f*-value** of **any alternative path from its ancestors**
- Maintains **all nodes on current path** plus all their **siblings** (from ***ancestor(n)***)
- When expanding node ***n***
  - $\forall\ n' \in children(n)$, compute $f(n')$
  - If an ancestor ***n″*** has a lower ***f*-value** than all ***n***'s children, then
    - Assign *f*-value of the **cheapest child** to ***n***
    - **Backtrack** to ***n″***
  - Otherwise, proceed as normal



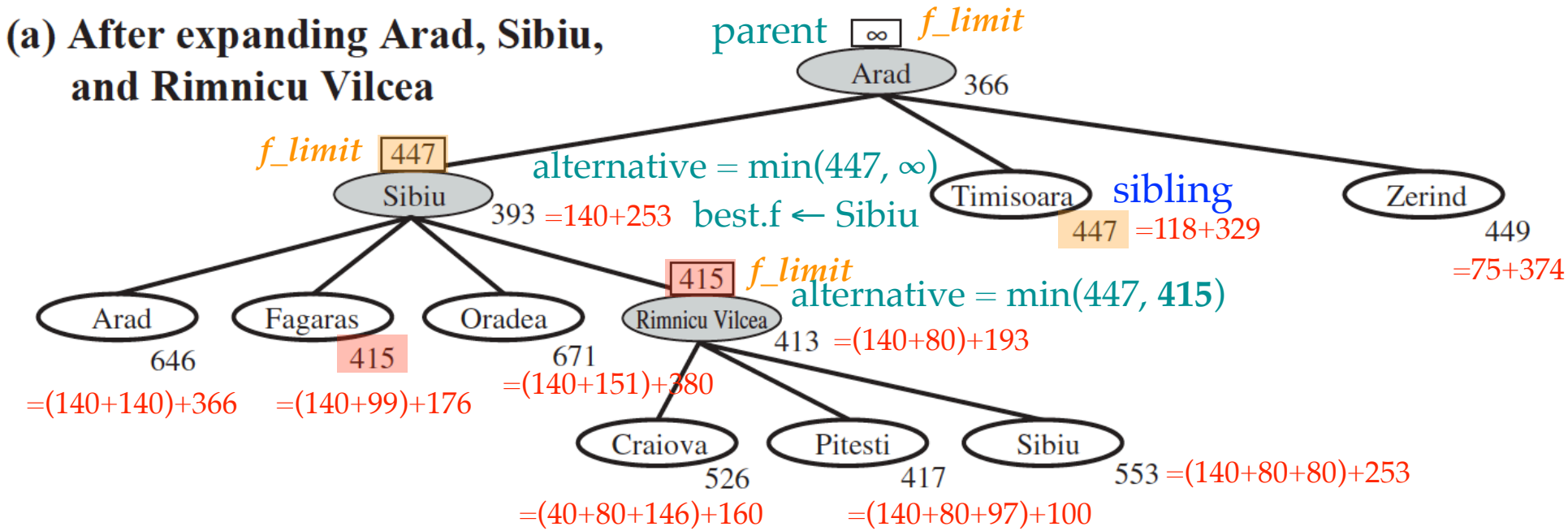


```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)
```

