Solving Problems by Searching

Learning Goals

- Problem-solving agents
- Example Problem
- Searching for Solution
- Uninformed Search Strategies
- Avoiding Repeated States
- Searching with Partial Information

Problem Solving Agents

Problem Solving Agents

- Problem Solving Agent:
  - An agent with several options can first examine different possible sequences of actions to choose the best sequence
  - use **atomic representations**,
    - states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms.
  - In general, *an agent with several immediate options of unknown value can decide what to do by first examining* future *actions that eventually lead to states of known value.*

Problem Solving Environment

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave search, execution
- Unknown state space → exploration problem
- *Under these assumptions, the solution to any problem is a fixed sequence of actions.*

Well-defined searching problems and solutions /problem formulation

- State space (forms a directed network or graph)
  - Initial state
  - Successor function
    - description of the possible actions
    - **ACTIONS(s)** returns the set of actions that can be executed in s.
    - **transition model:** A description of what each action does, specified by a function **RESULT(s, a)** that returns the state that results from doing action a in state s.
- **Goal test**: determines whether a given state is a goal state.
- Path cost / step cost
  - (A **path** in the state space is a sequence of states connected by a sequence of actions.)
- Solution/ optimal solution

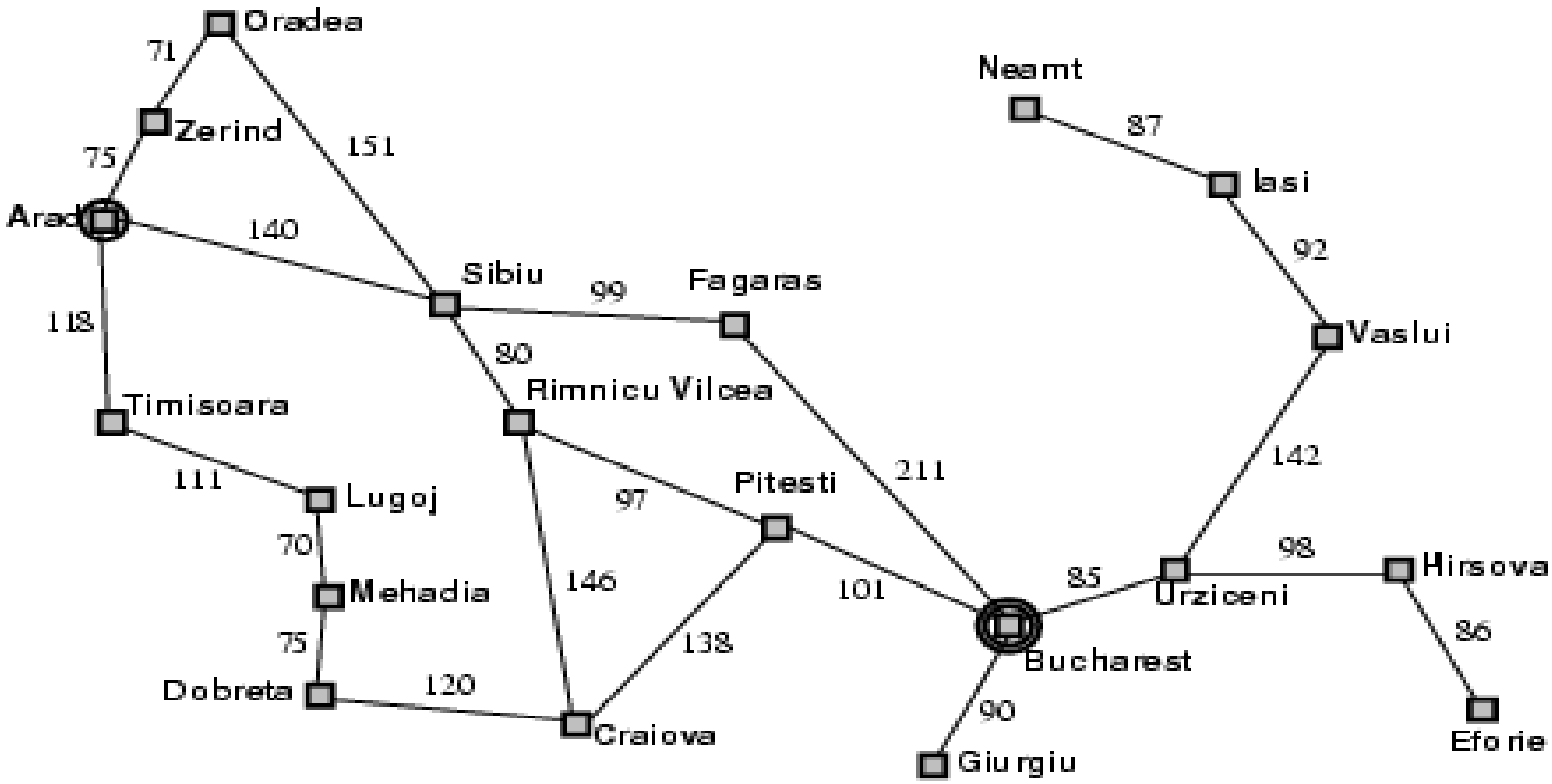
Problem-solving agents

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

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

http://aima.cs.berkeley.edu/figures.html

Example Problem 1: Romania

- What is the shortest path from Arad to Bucharest?



http://aima.cs.berkeley.edu/figures.html

More concrete problem definition

A state space	***Choose a representation***
An initial state	***Choose an element from the representation***
A goal state	***Create `goal_function(state)` such that TRUE is returned upon reaching goal***
A function defining state transitions	***`successor_function(state)` = {<action, state>, <action, state>, …}***
A function defining the "cost" of a state sequence	***`cost (sequence)` = number***

Important notes about this example

- Static environment (available states, successor function, and cost functions don't change)
- Observable (the agent knows where it is… percept == state)
- Discrete (the actions are discrete)
- Deterministic (successor function is always the same)

Single-state problem formulation

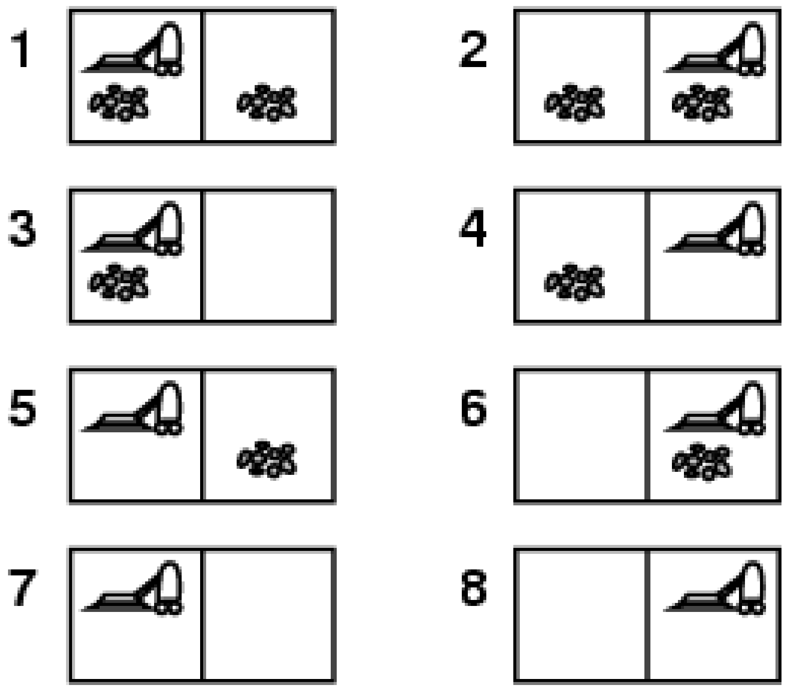
A problem is defined by four items:

1. initial state e.g., "at Arad”
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

Example Problem 2: vacuum world

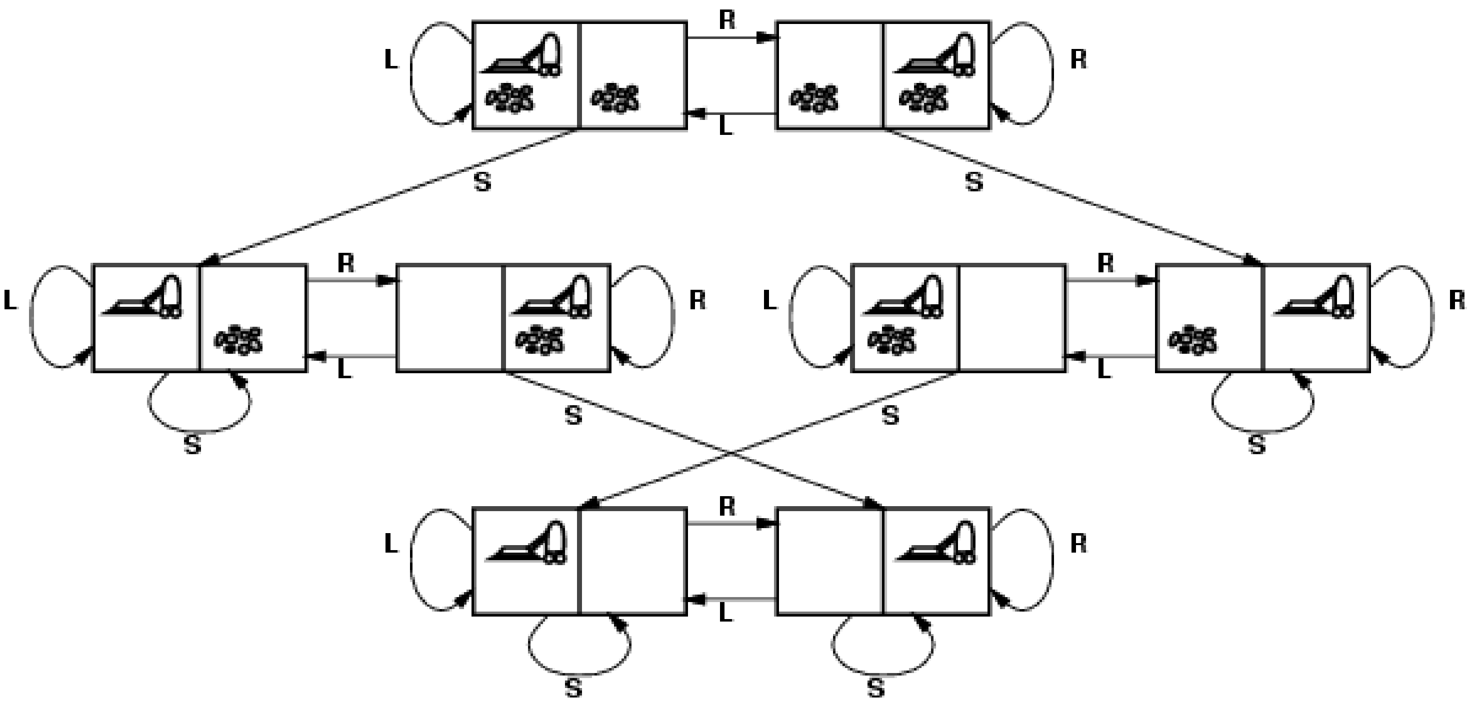
- Single state problem: #5, solution?
- Sensorless,
  - start in (unknow) {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
    Solution?
    [Right,Suck,Left,Suck]



- Contingency (偶發事故)
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*

http://aima.cs.berkeley.edu/figures.html

Vacuum world state space graph



http://aima.cs.berkeley.edu/figures.html

- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example Problem 2: 8-Puzzle

- 9!/2=181,440 states

7	2	4
5		6
8	3	1

Initial state

	1	2
3	4	5
6	7	8

Goal state

Search is about the exploration of alternatives

8-Puzzle: State Space

8	2	
3	4	7
5	1	6

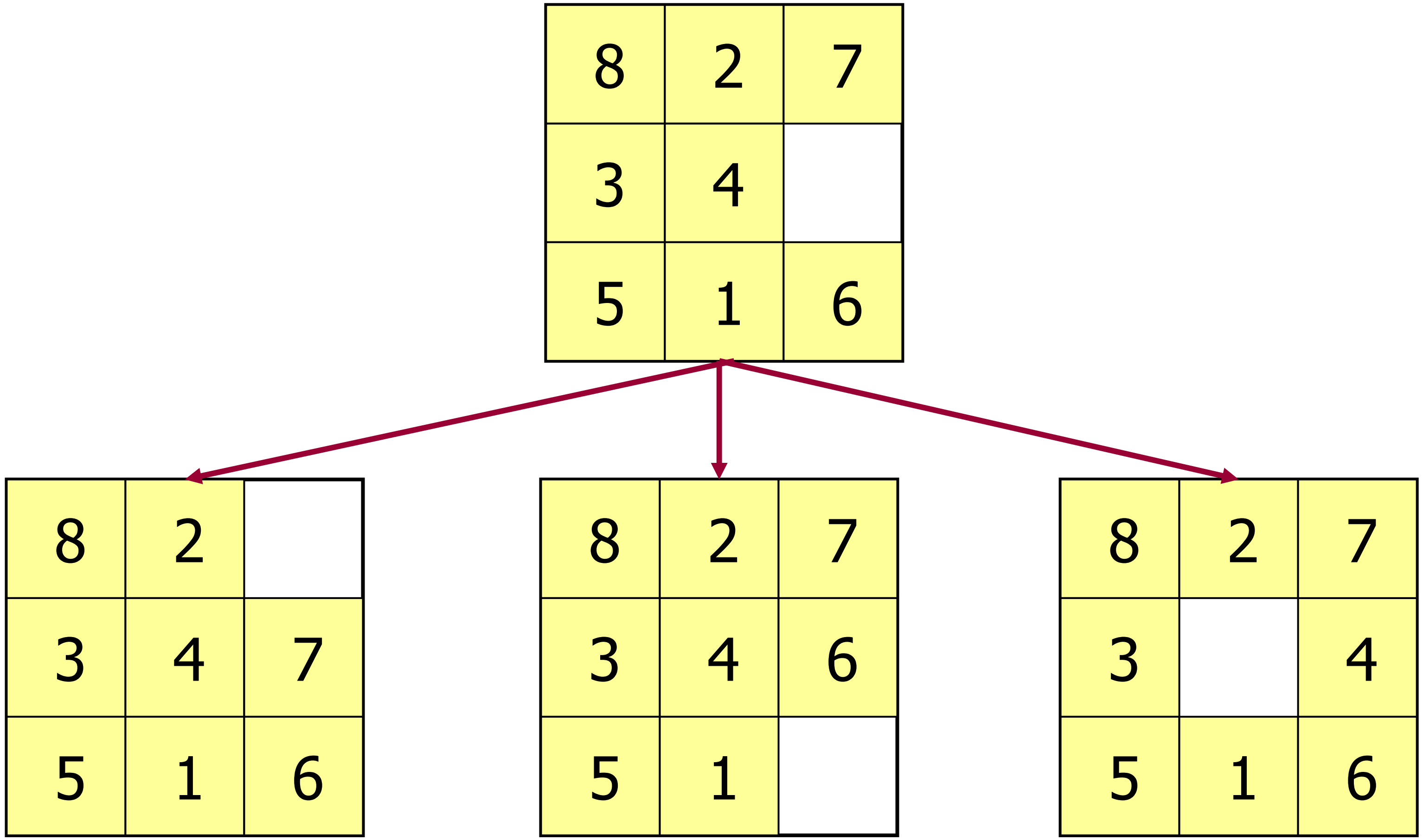
8	2	7
3	4	
5	1	6

8		2
3	4	7
5	1	6

	8	2
3	4	7
5	1	6

...

8-Puzzle: Successor Function



15-Puzzle

Introduced in 1878 by Sam Loyd, who dubbed himself "America's greatest puzzle-expert"

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

15-Puzzle 1.3 trillion states

Sam Loyd offered $1,000 of his own money to the first person who would solve the following problem:

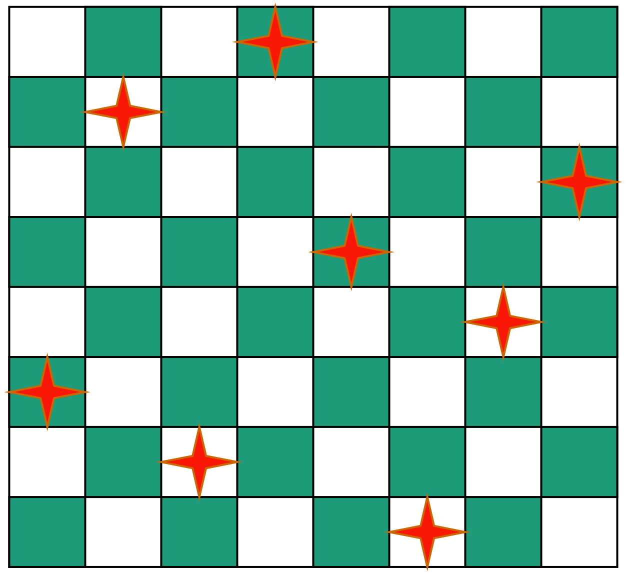
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

?

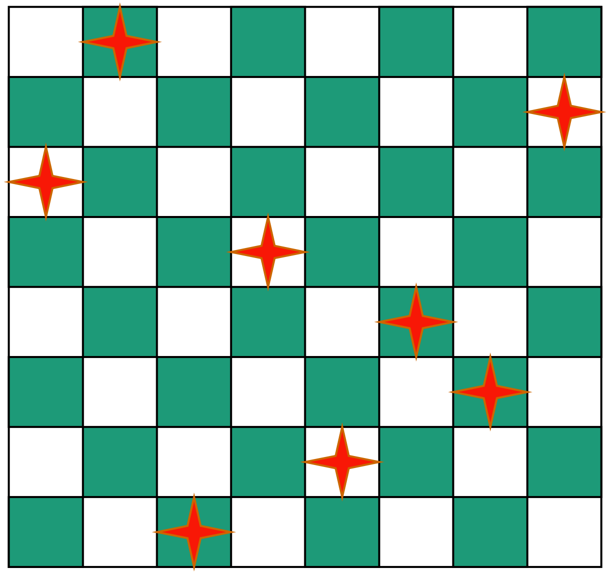
1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Example problem 4 :8-Queens Problem

Place 8 queens in a chessboard so that no two queens are in the same row, column, or diagonal.