cutoff

```
function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

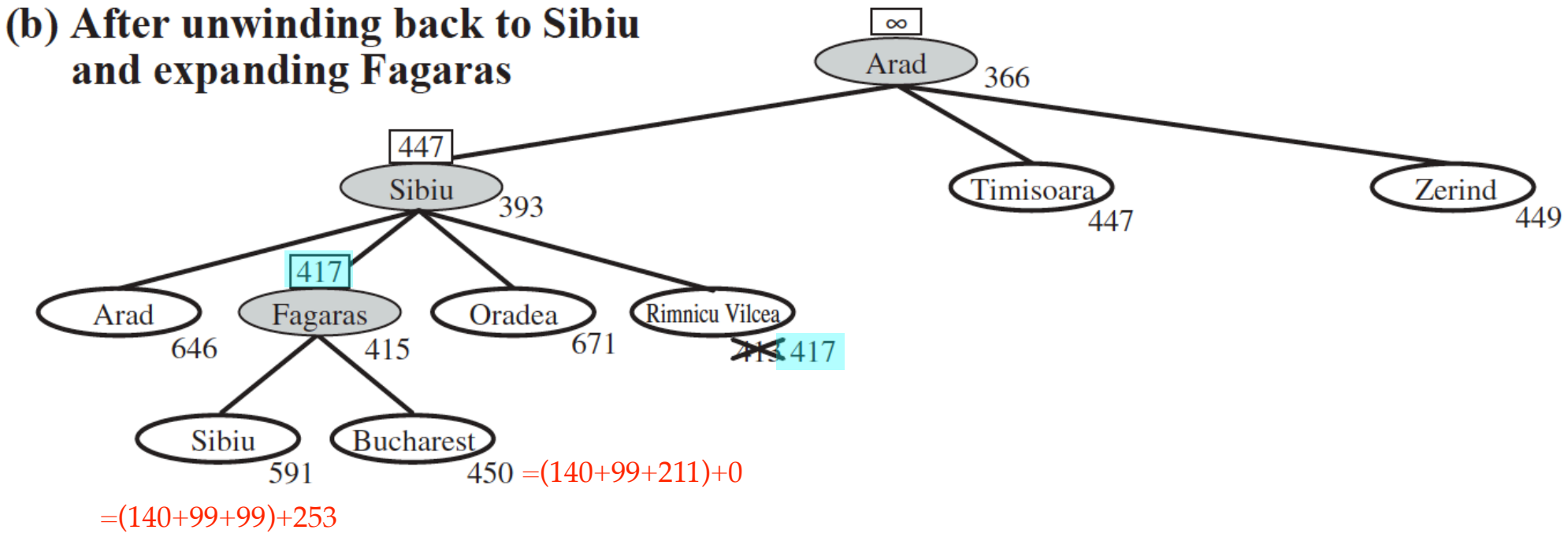
[parents] [sibling]



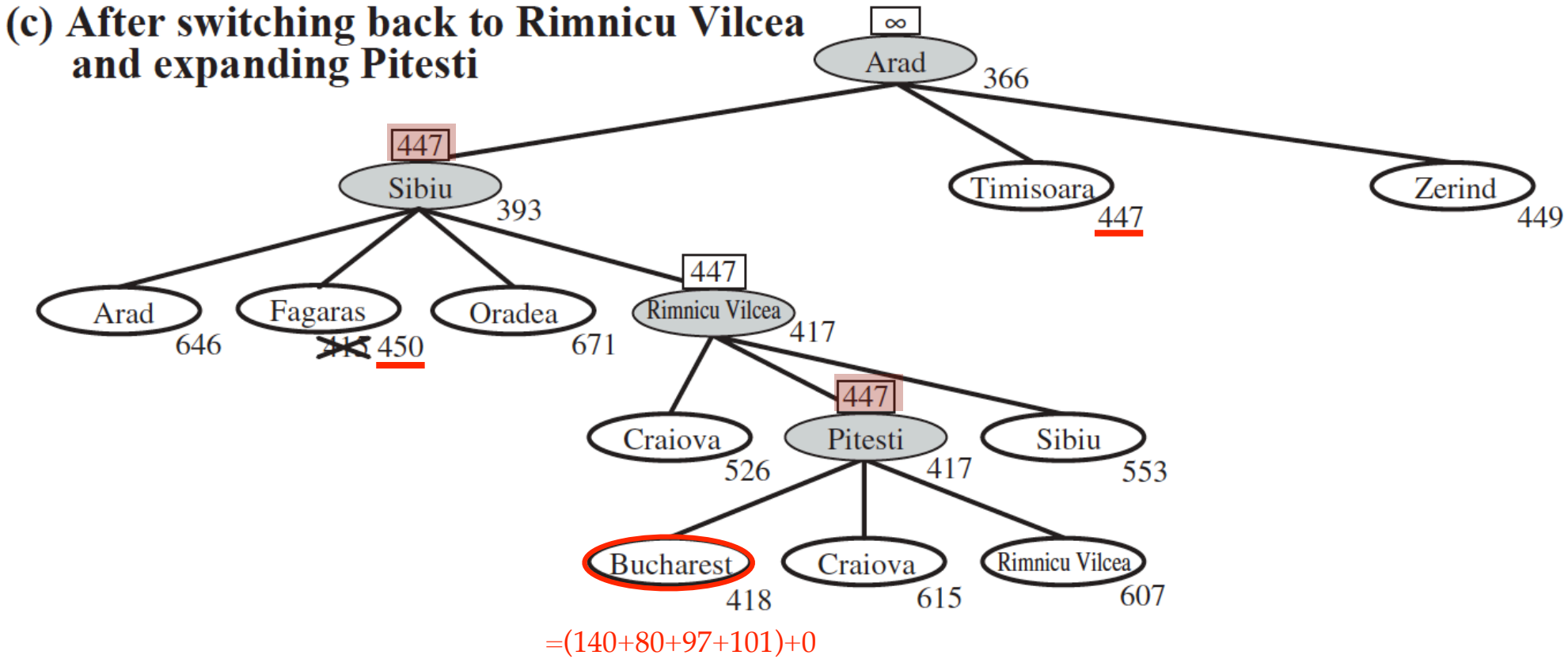
- Stages in an RBFS search for the shortest route to Bucharest

Straight-line distance (h_{SLD})			
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

- The ***f*-limit** value for each recursive call is shown on **top** of each current node, and every node is labeled with its ***f*-cost**
- (a) The path via Rimnicu Vilcea is followed **until** the **current best leaf (Pitesti, 417)** has a value that is **worse** than the best alternative path (Fagaras, 415)



- (b) The recursion **unwinds** and the best leaf value of the forgotten subtree (**417**) is **backed up to Rimnicu Vilcea**; then Fagaras is expanded, revealing a best leaf value of 450 (**Bucharest**) [450 > 417]



- (c) The recursion **unwinds** and the best leaf value of the forgotten subtree (**450**) is **backed up to Fagaras**; then Rimnicu Vilcea (**417**) is expanded
- This time, because the best alternative path (through Timisoara) costs at least 447 [447 > 418], the expansion continues to Bucharest (**418**)

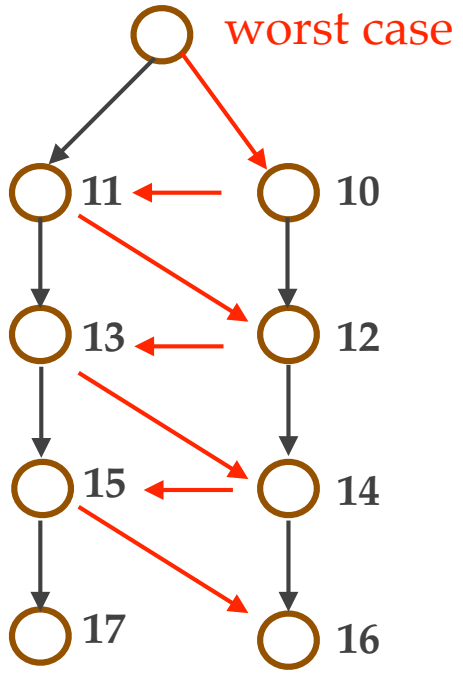
- **Properties of RBFS**
  - **Completeness** and **optimality** same as A*
    - **Completeness**

      $f_k, f_{k+1}, \ldots, f_\infty \leqq f(goal)$

      - **Yes**. Unless **infinite nodes with** $f \leqq f(goal)$

        [most time $c(n, a, n') \geq \epsilon$]
    - **Optimality**
      - Optimal on **trees** if *h* is **admissible**
      - Optimal on **graphs** if *h* is **admissible** and **consistent**

- **Time complexity**: Depends on accuracy of *h* and on **how often best path changes**
- **Space complexity**: *O(bd)*
- Each time RBFS **changes** its mind corresponds to **one iteration of IDA***
- RBFS may need to **re-expand forgotten** nodes to re-create **best-path**



Summary

If $b = 4$, $d = 100$ and 1 record = 2KB, then total requires 800KB memory

[O(bd)] [O(bd)]

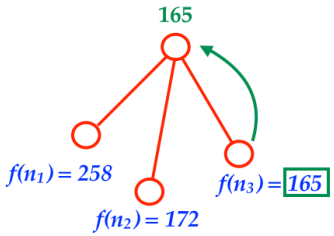
- In a sense, both IDA* and RBFS use **too little memory**
  - Between iterations, **IDA*** maintains only **one number**, the **current *f*-limit (*currentCutoff*)**
  - **RBFS** maintains **more**, but uses only **linear space**: if more space were available, it would not benefit from it
- It seems reasonable to **use all the memory available** — the more, the better (don't change path before using up memory)
- We'd like a **memory-bounded** version of **A*** (don't change path before use up memory)

e.g. Change path only if use up 500Mb memory.

Simplified Memory-Bounded A* (SMA*)

e.g. 500MB

- **Idea**: Run **A*** as normal **until memory is full**. Then **replace** something in memory with **newly generated nodes**



- **SMA***
  - When memory is full, **drop the worst leaf** — node with **highest *f*-value**