A solution



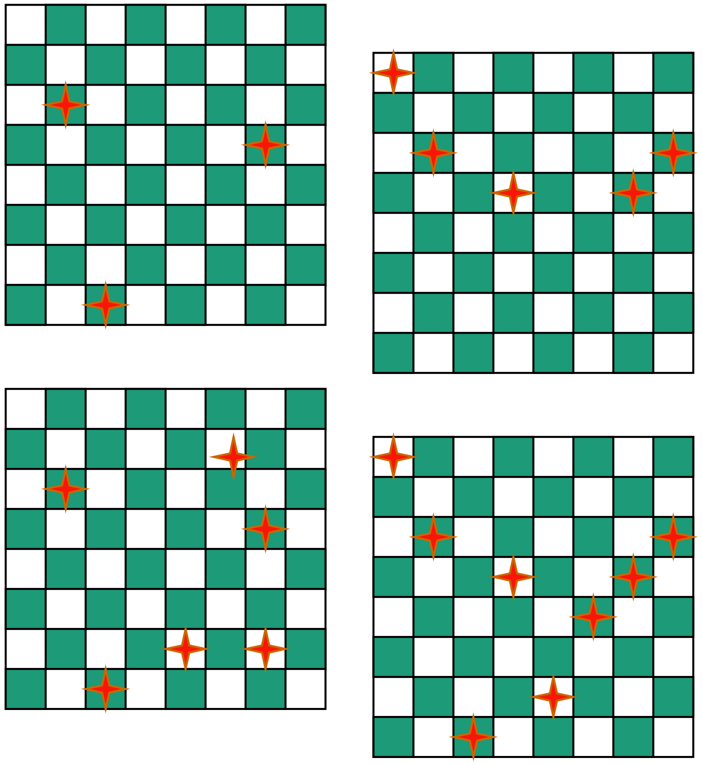
Not a solution

Problem Formulation

Two types of Formulation

- An **incremental formulation**
  - involves operators that *augment* the state description, starting with an empty state;
  - for the 8-queens problem, this means that each action adds a queen to the state.
  - $64 \times 63 \times \ldots \times 57 = 1.8 \times 10^{14}$

Incremental formulation



- **States:** all arrangements of 0, 1, 2, ..., or 8 queens on the board
- **Initial state:** 0 queen on the board
- **Successor function:** each of the successors is obtained by adding one queen in an empty square
- **Arc cost:** irrelevant
- **Goal test:** 8 queens are on the board, with no two of them attacking each other

$64 \text{x} 63 \text{x} \ldots \text{x} 57 = 1.8 \times 10^{14}$

Complete-state Formulation

- A **complete-state formulation**
  - starts with all 8 queens on the board and moves them around.
  - In either case, the path cost is of no interest because only the final state counts.
  - 2057 states
  - N=100, 10^{400}->10^{52}
- But techniques exist to solve n-queens problems efficiently for large values of n

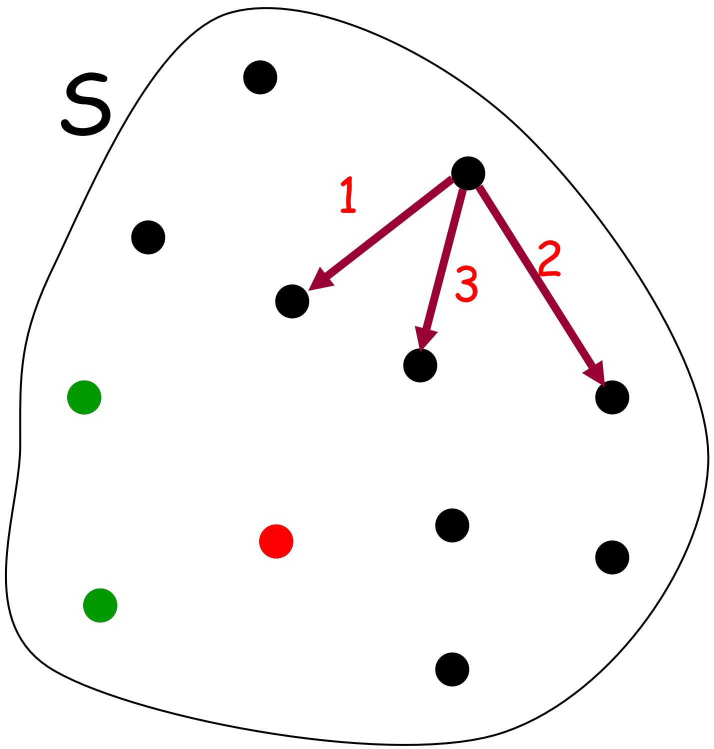
  They exploit the fact that there are many solutions well distributed in the state space

Real-Word Problems

- Touring problem
- TSP problem
- VLSI Layout
- Robot navigation
- Automatic assembly sequencing
- protein design

Searching for Solution

Searching for Solutions



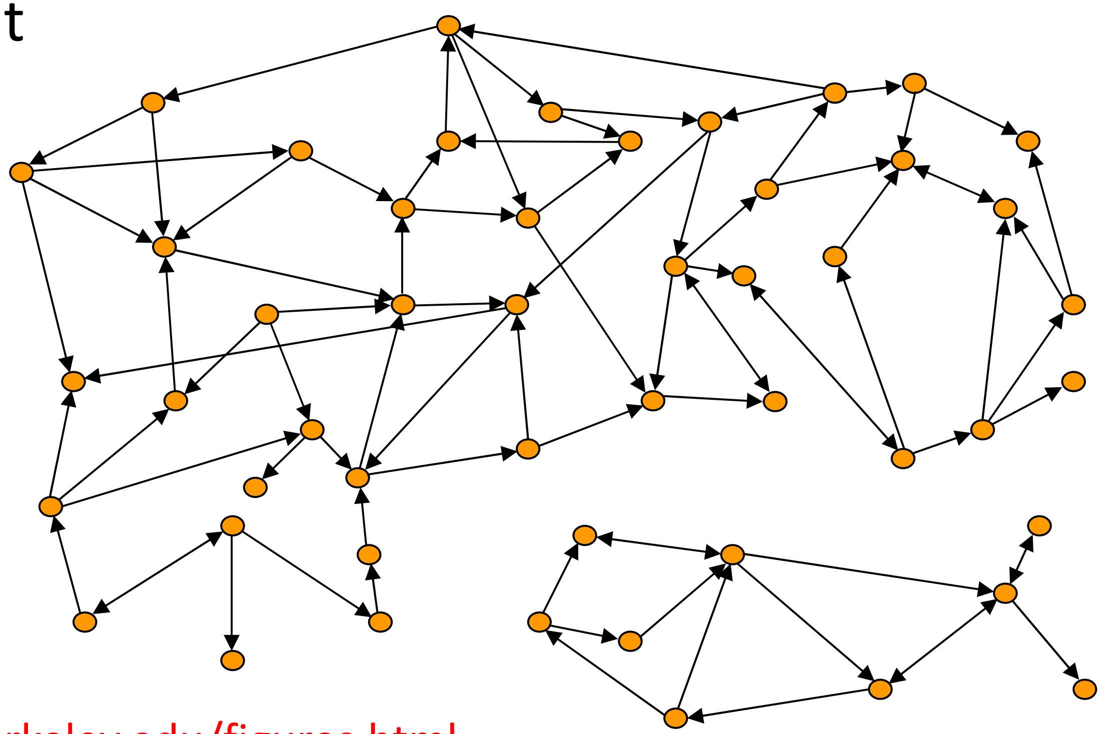
- State space S
- Successor function:

  $x \in S \rightarrow \text{SUCCESSORS}(x) \in 2^S$
- Arc cost
- Initial state s_0
- Goal test:

  $x \in S \rightarrow \text{GOAL?}(x) = T \text{ or } F$

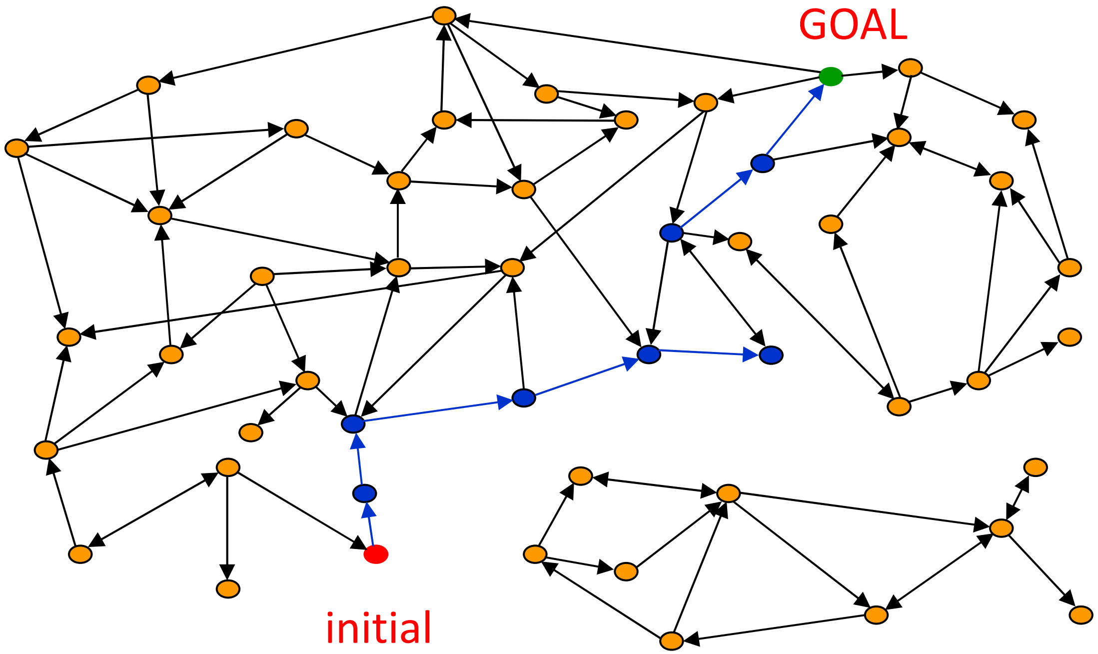
State Graph

- It is defined as follows:
  - Each state is represented by a distinct node
  - An arc connects a node s to a node s' if s' if s' $\in$ SUCCESSORS(s)
- The state graph may contain more than one connected component

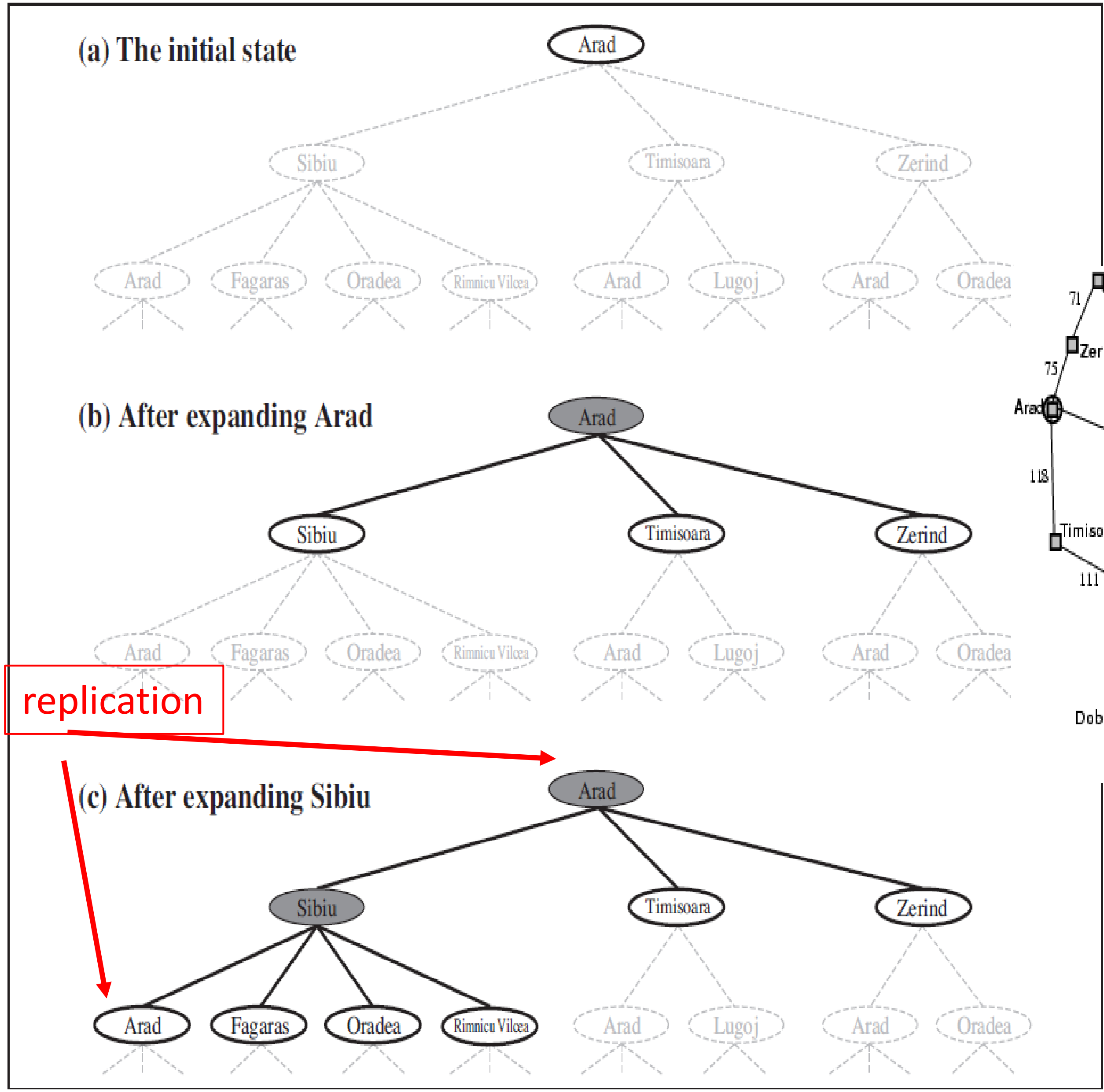


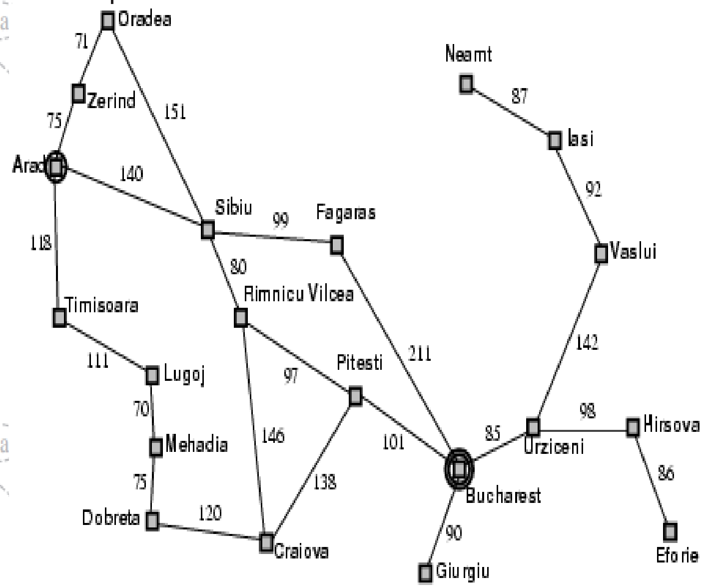
http://aima.cs.berkeley.edu/figures.html

Solution to the Search Problem



- A solution is a path connecting the initial to a goal node (any one)
- The cost of a path is the sum of the edge costs along this path
- An optimal solution is a solution path of minimum cost
- There might be no solution !