  - Like RBFS, SMA* **backs up** the **lowest *f*-value** of this forgotten nodes to it's **parent**, so we know when to **go back** to it
  - If all **descends** of a node ***n*** are **forgotten**, we don't know which way to go from ***n***, but we know if it's worth **re-exploring *n***

- **Problem**: What if many (e.g. 50%) nodes have the **same *f*-value**?
  - **Solution**: **delete** the **oldest** and **expand** the **newest** (at the cost of adding **time stamp** to each node)
- SMA* works as long as there is **enough memory for the complete optimal path**
  - If not, SMA* needs to **switch continuously** between **candidate paths**
  - Causes a similar problem to **thrashing** in **disk paging** systems

In computer science, **thrashing** occurs when a computer's virtual memory resources are overused, leading to a constant state of **paging** and **page faults**.

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

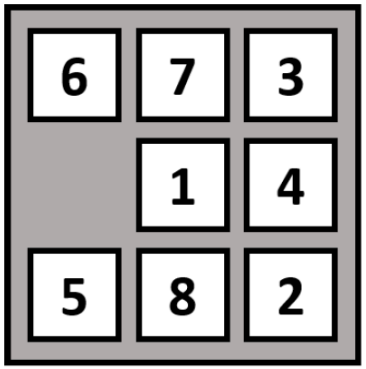
  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
         f(s) ← ∞
      else
         f(s) ← MAX(f(n),g(s)+h(s))
      if all of n's successors have been generated then
         update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
         delete shallowest, highest-f-cost node in Queue
         remove it from its parent's successor list
         insert its parent on Queue if necessary
      insert s in Queue
   end
```

3.6 Heuristic Functions

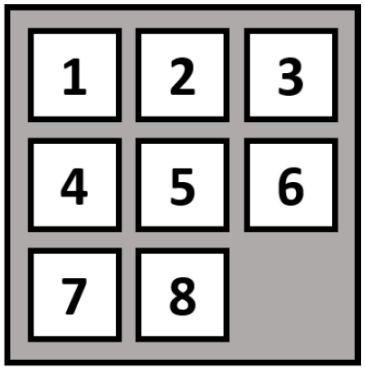
- The effect of **heuristic accuracy** on performance
- Generating **admissible heuristics** from **relaxed problems**
- Generating **admissible heuristics** from **sub-problems**: **pattern databases**
- Learning heuristics from **experience**

Admissible Heuristics for 8-Puzzle

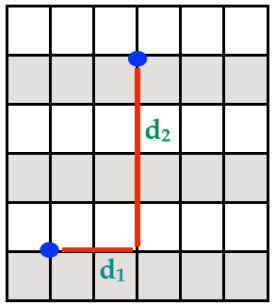
- h_1 = the number of **misplaced tiles** (admissible, never overestimate)
- h_2 = the sum of **Manhattan distances** of the tiles from their goal positions (admissible, never overestimate) ($h_2 = d_1 + d_2$)



Start State



Goal State



$h_2 = d_1 + d_2$

- $h_1(s_0) = 6$ (1, 2, 4, 5, 6, 7)

2 → 4 → 3 → 7

- $h_2(s_0) = 2 + 3 + 0 + 2 + 2 + 3 + 3 + 0 = 15$

1 → 7 → 6

Performance of Heuristic

Definition

For two admissible heuristics h_1 and h_2, h_2 dominates h_1 **iff** $\forall n, h_2(n) \geq h_1(n)$.

$$h^*(n) \geqq h_2(n) \geqq h_1(a) \geqq 0$$

C*: the cost of the optimal goal

Theorem: A* using h_2 never expands more nodes than using h_1.

- Every node with $f(n) < C^*$ is expanded. $g(n) + h(n) < C^*$
- Every node with $h(n) < C^* - g(n)$ is expanded. $f(n) = g(n) + h(n)$
- $|\{n|\ h_2(n) < C^* - g(n)\}| \leq |\{n|\ h_1(n) < C^* - g(n)\}|$

h_2 moves faster — h_1 moves slower

- Given a state n, $C^* - g(n)$ is **fixed**
  - h_2 has better performance than h_1
  - A* search with h_1 will expand more nodes
- Given any **admissible** heuristics h_a and h_b
  - $h = max(h_a, h_b)$ is also **admissible**
  - h **dominates** h_a and h_b, respectively

- Typical **search costs** for the 8-puzzle (**average no. 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, $h_1(n)$, $h_2(n)$, …, $h_m(n)$, but **none of them dominates the others**
  - How can we **combine** them?

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

    - e.g. $h_3(n_1)$, $h_4(n_2)$, $h_1(n_3)$, $h_3(n_4)$, $h_2(n_5)$, …….
  - This **composite heuristic** uses whichever function is **most accurate** on the node in question (for each node n_i, choose the h_j which has the max h value)

- **Effective Branching Factor**
  - One way to characterize the **quality** of a heuristic is ***effective branching factor, b^****
  - Let A* generate ***N*** nodes to find a goal at **depth *d***
    - b^* is the branching factor that a **uniform tree of depth *d*** would have in order to contain ***N+1*** nodes

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

$$N+1 = ((b^*)^{d+1} - 1)/(b^* - 1)$$

$$N \approx (b^*)^d \Rightarrow \boxed{b^* \approx \sqrt[d]{N}}$$

  - For **sufficiently hard problems**, the measure b^* usually is fairly **constant** across different problem instances
  - A good way to **compare different heuristics**
  - A **well-designed heuristic** would have a value of b^* close to **1**