http://aima.cs.berkeley.edu/figures.html

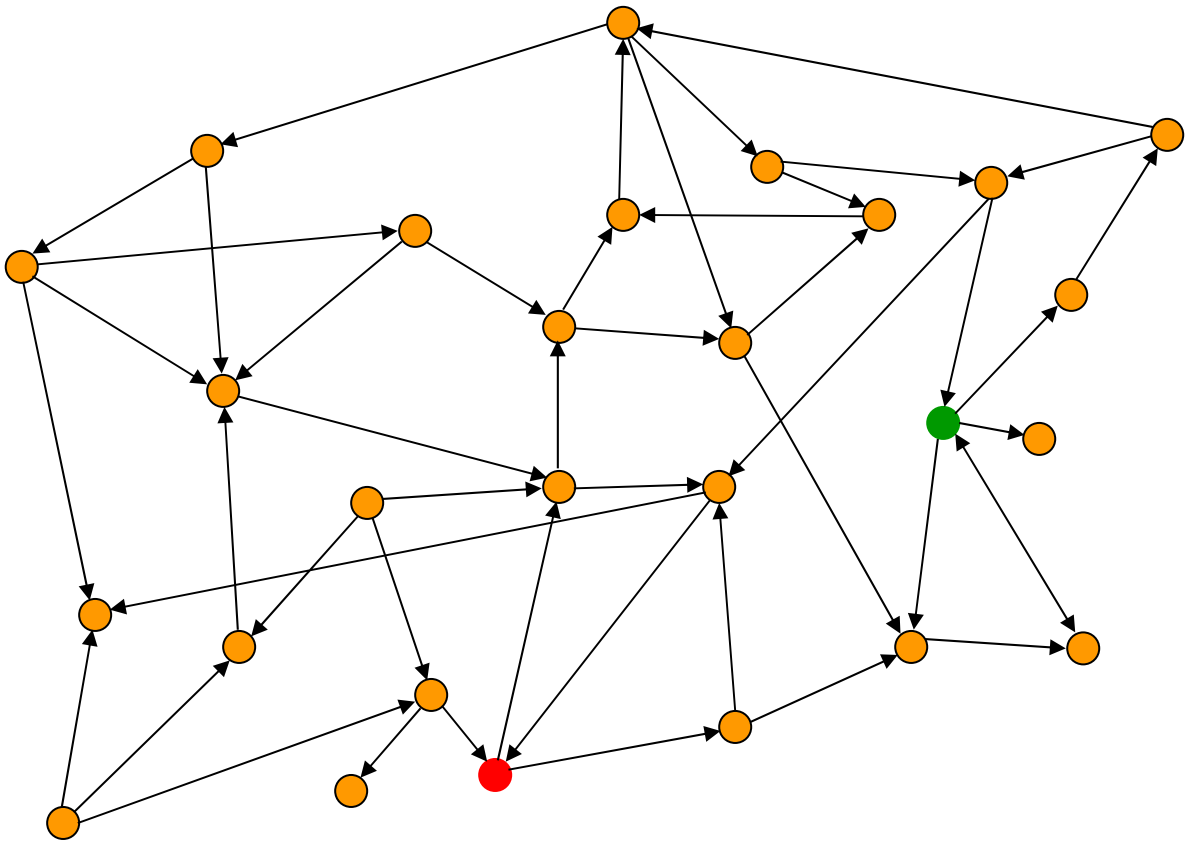
Tree-Search

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

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

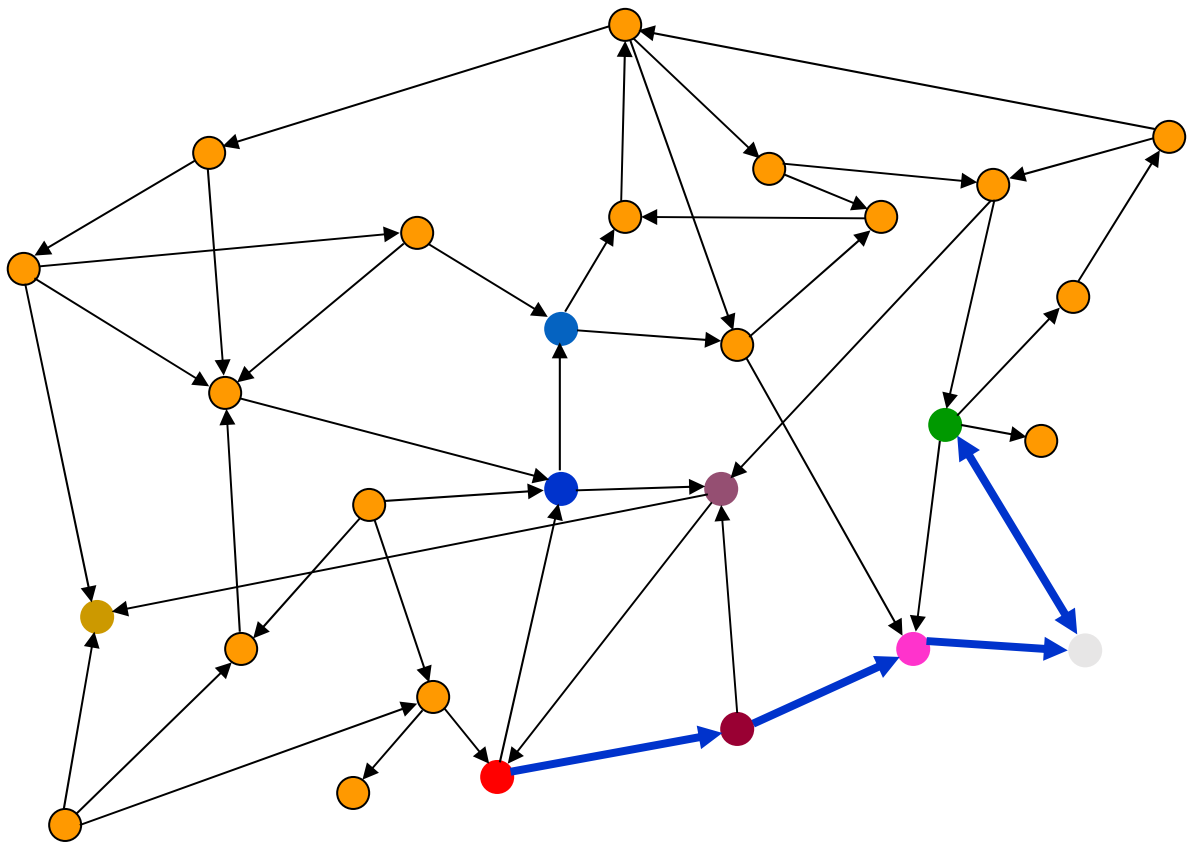
http://aima.cs.berkeley.edu/figures.html

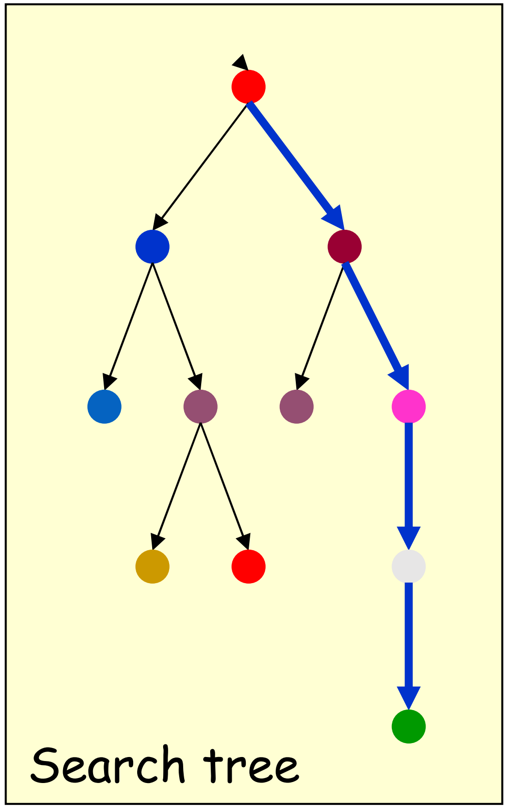
Searching the State Space



- Often it is not feasible to build a complete representation of the state graph
- A problem solver must construct a solution by exploring a small portion of the graph

Searching the State Space





Simple Problem-Solving-Agent Algorithm

1. s_0 ← sense/read initial state
2. GOAL? ← select/read goal test
3. Succ ← select/read **successor function**
4. solution ← **search**(s_0, GOAL?, Succ)
5. perform(solution)

Successor Function

- It implicitly represents all the actions that are feasible in each state.
- Only the results of the actions (the successor states) and their costs are returned by the function.

Path Cost

- An arc cost is a positive number measuring the “cost” of performing the action corresponding to the arc, e.g.:
  - 1 in the 8-puzzle example
  - expected time to merge two sub-assemblies
- We will assume that for any given problem the cost c of an arc always verifies: $c \geq \varepsilon > 0$, where ε is a constant

Goal State of 8-puzzle

- It may be explicitly described:

1	2	3
4	5	6
7	8	

- or partially described:

1	a	a
a	5	a
a	8	a

(“a” stands for “any”)

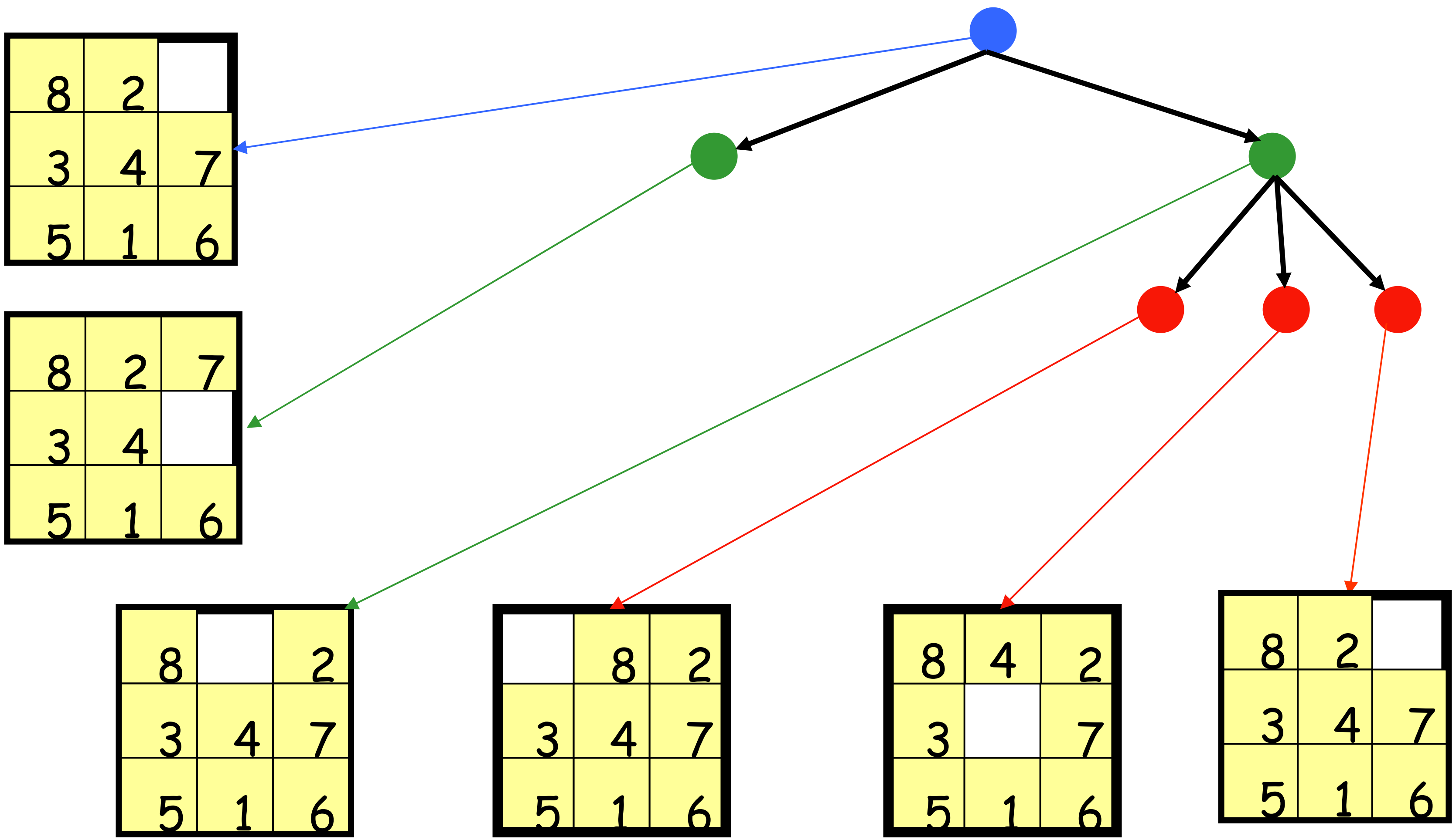
- or defined by a condition,
  e.g., the sum of every row, of every column, and of every diagonals equals 30

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

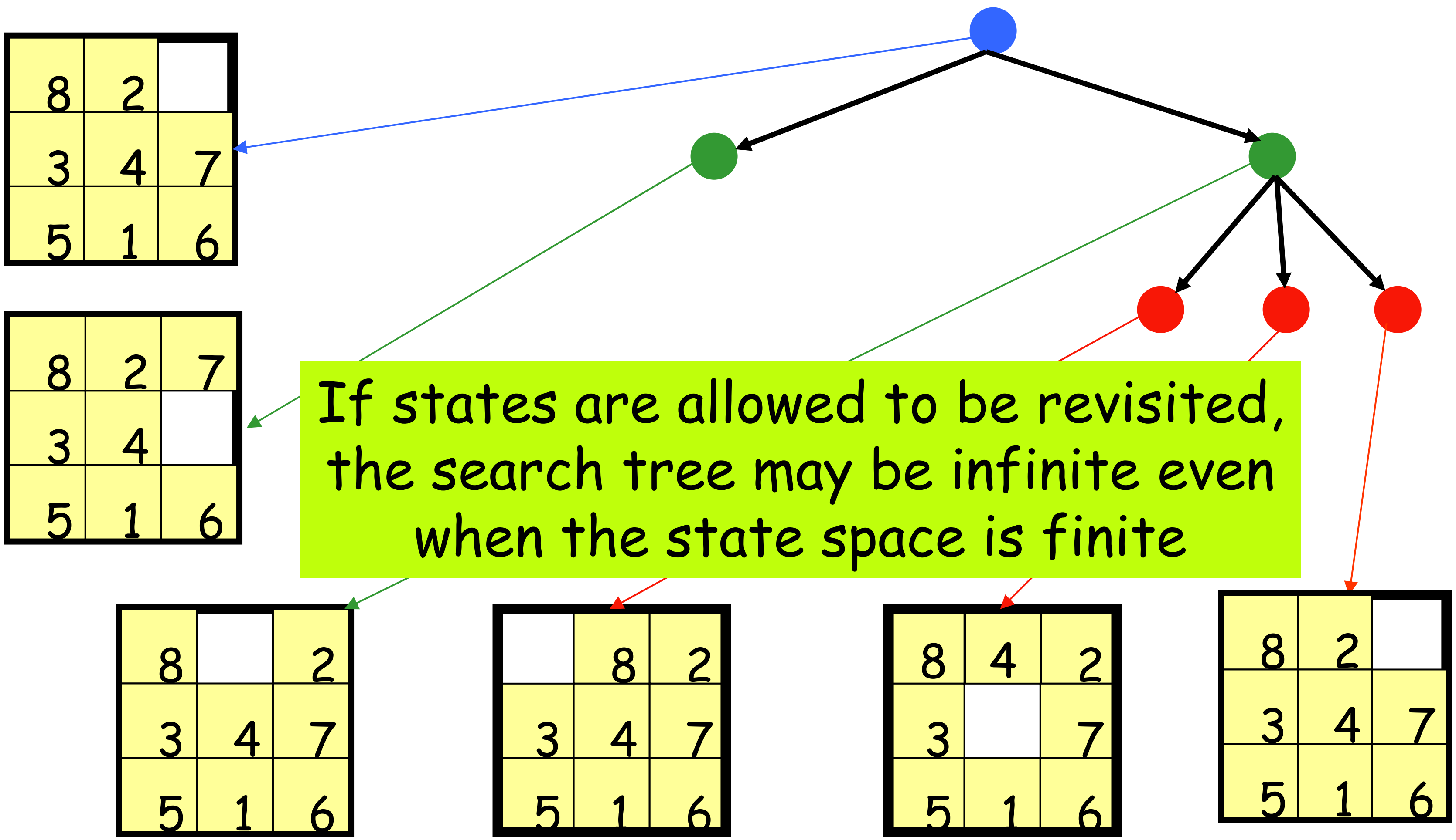
Basic Search Concepts

- Search tree
- Search node
- Node expansion
- Fringe of search tree
- Search strategy: At each stage it determines which node to expand

Search Nodes ≠ States

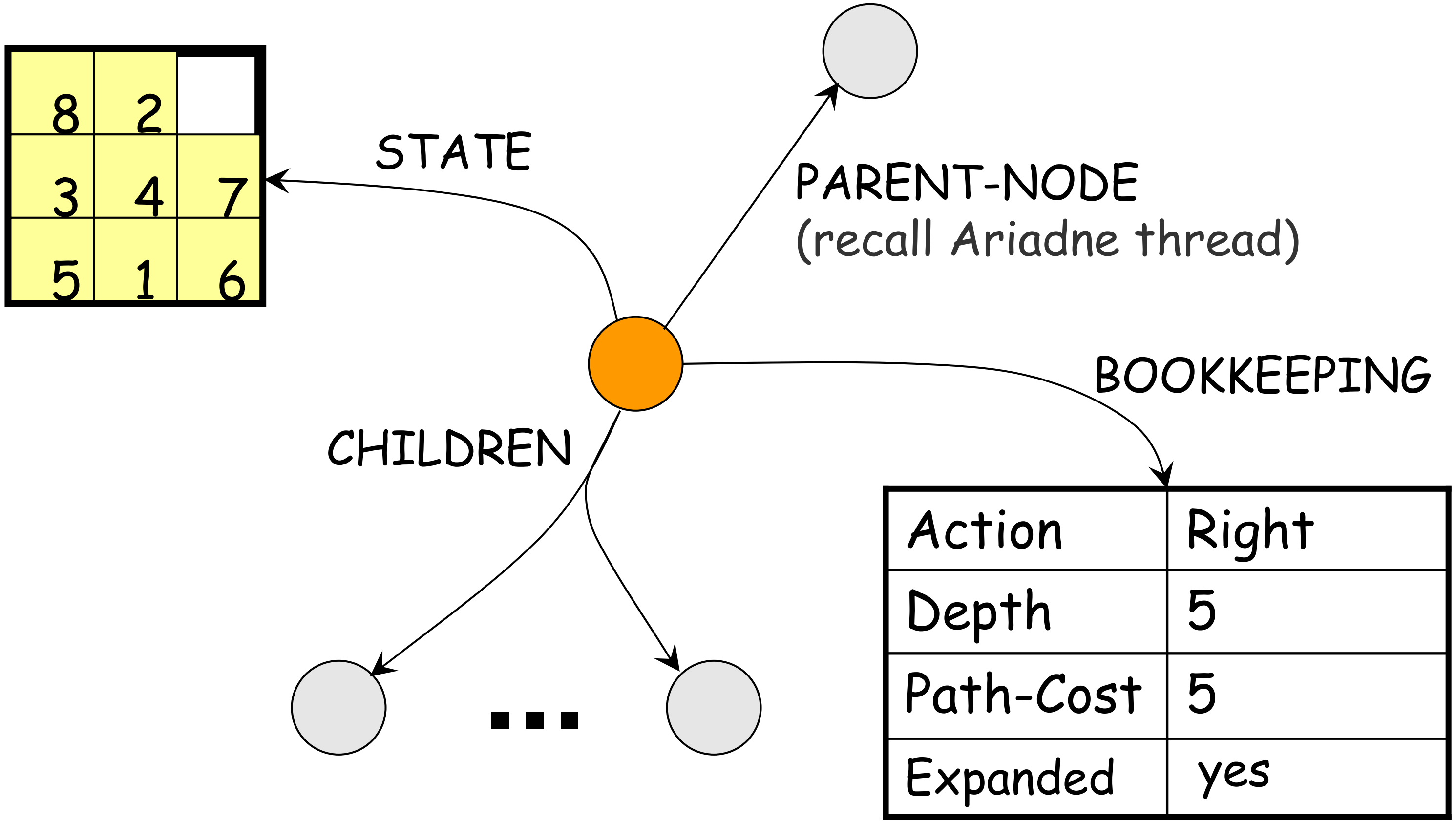


Search Nodes ≠ States



If states are allowed to be revisited, the search tree may be infinite even when the state space is finite

Data Structure of a Node



Depth of a node N = length of path from root to N

(Depth of the root = 0)