Effective Branching Factor Pseudo-code (Binary Search)

- ```
  PROCEDURE EFFBRANCH (START, END, N, D, DELTA)
        COMMENT DELTA IS A SMALL POSITIVE NUMBER FOR ACCURACY OF RESULT.
        MID := (START + END) / 2.
        IF (END - START < DELTA)
           THEN RETURN (MID).
        TEST := EFFPOLY (MID, D).
        IF (TEST < N+1)
           THEN RETURN (EFFBRANCH (MID, END, N, D, DELTA) )
           ELSE RETURN (EFFBRANCH (START, MID, N, D, DELTA) ).
  END EFFBRANCH.

  PROCEDURE EFFPOLY (B, D)
        ANSWER = 1.
        TEMP = 1.
        FOR I FROM 1 TO (D-1) DO
           TEMP := TEMP * B.
           ANSWER := ANSWER + TEMP.
        ENDDO.
        RETURN (ANSWER).
  END EFFPOLY.
  ```

- For binary search please see: http://en.wikipedia.org/wiki/Binary_search_algorithm
- An attractive alternative is to use Newton's Method (next lecture) to solve for the root (i.e., f(b)=0) of

  f(b) = 1 + b + ... + b^d - (N+1)
  ```

- Comparison of the **search costs** and **effective branching factors** for the IDS and A* algorithms with h_1, h_2
- Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d

$h_1(n) \leq h_2(n)$

h_2 has smaller branch factor

	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 (huge)	227	73	2.78	1.42	1.24
14	hard to compare	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

approximate to fixed value

the deeper the more accurate

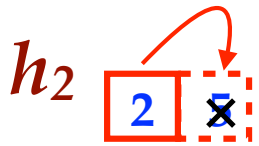
Comparison of Search Strategies

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: $O(b^m)$ Best case: $O(bd)$	
A*	Yes	Yes	Number of nodes with $g(n)+h(n) \leq C^*$	

Generating Admissible Heuristics from Relaxed Problems

- A problem with **fewer restrictions** on the **actions** is called a **relaxed problem** **(remove some constraints)**
- **Admissible heuristics** can be derived from exact solution cost to a **relaxed** version of the problem
- The **cost** of an optimal solution to a **relaxed problem** is an **admissible heuristic** for the original problem

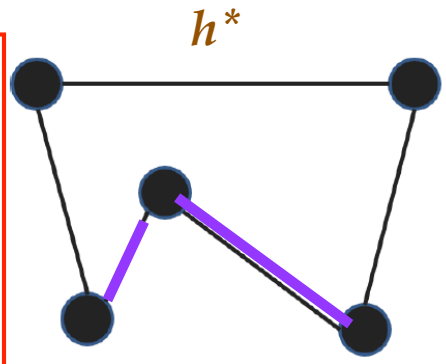
- In 8-puzzle
  - h_1 (admissible) is derived from that **a tile can move to anywhere** in **one step**, then $h_1(n)$ gives the **shortest solution**
  - h_2 (admissible) is derived from that **a tile can move to any adjacent square** in **one step**, then $h_2(n)$ gives the **shortest solution**

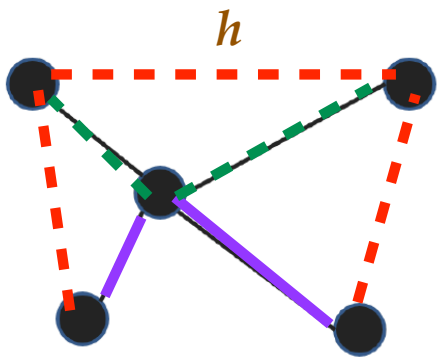


- <u>Key</u>: The optimal solution **cost** of a **relaxed problem** is **no greater than** the optimal solution cost of the **original problem**

- **Problem Relaxation Example: TSP**
  - Traveling salesman problem (**TSP**)
  - Known to be ***NP*-hard** (solution can't be derived in polynomial time)

TSP：Given a list of cities and the distances between each pair of cities, what is **the shortest possible route that visits each city and returns to the origin city**



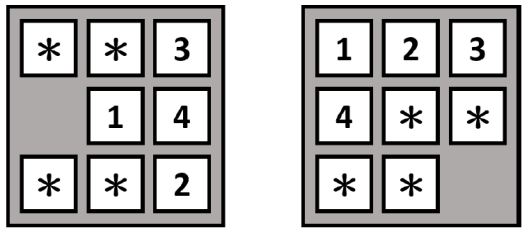


MST (Minimum Spanning Tree)：A subset of the **edges** of a **connected, edge-weighted undirected graph** that connects all the vertices together, without any cycles and with the **minimum possible total edge weight**.