CHILD-NODE & expansion

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

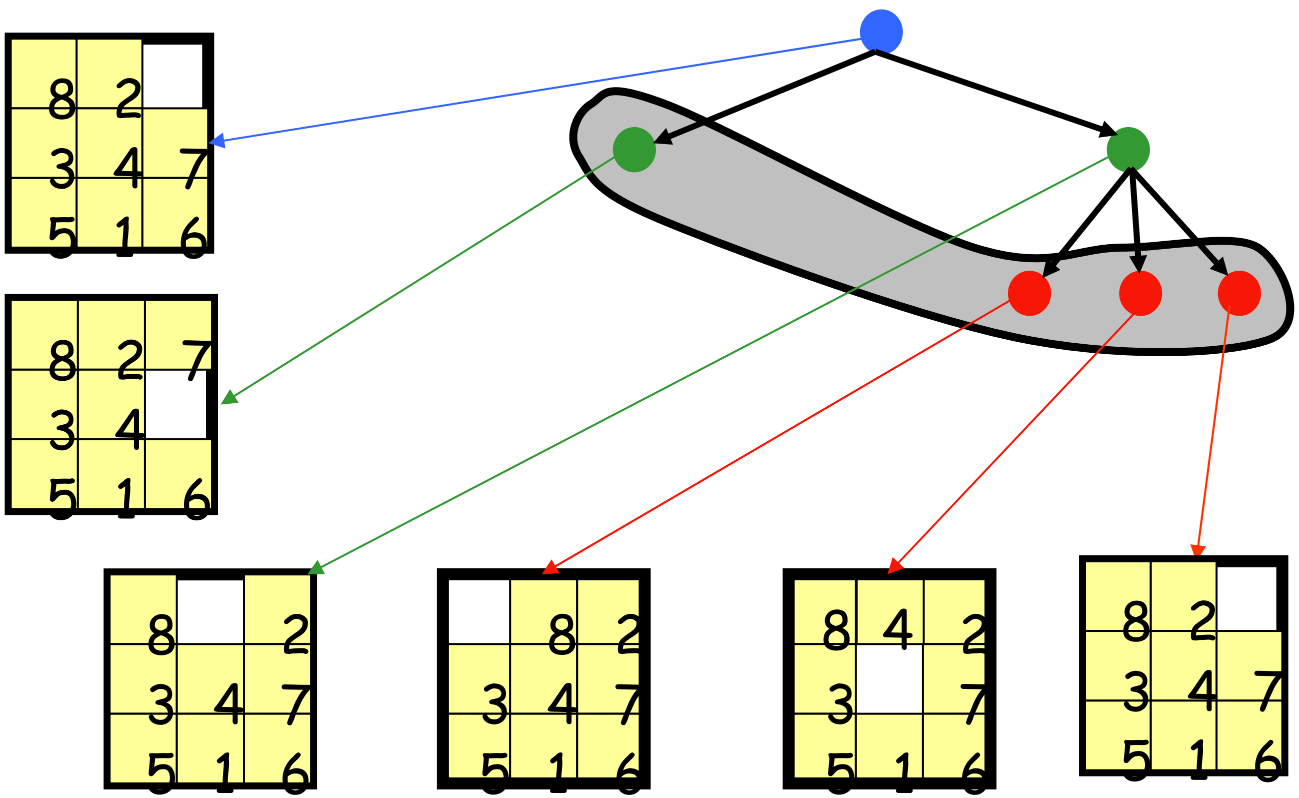
The expansion of a node N of the search tree consists of:

1) Evaluating the successor function on STATE(N)
2) Generating a child of N for each state returned by the function

http://aima.cs.berkeley.edu/figures.html

Fringe and Search Strategy

- The fringe is the set of all search nodes that haven't been expanded yet



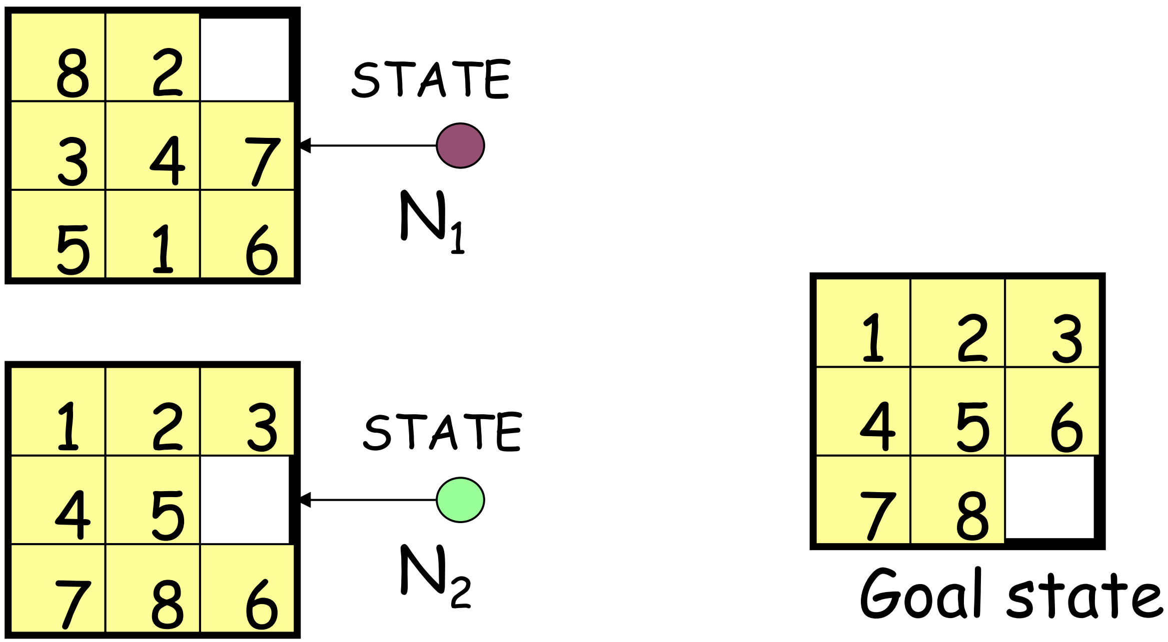
Is it identical to the set of leaves?

Search Algorithm

1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node, FRINGE)
3. Repeat:
   a. If empty(FRINGE) then return failure
   b. n ← REMOVE(FRINGE)
   c. s ← STATE(n)
   d. For every state s' in SUCCESSORS(s)
      i. Create a new node n' as a child of n
      ii. If GOAL?(s') then return path or goal state
      iii. INSERT(n', FRINGE)

Blind vs. Heuristic Strategies

- Blind (or un-informed) strategies
  - do not exploit state descriptions to select which node to expand next
  - N_1 and N_2 are just two nodes (at some depth in the search tree)
- Heuristic (or informed) strategies
  - exploits state descriptions to select the “most promising” node to expand
  - counting the number of misplaced tiles, N_2 is more promising than N_1



Important Remark

- Some search problems, such as the $(n^2\text{-}1)$-puzzle, are NP-hard
  - 8-puzzle → 9! = 362,880 states (0.036 sec)
  - 15-puzzle → 16! ~ 1.3×10^{12} states (< 4 hours)
  - 24-puzzle → 25! ~ 10^{25} states (> 10^9 years)

    But only half of these states are reachable from any given state
- One can't expect to solve all instances of such problems in less than exponential time
- One may still strive to solve each instance as efficiently as possible

Performance of Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

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

Uninformed Search Strategies Blind Strategies

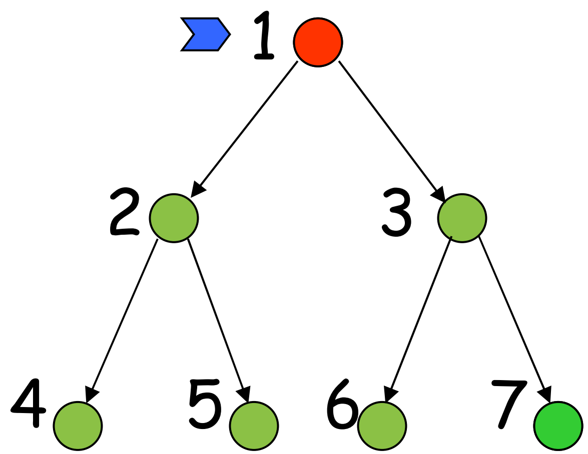
- Breadth-first
  - Bidirectional

- Depth-first
  - Depth-limited
  - Iterative deepening

breadth-first search (BFS)

Breadth-First Strategy

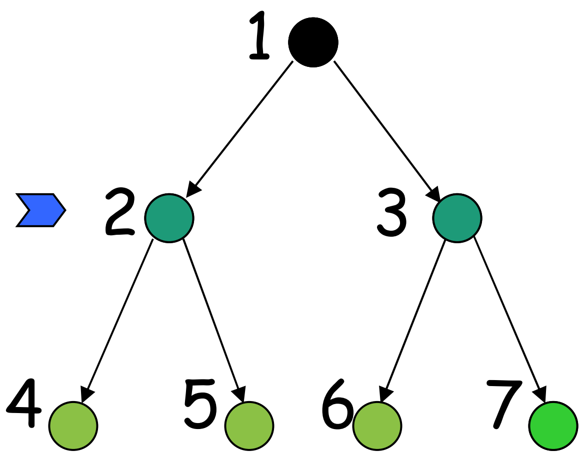
New nodes are inserted at the end of FRINGE



FRINGE = (1)

Breadth-First Strategy

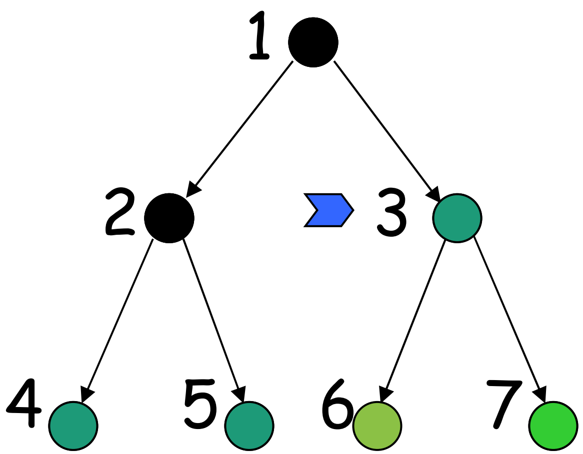
New nodes are inserted at the end of FRINGE



FRINGE = (2, 3)

Breadth-First Strategy

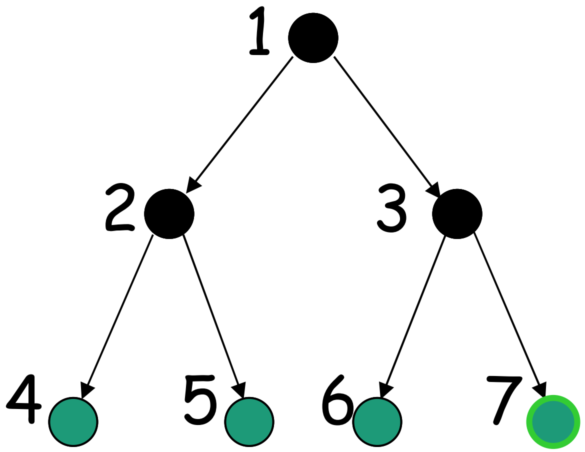
New nodes are inserted at the end of FRINGE



FRINGE = (3, 4, 5)

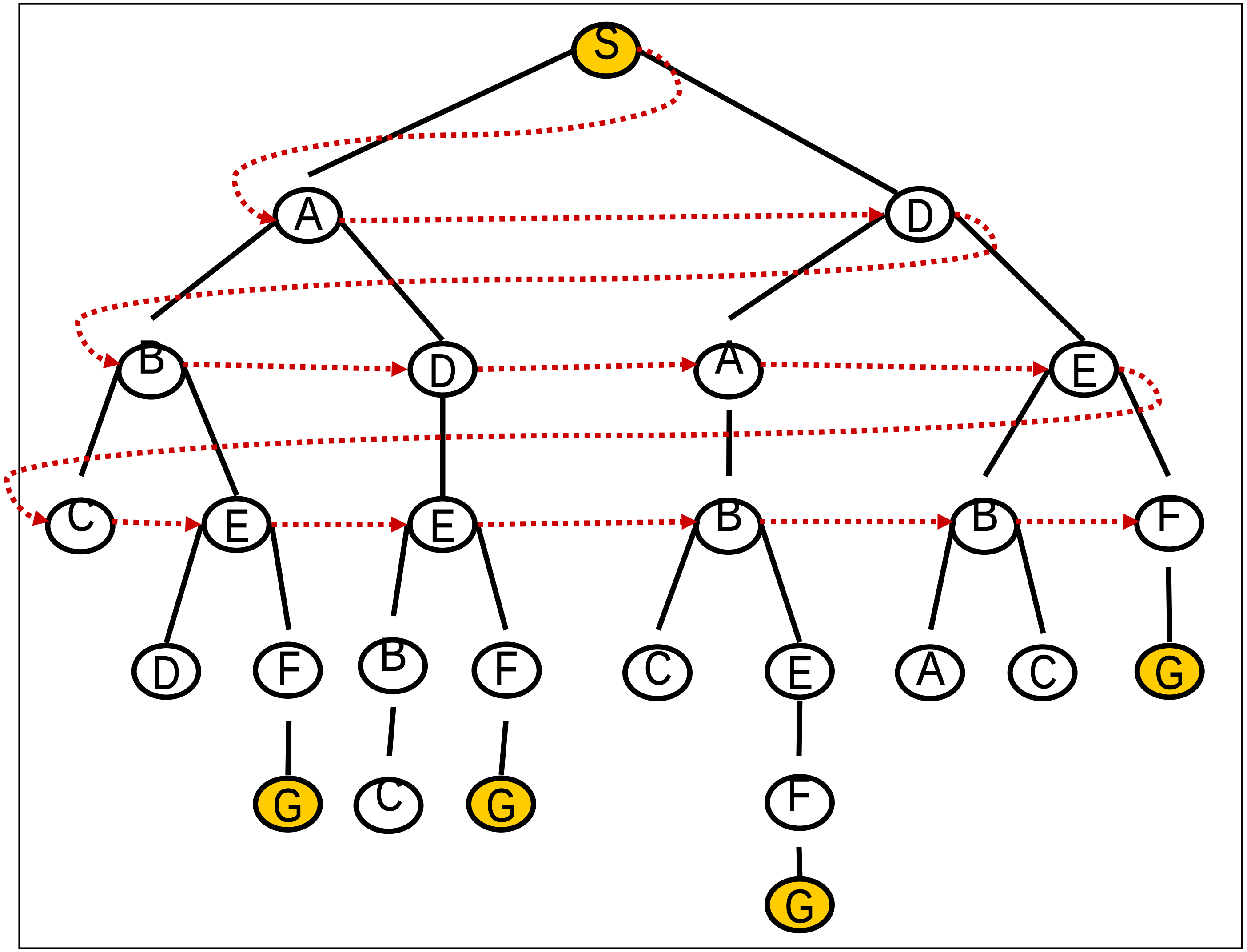
Breadth-First Strategy

New nodes are inserted at the end of FRINGE



FRINGE = (4, 5, 6, 7)

Breadth-first search



Move downwards, level by level, until goal is reached.

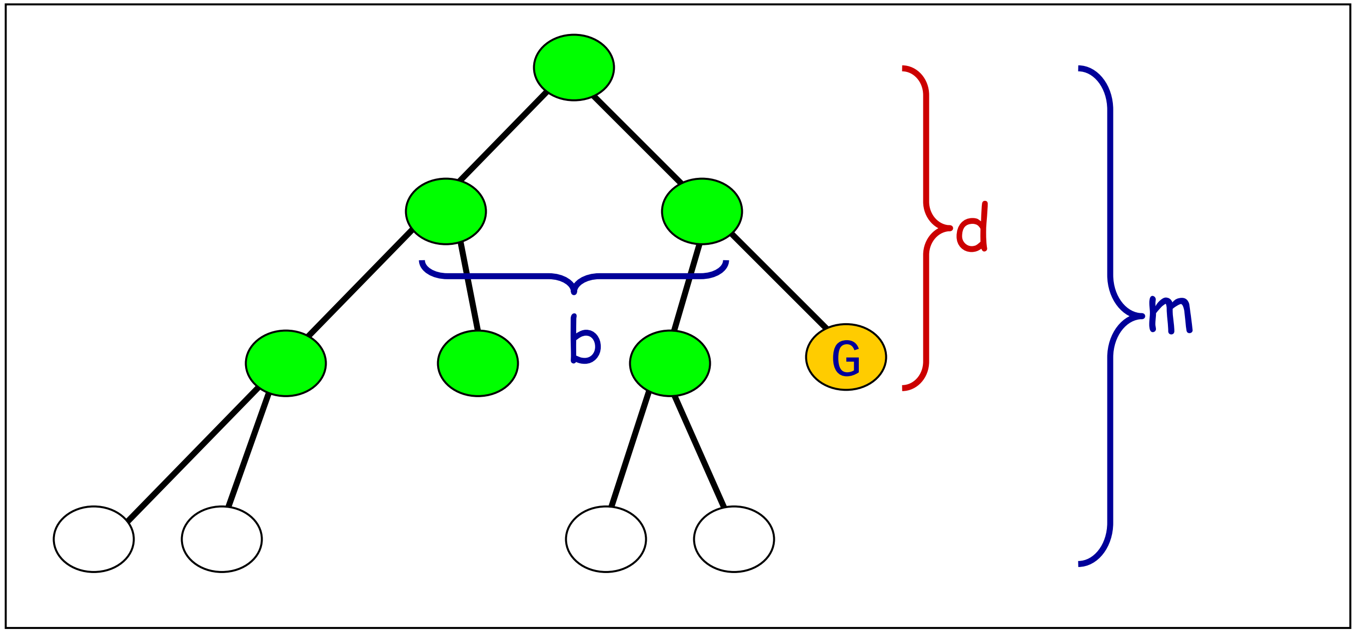
Breadth-first search

```
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)
```

http://aima.cs.berkeley.edu/figures.html

Time complexity of breadth-first search

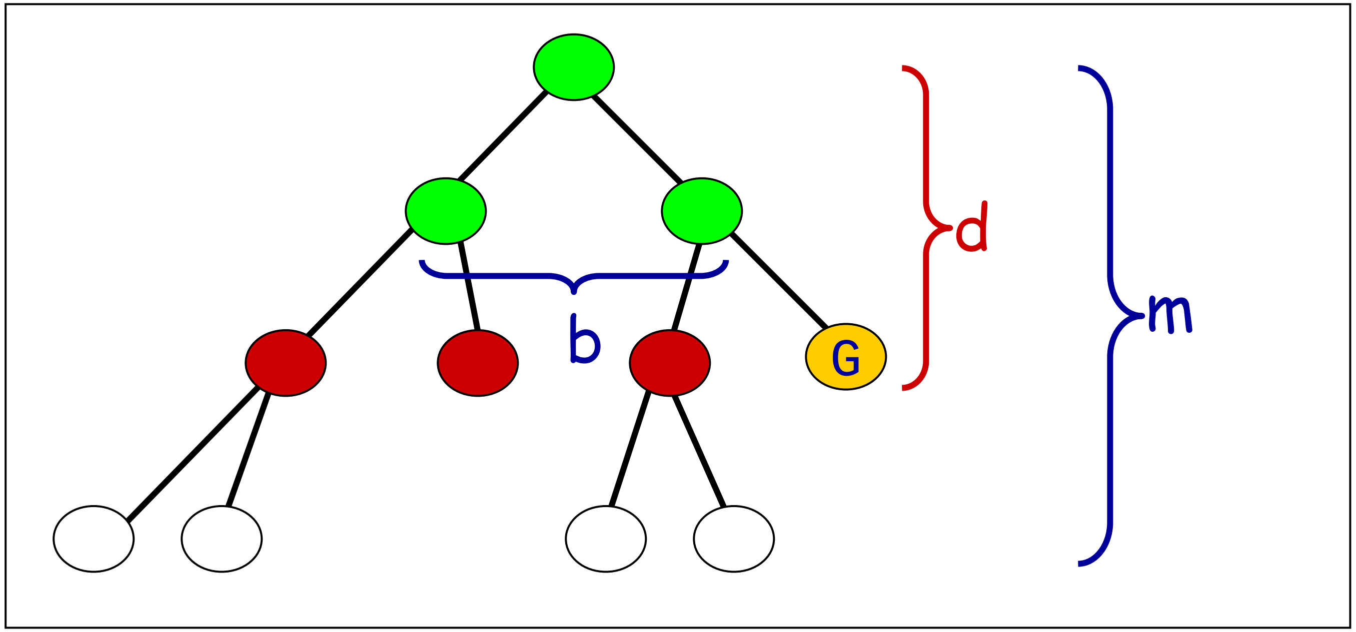
- If a goal node is found on depth d of the tree, all nodes up till that depth are created.

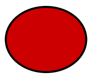


- Thus: $O(b^d)$

Space complexity of breadth-first

- Largest number of nodes in QUEUE is reached on the level d of the goal node.



- QUEUE contains all  and  nodes. (Thus: 4) .
- Number of nodes generated:

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

Time and Memory Requirements

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim 10^6$	1 sec	100 Mb
8	$\sim 10^8$	100 sec	10 Gbytes
10	$\sim 10^{10}$	2.8 hours	1 Tbyte
12	$\sim 10^{12}$	11.6 days	100 Tbytes
14	$\sim 10^{14}$	3.2 years	10,000 Tbytes

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

Code example BFS.py

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (len(self.graph))

        # Create a queue for BFS
        queue = []
```

```python
        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```

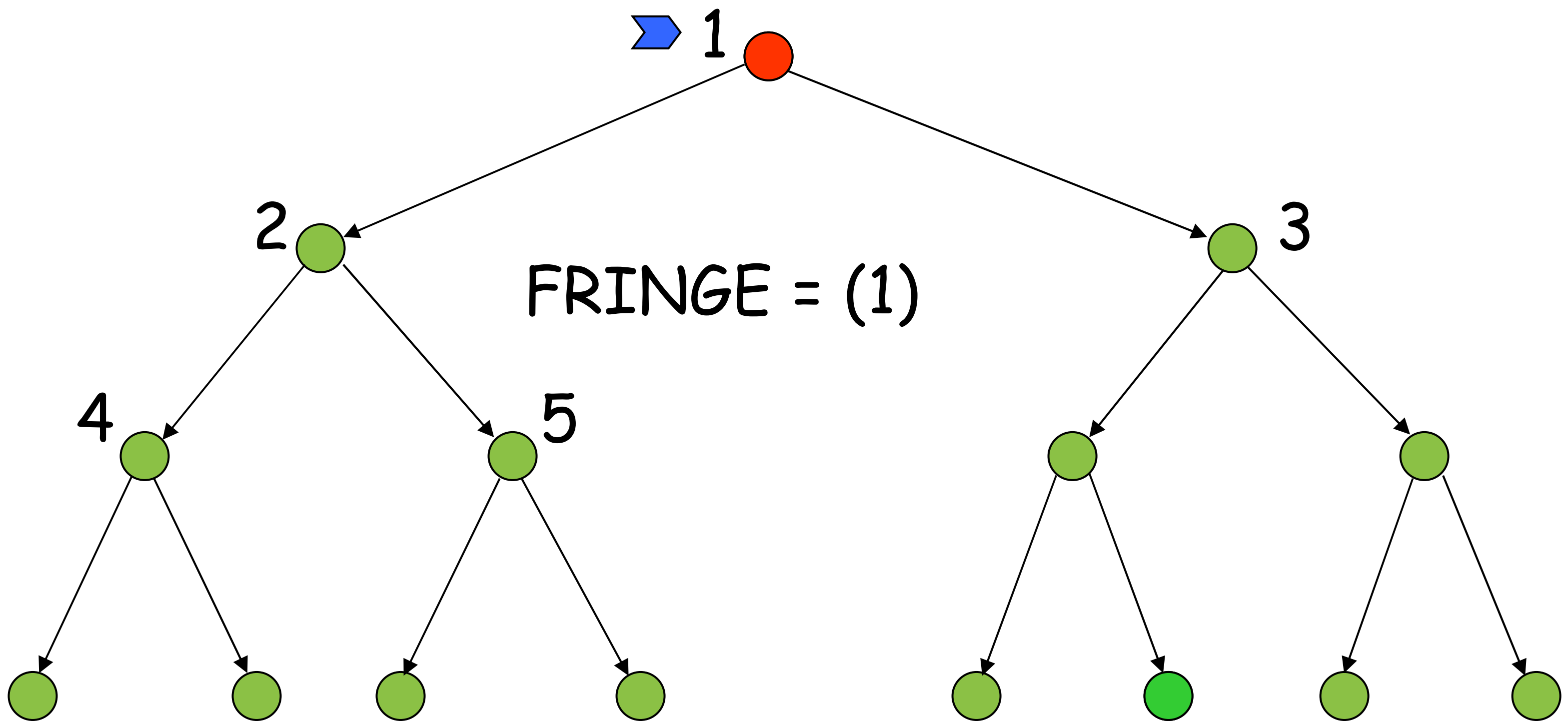
```python
# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
g.BFS(2)

# This code is contributed by Neelam Yadav
```