  - Can be relaxed to **minimum spanning tree (MST)**
    Take **MST** as heuristic (h) to solve the **TSP** problem (h^*), h is admissible
  - **MST** cost is **never greater than** the **shortest tour**
    Triangle inequality should hold
  - Cost of MST can be computed in $O(n^2)$

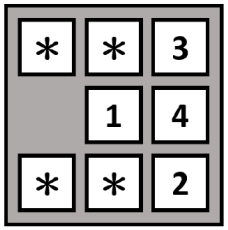
TSP：NP-hard
MST：$O(n^2)$

Generating Admissible Heuristics from Sub-problems: Pattern Databases

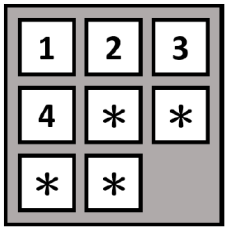


- **Admissible heuristic** can also be derived from a **subproblem** (do not remove problem constraints)
- **Pattern databases** store **exact solution costs** (**precomputed**) for every possible **subproblem instances**, e.g.
  - Compute 1-2-blank and don't care 3-4-5-6-7-8 (mark in *)
  - Compute 3-4-blank and don't care 1-2-3-4-5-6 (mark in *)
  - .....
  - Compute 1-2-3-4-blank and don't care 5-6-7-8 (mark in *)
  - Compute 5-6-7-8-blank and don't care 1-2-3-4 (mark in *)

- Can we use the costs of 1-2-3-4 (h_1) and 5-6-7-8 (h_2)?
  - **Simple addition breaks** the **admissibility** (may be overestimate, $h_1+h_2 \geqq h^*$)
  - How about count only those moves involving 1-2-3-4 or 5-6-7-8, respectively, and deduct the no. of movements of * (indicating 5, 6, 7, 8) while manipulating 1-2-3-4, and deduct the no. of movements of * (indicating 1, 2, 3, 4) while manipulating 5-6-7-8
    - Then the **addition** is still **admissible**
    - This is the idea behind **disjoint pattern databases**



Start State



Goal State

Learning Heuristics from Experience

- Convert a **state** into the **feature** domain [machine learning]
  - Feature $f_1(n)$: no. of **misplaced tiles**
  - Feature $f_2(n)$: no. of pairs of **adjacent tiles** that are not adjacent in the goal state
- Both $f_1(\textbf{goal}) = 0$ and $f_2(\textbf{goal}) = 0$

- $h(n) = c_1 f_1(n) + c_2 f_2(n)$ with $c_1 > 0; c_2 > 0$ $h(n) = \sum c_i f_i(n)$
  - c_1 and c_2 are adjusted (using **linear regression**) to give the **best fit** to the actual data on solution costs
- We could take randomly generated 8-puzzle and gather **statistics** to decide **constants**

  (linear regression)
- **No guarantee** to be **admissible** or **consistent**

Summary

- **Heuristic** functions estimate costs of **shortest paths**
- Good heuristics can dramatically reduce search cost
- **Greedy best-first search** expands **lowest** h
  - In general **not complete nor optimal**
- **A* search** expands **lowest** $g + h$
  - **Optimal** when h is **admissible** (and **consistent**)

- **Memory limitation** is an important issue to heuristic search (IDA*, RBFS, SMA*)
  - Search with **forgetting** and **re-expanding** are the keys, but still suffers from different conditions
- A **more efficient heuristic** can be generated from **several admissible heuristics**
- Admissible heuristics can be derived from **relaxed problems**, **subproblems**, and **experience**

(no guarantee admissible & consistent)