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Depth-First Strategy

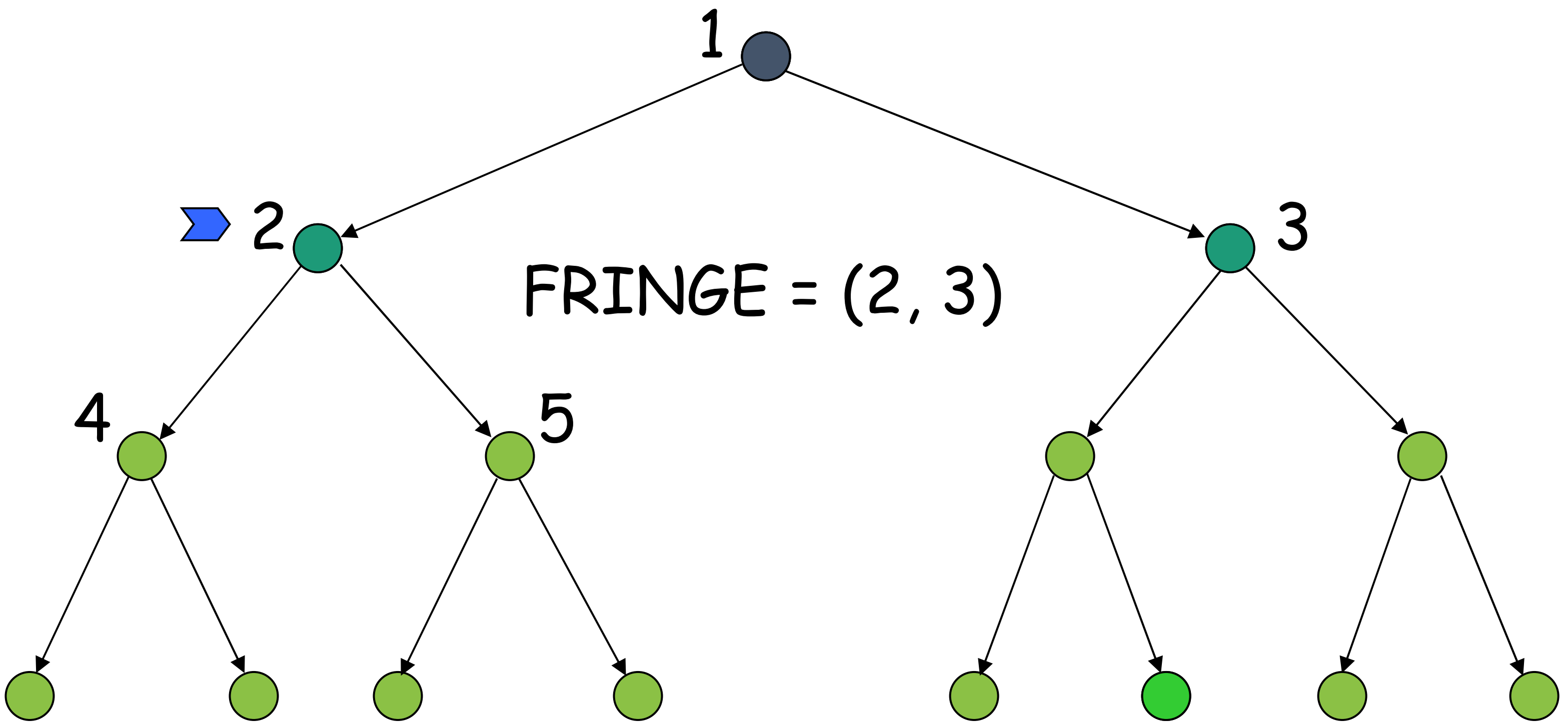
Depth-First Strategy

New nodes are inserted at the front of FRINGE



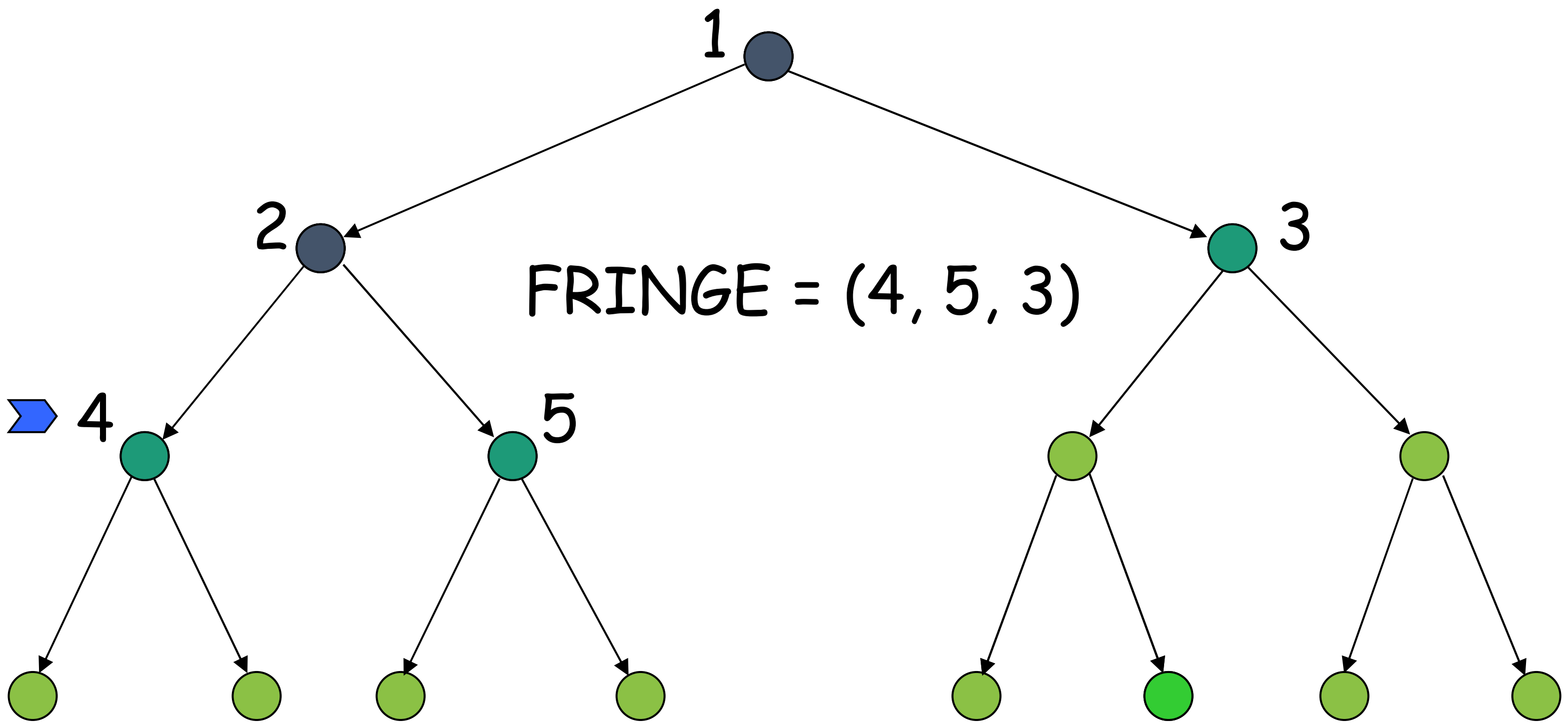
Depth-First Strategy

New nodes are inserted at the front of FRINGE



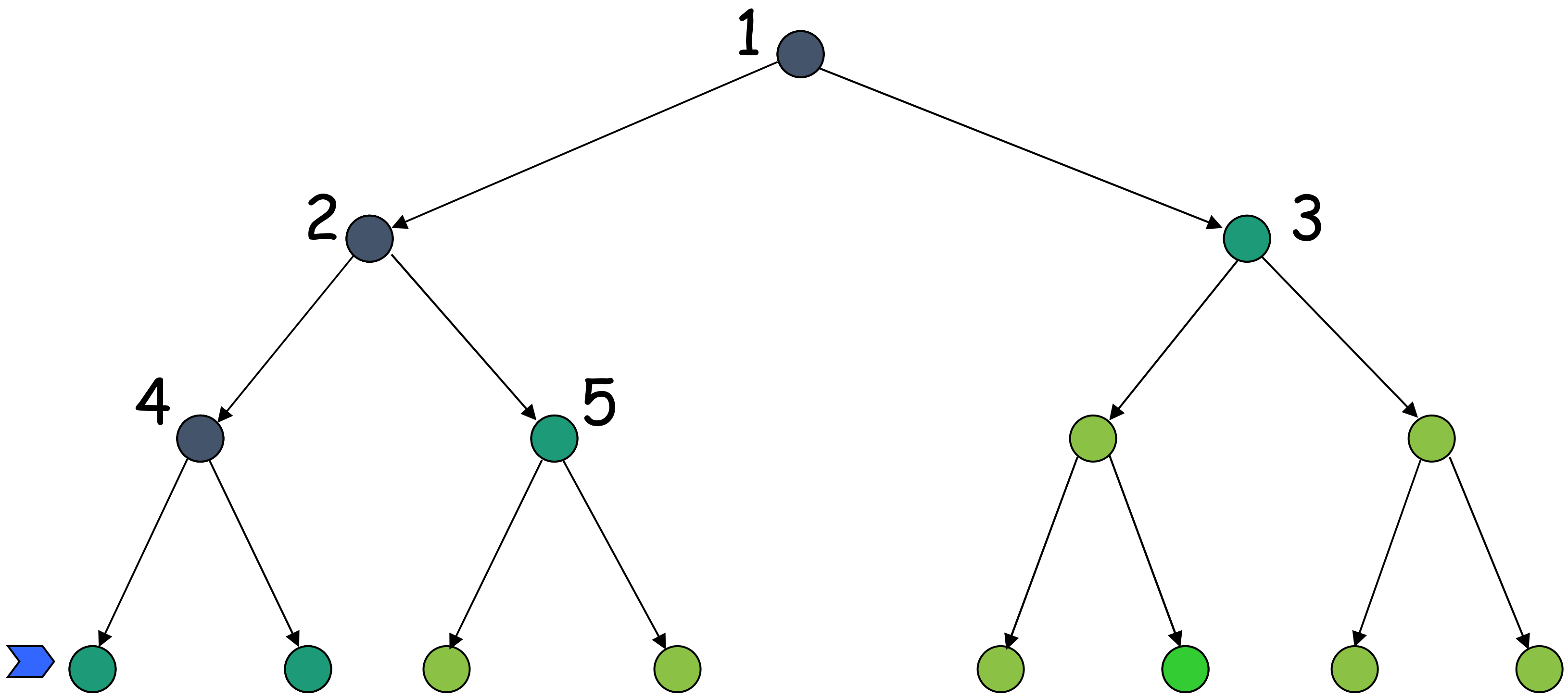
Depth-First Strategy

New nodes are inserted at the front of FRINGE



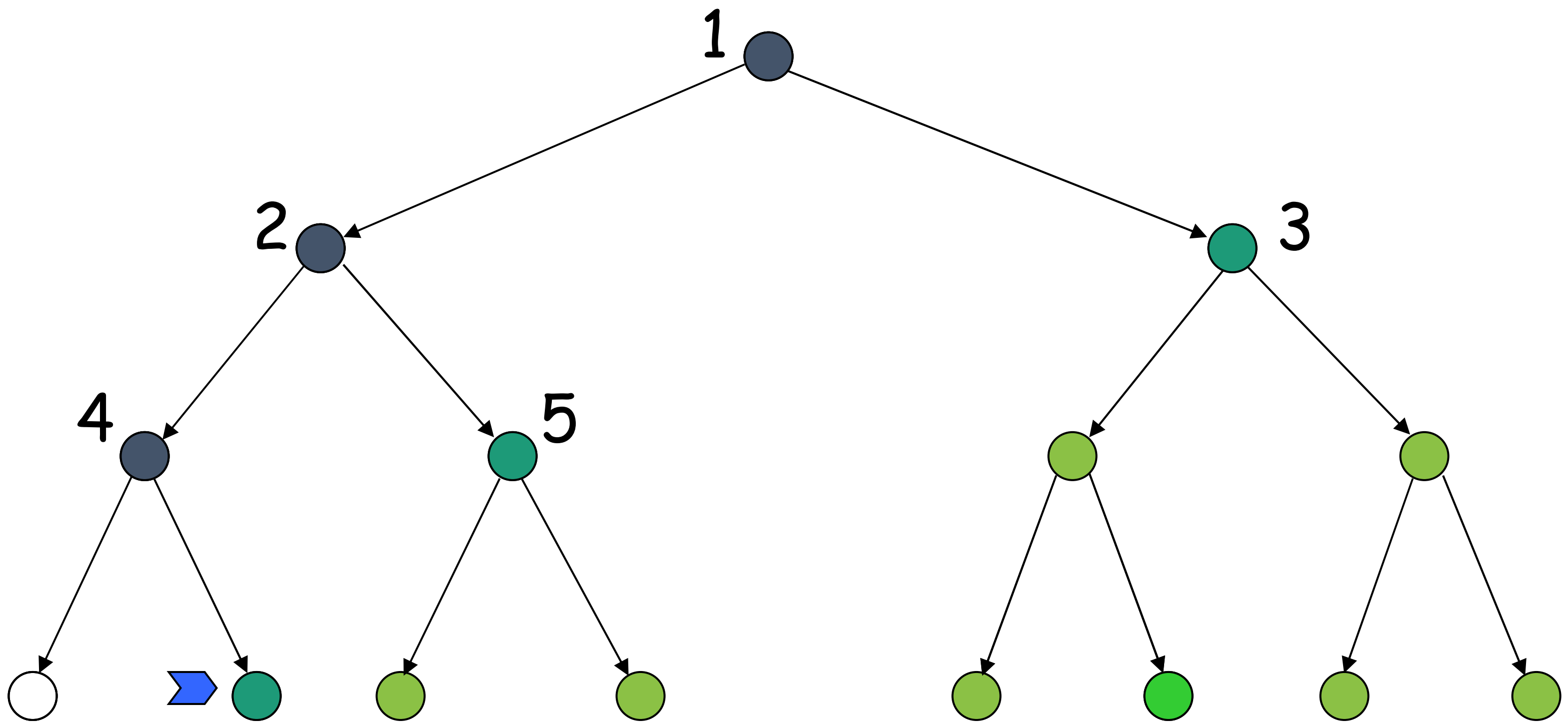
Depth-First Strategy

New nodes are inserted at the front of FRINGE



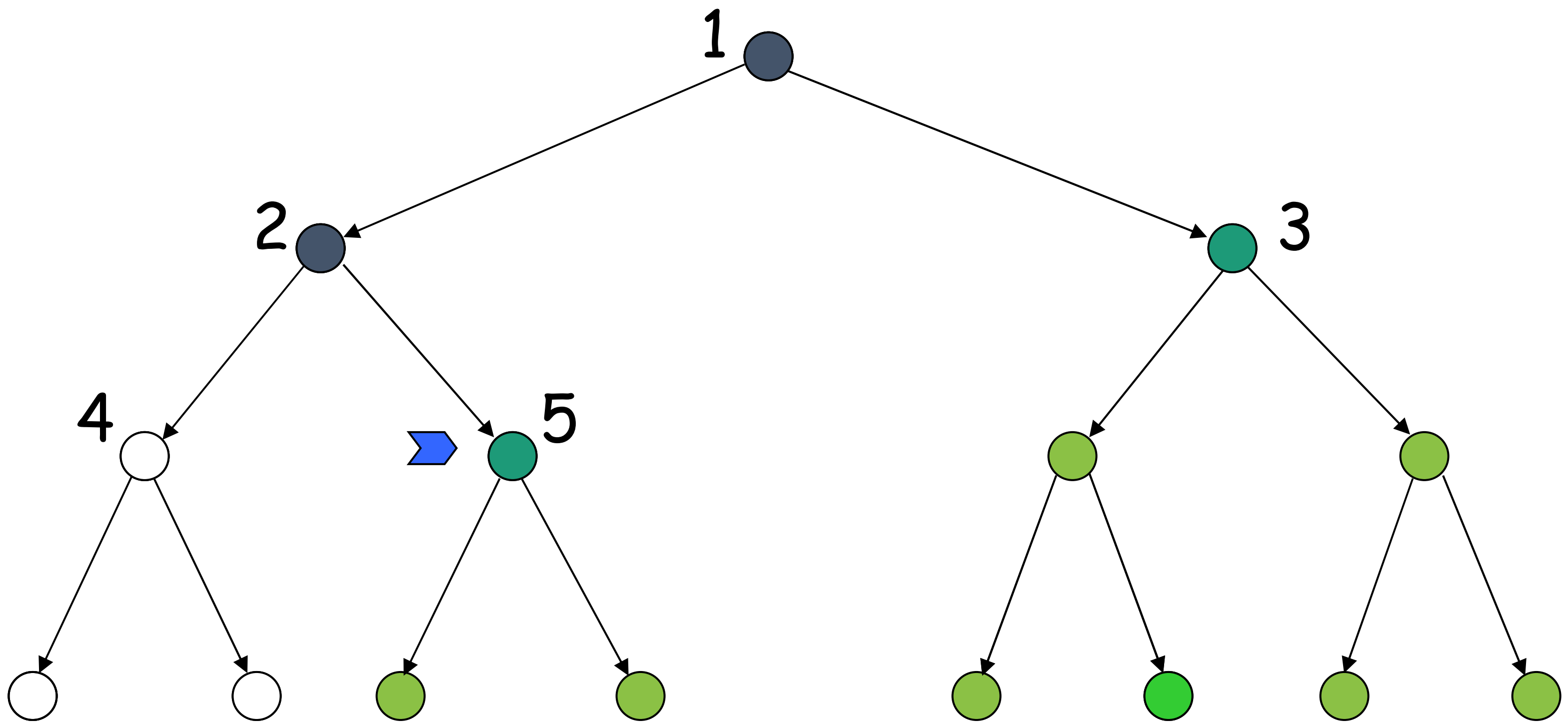
Depth-First Strategy

New nodes are inserted at the front of FRINGE



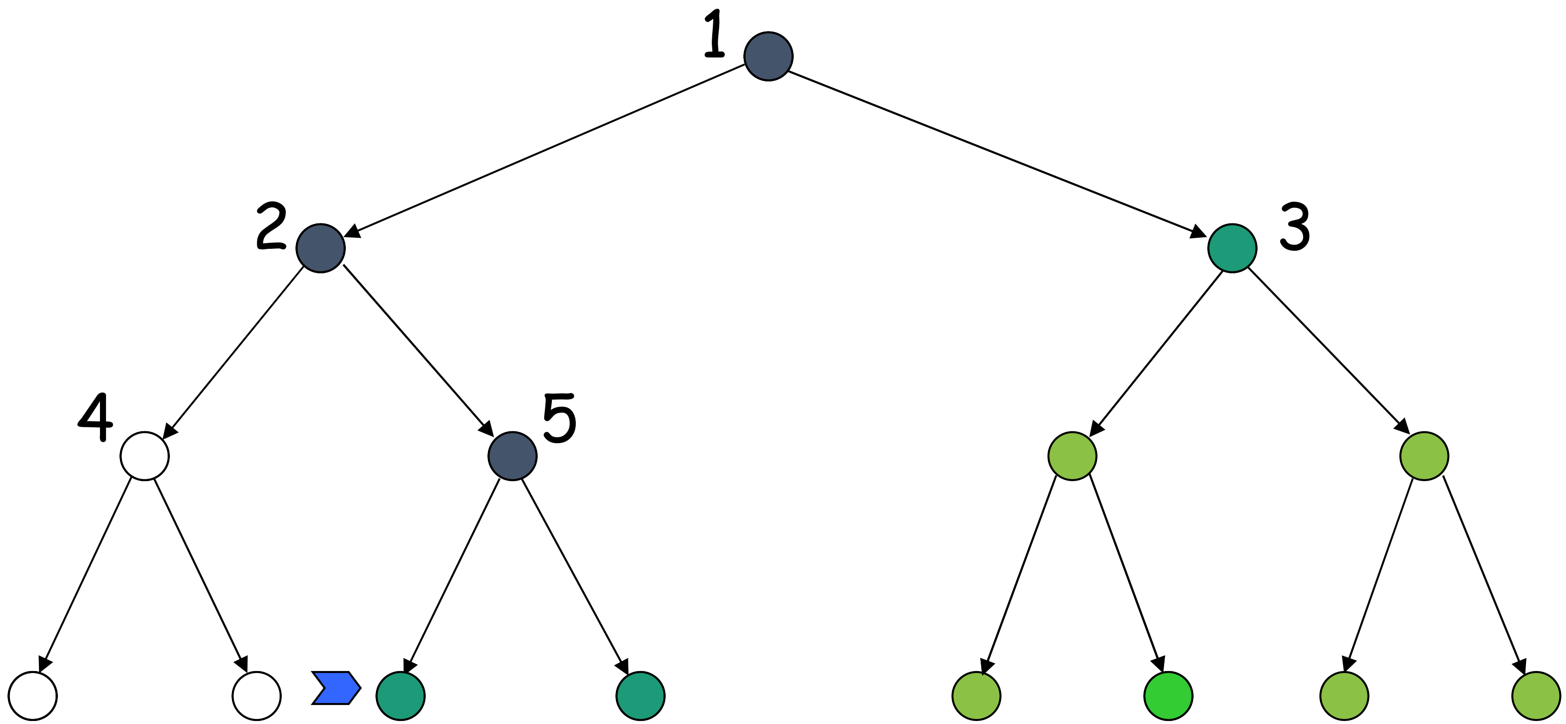
Depth-First Strategy

New nodes are inserted at the front of FRINGE



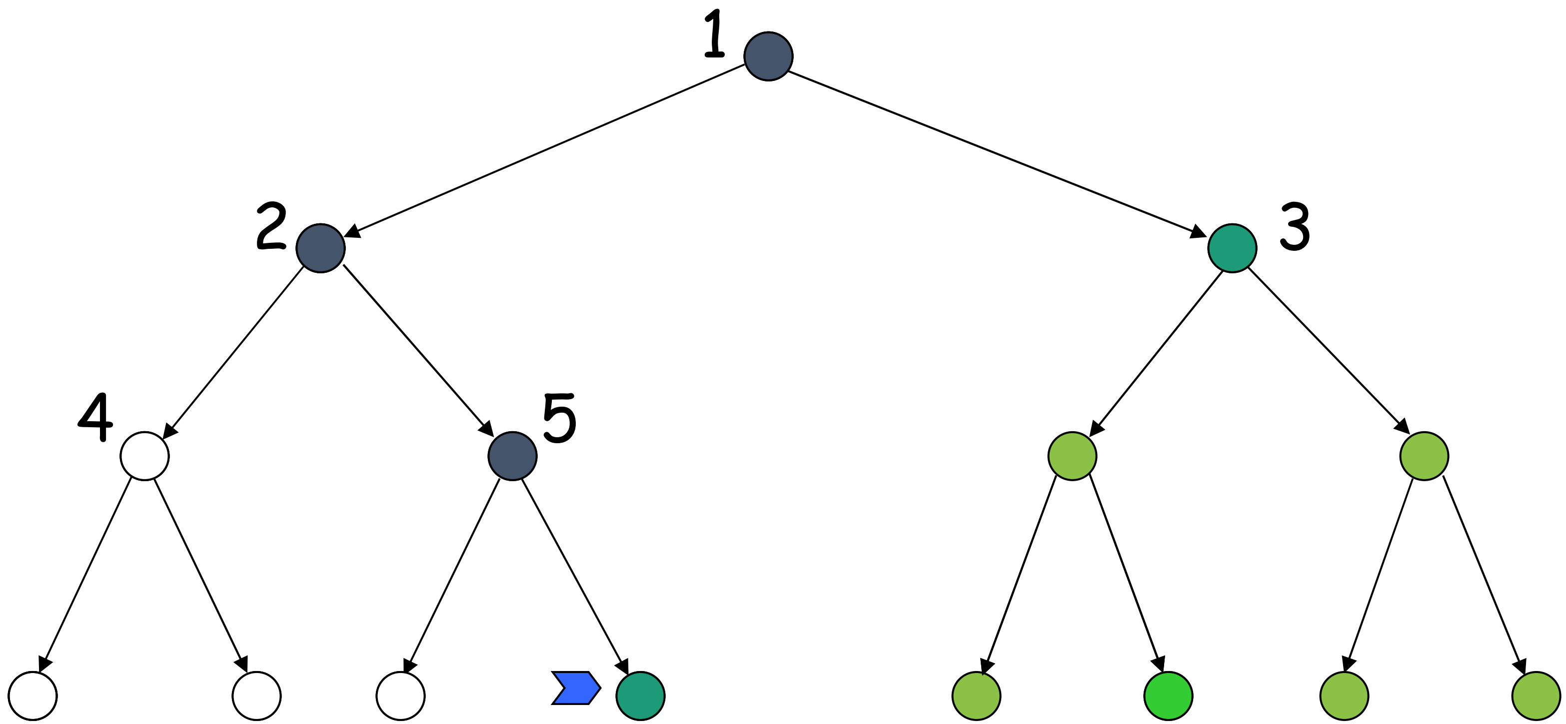
Depth-First Strategy

New nodes are inserted at the front of FRINGE



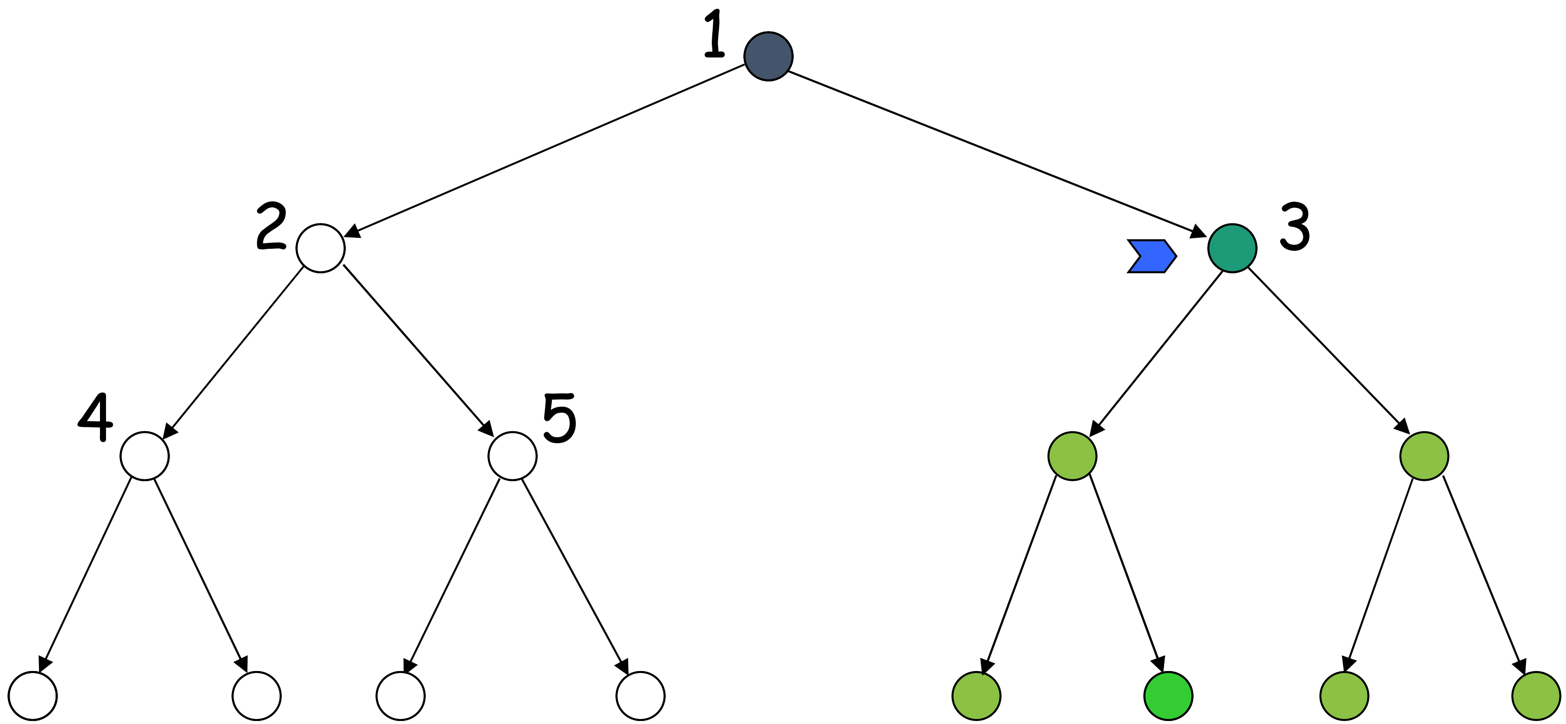
Depth-First Strategy

New nodes are inserted at the front of FRINGE



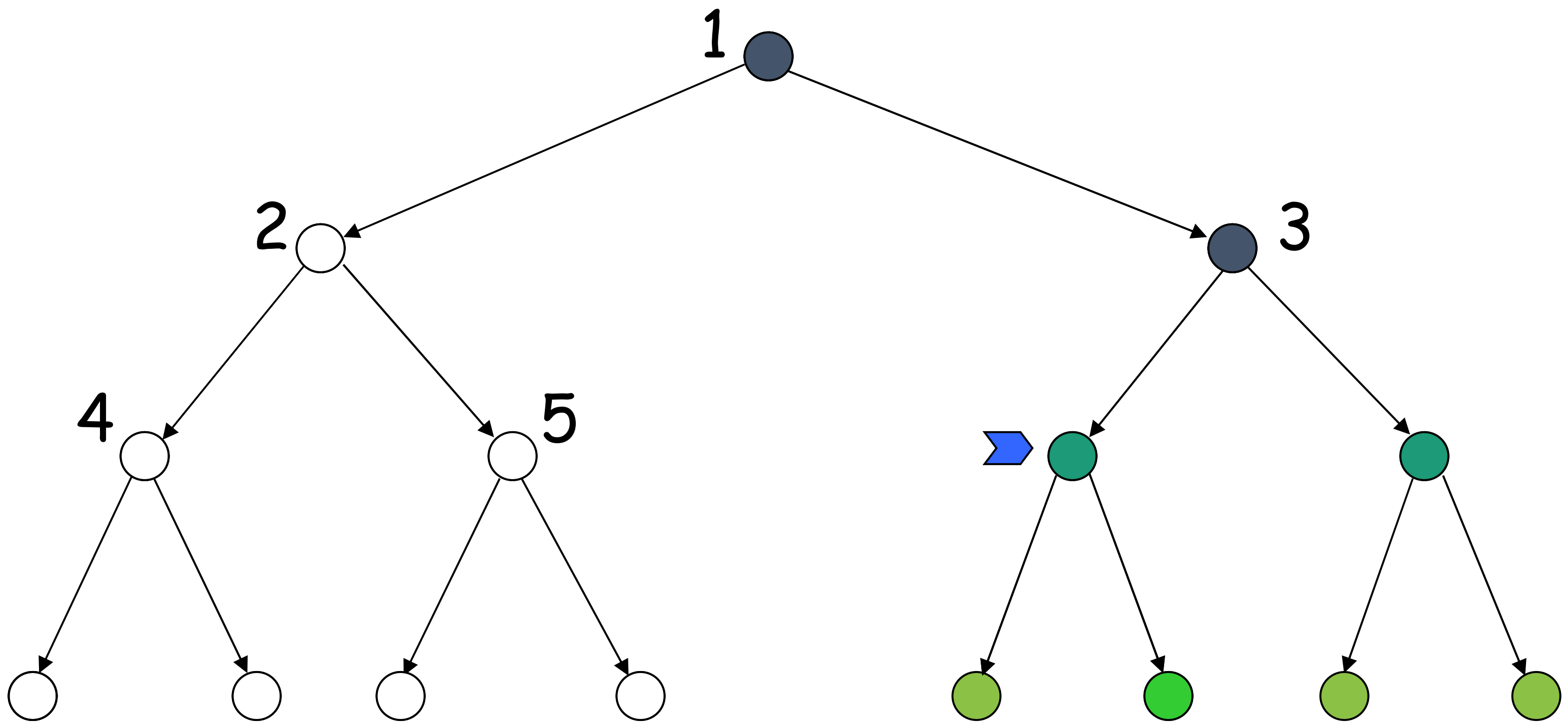
Depth-First Strategy

New nodes are inserted at the front of FRINGE



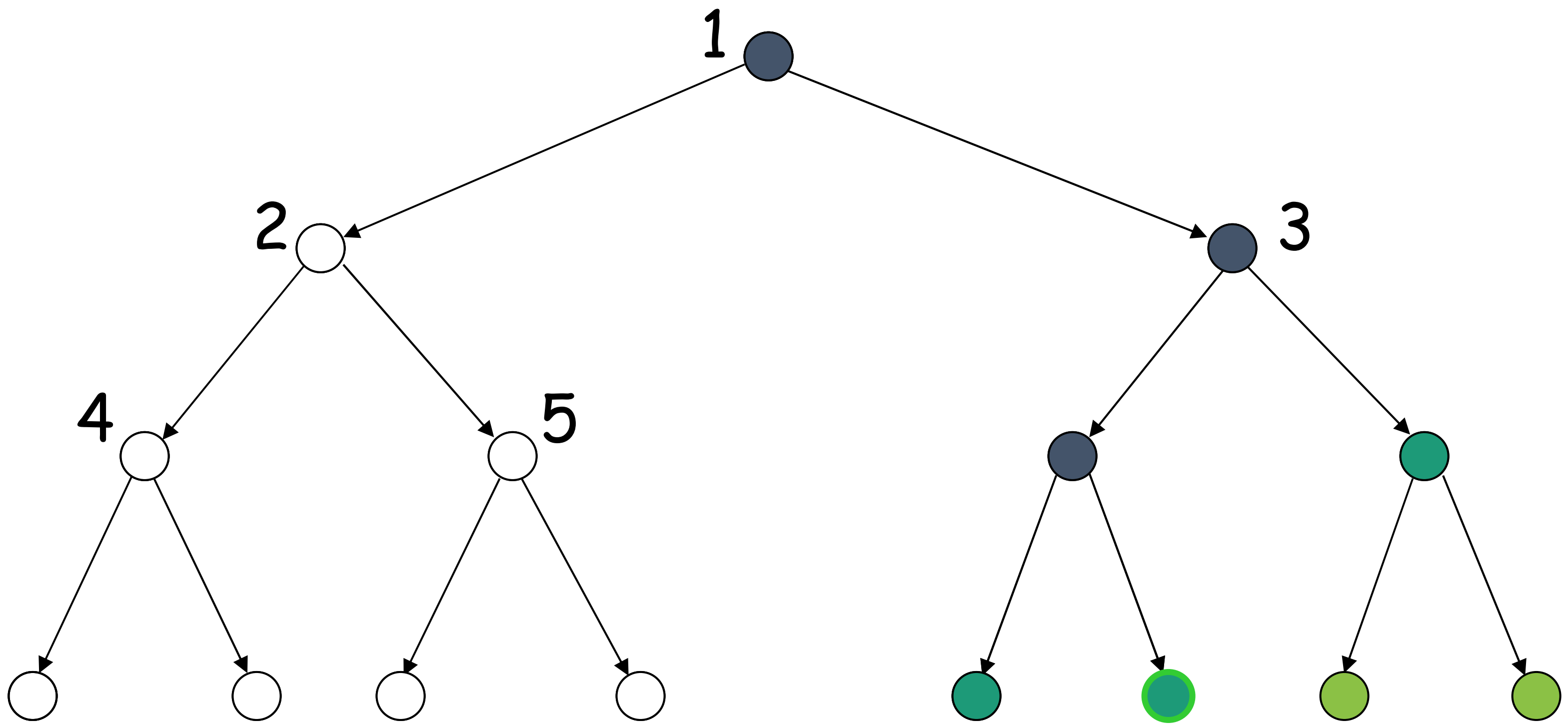
Depth-First Strategy

New nodes are inserted at the front of FRINGE

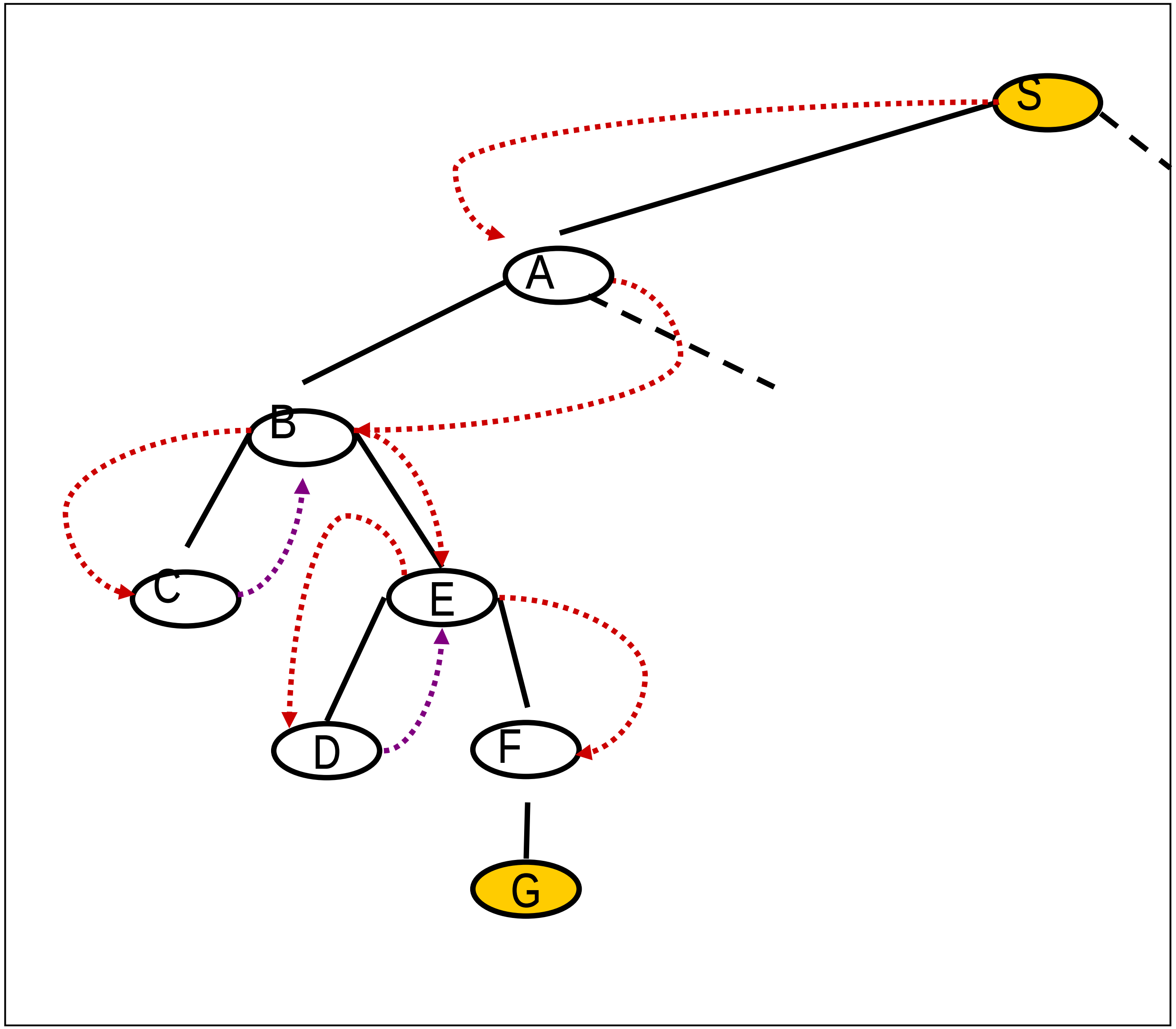


Depth-First Strategy

New nodes are inserted at the front of FRINGE

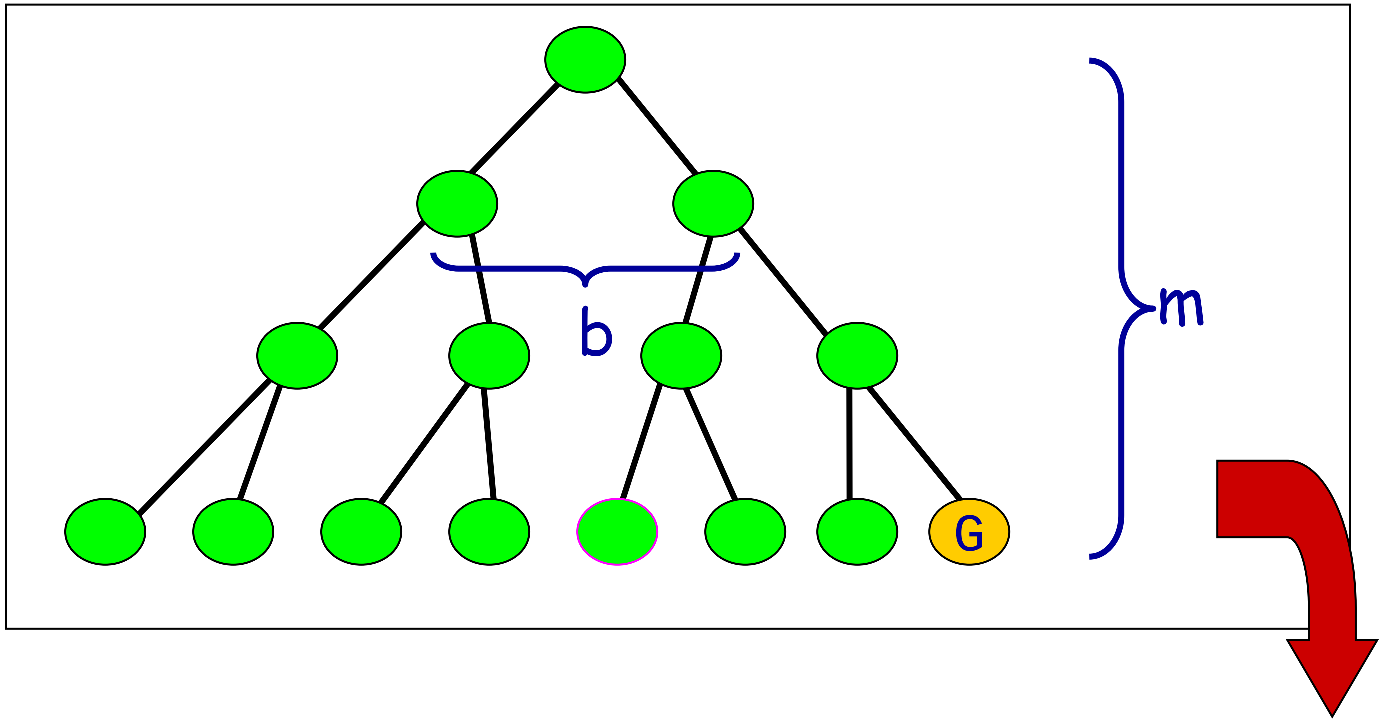


Depth First Search



Time complexity of DFS

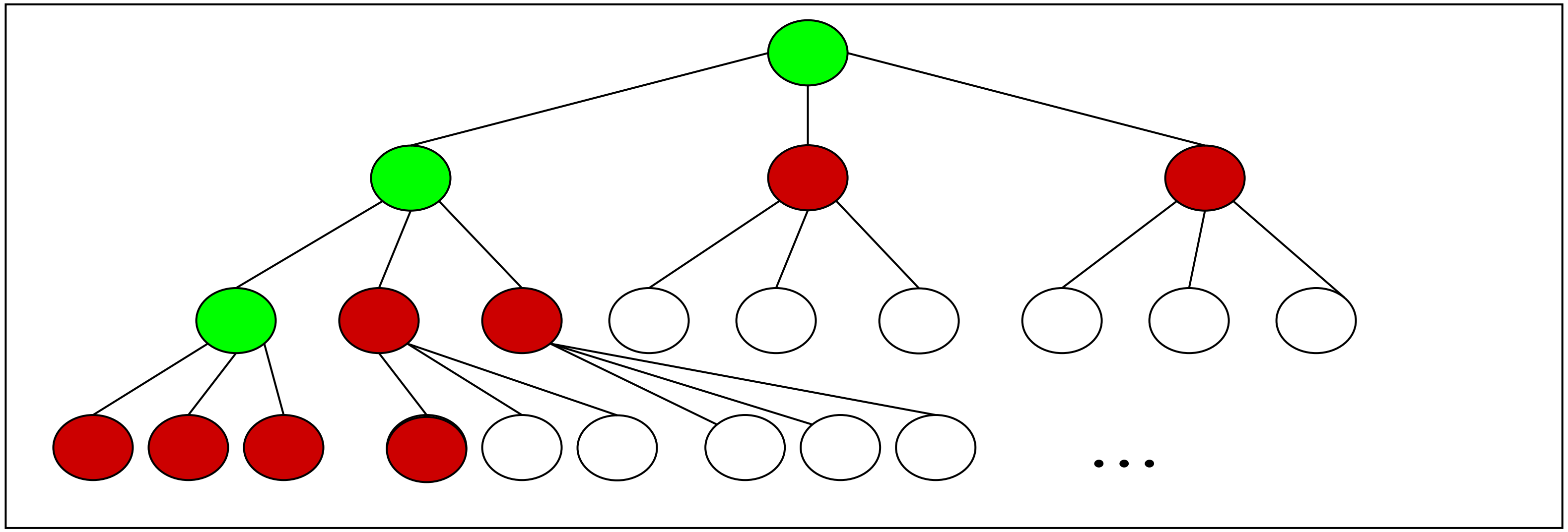
- In the **worst case**:
  - the (only) goal node may be on the right-most branch,



- Time complexity $b^m + b^{m-1} + \cdots + 1 = \frac{b^{m+1}-1}{b-1} = O(b^m)$

Space complexity of DFS

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: m = 3, b = 3 :



- **QUEUE** contains all nodes. Thus: 7.
- In General: ((b-1) * m) + 1
- Order: O(m*b)

Code example DFS.py

```
# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency |
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print (v),

        # Recur for all the vertices adjacent to
        # this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)
```

```python
    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self):
        V = len(self.graph)  #total vertices

        # Mark all the vertices as not visited
        visited =[False]*(V)

        # Call the recursive helper function to print
        # DFS traversal starting from all vertices one
        # by one
        for i in range(V):
            if visited[i] == False:
                self.DFSUtil(i, visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Depth First Traversal")
g.DFS()
```

Depth-Limited Search

Depth-Limited Search

- Depth-first with depth cutoff *k* (depth below which nodes are not expanded)

- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)

Depth-Limited Search

```
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
```

http://aima.cs.berkeley.edu/figures.html

Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS)

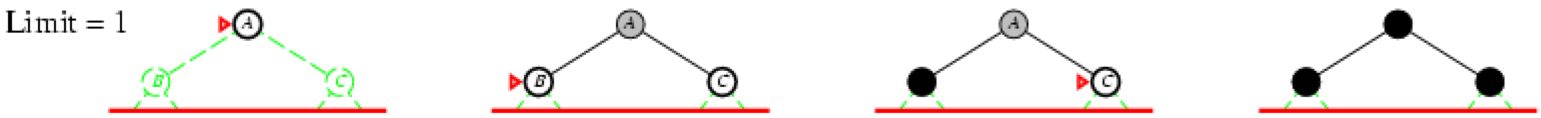
- Provides the best of both breadth-first and depth-first search
- Main idea: Totally horrifying !

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

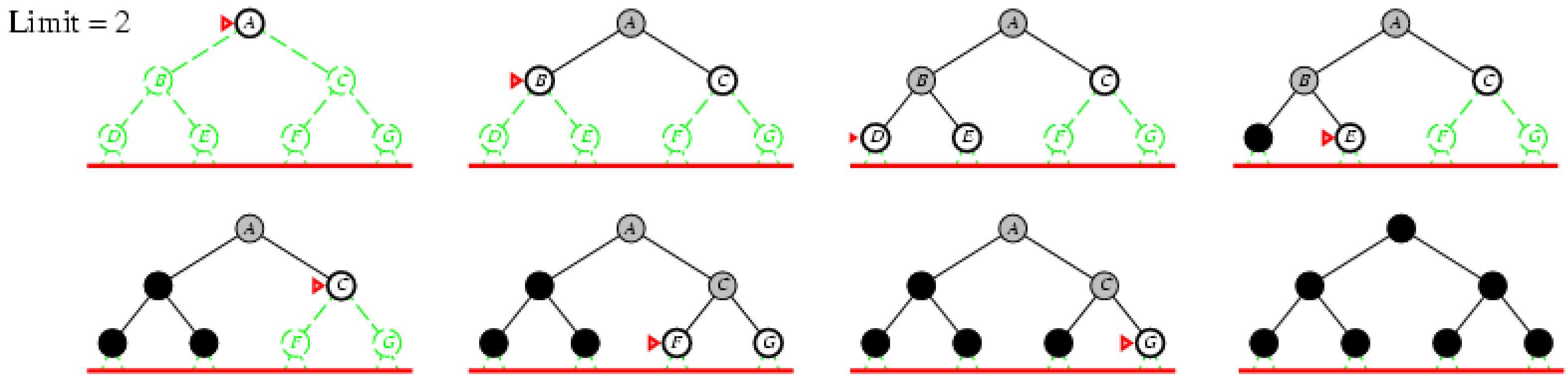
Iterative deepening search *l* =0

Limit = 0

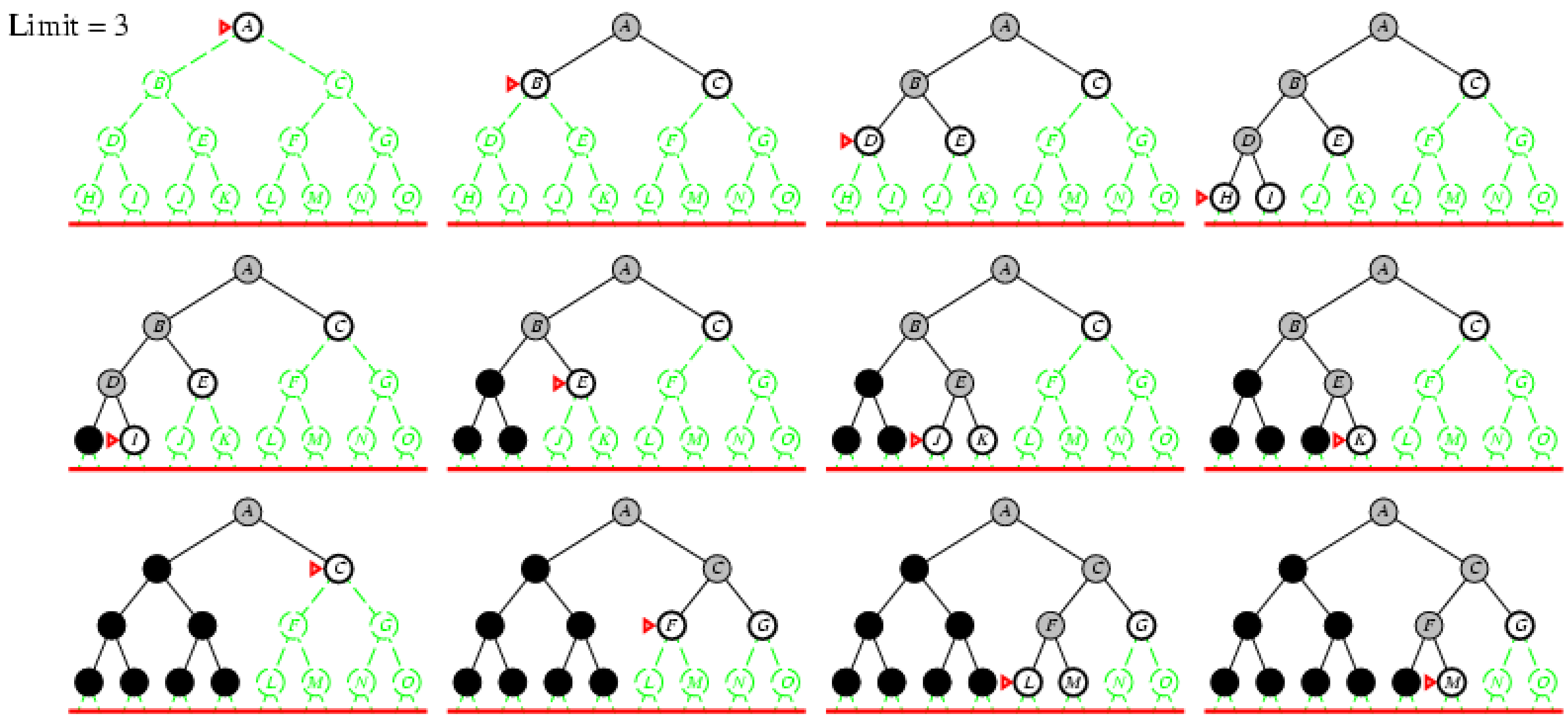
Iterative deepening search *l* =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

- Number of nodes generated in a depth-limited search (DLS) to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

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

$$\mathrm{N_{IDS}} = (\mathrm{d}+1)\mathrm{b}^0 + \mathrm{d}\ \mathrm{b}^1 + (\mathrm{d}-1)\mathrm{b}^2 + \ldots + 3\mathrm{b}^{\mathrm{d}-2} + 2\mathrm{b}^{\mathrm{d}-1} + 1\mathrm{b}^\mathrm{d}$$

- For $b = 10, d = 5$,
  - N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

- Overhead = (123,456 - 111,111)/111,111 = 11%

Performance

- **Iterative deepening search is:**
  - Complete
  - Optimal if step cost =1
- Time complexity is:
  $(d+1)(1) + db + (d-1)b^2 + \ldots + (1)\, b^d = O(b^d)$
- Space complexity is: $O(bd)$ or $O(d)$

$$db + (d-1)b^2 + \ldots + (1)\, b^d$$

$$= b^d + 2b^{d-1} + 3b^{d-2} + \ldots + db$$

$$= (1 + 2b^{-1} + 3b^{-2} + \ldots + db^{-d}) \times b^d$$

$$\leq \left(\Sigma_{i=1,\ldots,\infty}\; ib^{(1-i)}\right) \times b^d \;=\; b^d\, (b/(b-1))^2$$

Code example IDS.py

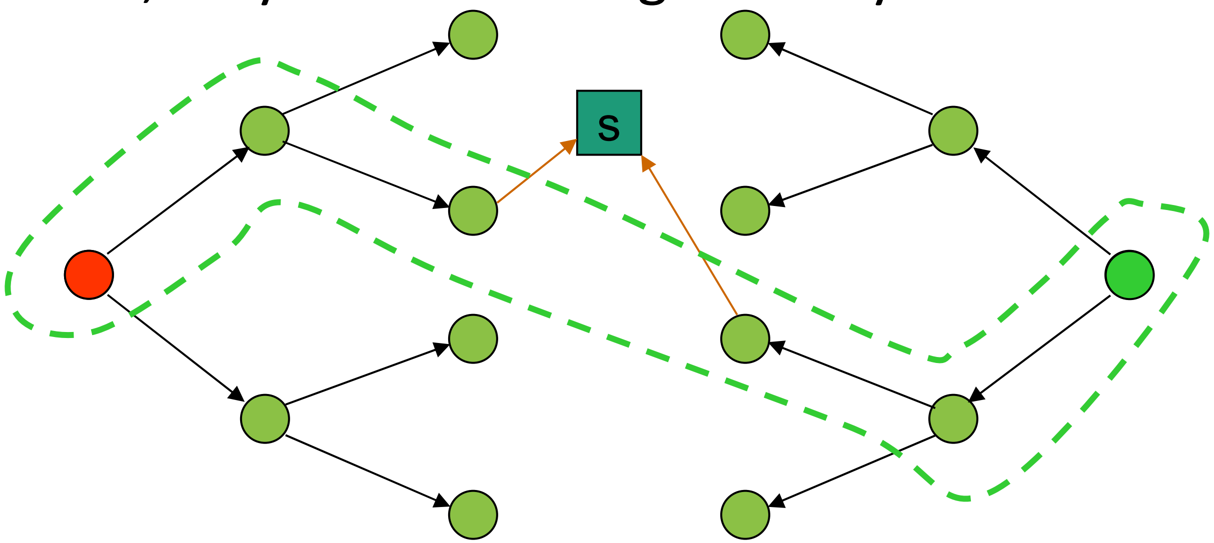
```python
# Python program to print DFS traversal from a given
# given graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    def __init__(self,vertices):

        # No. of vertices
        self.V = vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function to perform a Depth-Limited search
    # from given source 'src'
    def DLS(self,src,target,maxDepth):

        if src == target : return True

        # If reached the maximum depth, stop recursing.
        if maxDepth <= 0 : return False

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1)):
                    return True
        return False
```

```python
    # IDDFS to search if target is reachable from v.
    # It uses recursive DLS()
    def IDS(self,src, target, maxDepth):

        # Repeatedly depth-limit search till the
        # maximum depth
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False

# Create a graph given in the above diagram
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

if g.IDS(src, target, maxDepth) == True:
    print ("Target is reachable from source " +
        "within max depth")
else :
    print ("Target is NOT reachable from source " +
        "within max depth")
```

Bidirectional Strategy

Bidirectional Strategy

- Both search forward from initial state, and backwards from goal.
- Stop when the two searches **meet in the middle**.
- Problem: how do we search backwards from goal??
  - predecessor of node n = all nodes that have n as successor
  - this may not always be easy to compute!
  - if several goal states, apply predecessor function to them just as we applied successor (only works well **if goals are explicitly known**; may be difficult if goals only characterized implicitly).



Bidirectional search Algorithm

1. QUEUE1 <-- path only containing the root;
   QUEUE2 <-- path only containing the goal;
2. WHILE both QUEUEs are not empty
   AND QUEUE1 and QUEUE2 do NOT share a state
   DO remove their first paths;
      create their new paths (to all children);
      reject their new paths with loops;
      add their new paths to back;
3. IF QUEUE1 and QUEUE2 share a state
      THEN success;
      ELSE failure;

Bidirectional search

- Completeness: Yes,
- Time complexity: $2*O(b^{d/2}) = O(b^{d/2})$
- Space complexity: $O(b^{m/2})$
- Optimality: Yes

- To avoid one by one comparison, we need a hash table of size $O(b^{m/2})$
- *If hash table is used, the cost of comparison is O(1)*

uninformed search strategies

	BFS	DFS	DLS	IDS	Bidierectional DLS
Time	b^d	b^m	b^l	$b^{d/2}$	$b^{d/2}$
Space	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	No	No	Yes	Yes
Complete?	Yes	No	Yes	Yes if l >d	Yes

- ***b*** **– max branching factor of the search tree**
- ***d*** **– depth of the least-cost solution**
- ***m*** **– max depth of the state-space (may be infinity)**
- ***l*** **– depth cutoff**

Comparison of Strategies

- Breadth-first is complete and optimal, but has high space complexity
- Depth-first is space efficient, but is neither complete, nor optimal
- Iterative deepening is complete and optimal, with the same space complexity as depth-first and almost the same time complexity as breadth-first

Informed (Heuristic) Search Strategies

Informed (Heuristic) Search Strategies

- Use problem–specific knowledge beyond the definition of the problem itself.
- Can fine solutions more efficiently than an uninformed strategy.

Greedy Best-first search

- An instance of TREE-SEARCG or GRAPH-SEARCH
- Idea:

  use an evaluation function f(n) for each node; estimate of *“desirability”*

  ⇒expand most desirable unexpanded node.

  ⇒f(n) estimated cost of the cheapest path from the state at node *n* to a goal state.

  ⇒The node with the lowest evaluation is selected for expansion.

  ⇒Measure = distance to goal state.

- Implementation: priority queue.

  Queueing Fn = insert successors in decreasing order of desirability

- Special cases:

  greedy search, A* search,

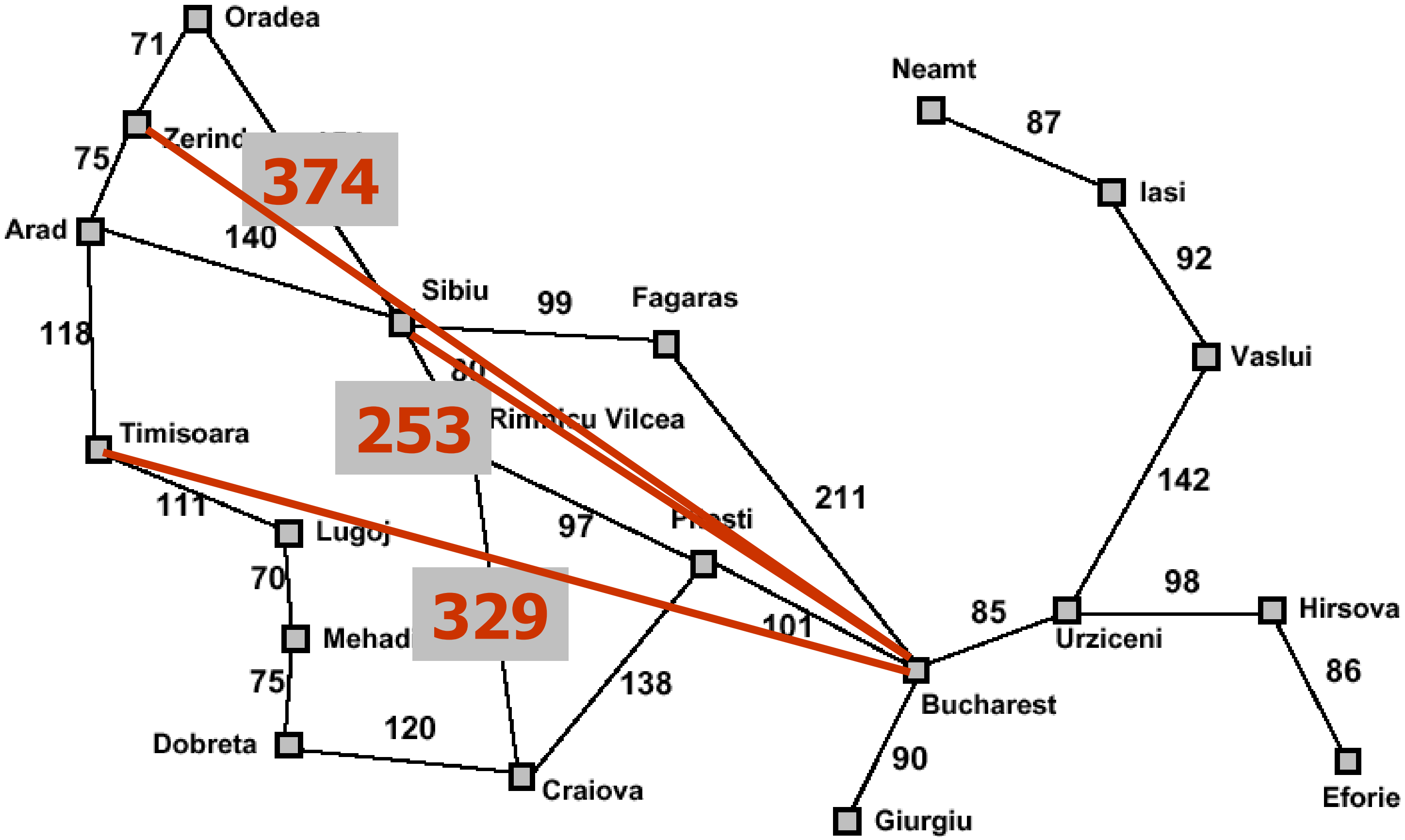
Best-First Search

- Best ≠ best path to goal.
- Best = Appears to be the best according to the evaluation function.
- If **f(n)** is accurate, then OK. (f(n) =??)
- True meaning: “seemingly-best-first search”
- Greedy method.
- Heuristic function **h(n):**
  - estimated cost of the cheapest path from node n to a goal node.
  - **h(n)**: nonnegative
  - If n is a goal node, then h(n)=0.

Greedy best-first search

- Tried to expand the node that is closet to the goal.
- Let f(n)=h(n).
- Example: **straight-line distance** heuristic, which we will call $\boldsymbol{h_{SLD}}$.
- Greedy: at each step it tries to get as close to the goal as it can.

Romania with step costs in km



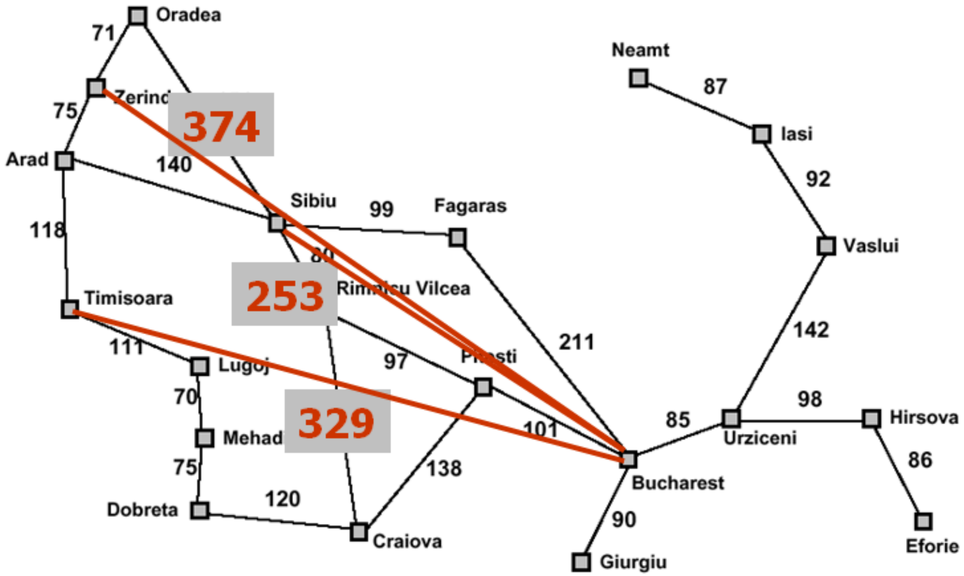
h_{SLD}

Straight–line distance
to Bucharest

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

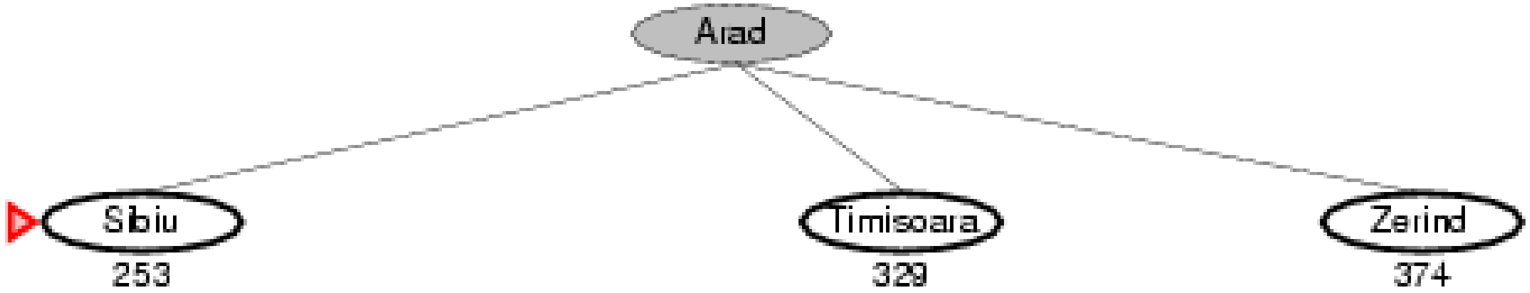
http://aima.cs.berkeley.edu/figures.html

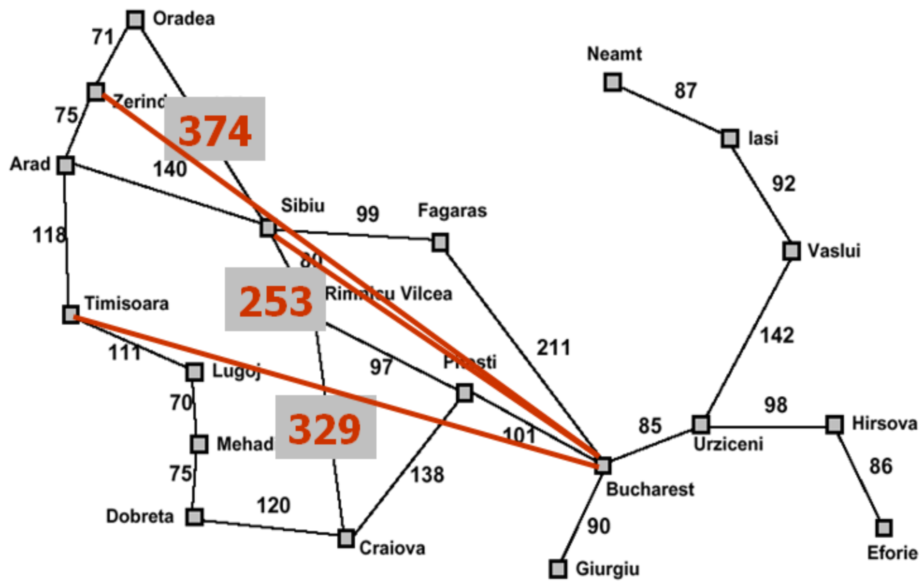
Greedy best-first search example



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

Greedy best-first search example

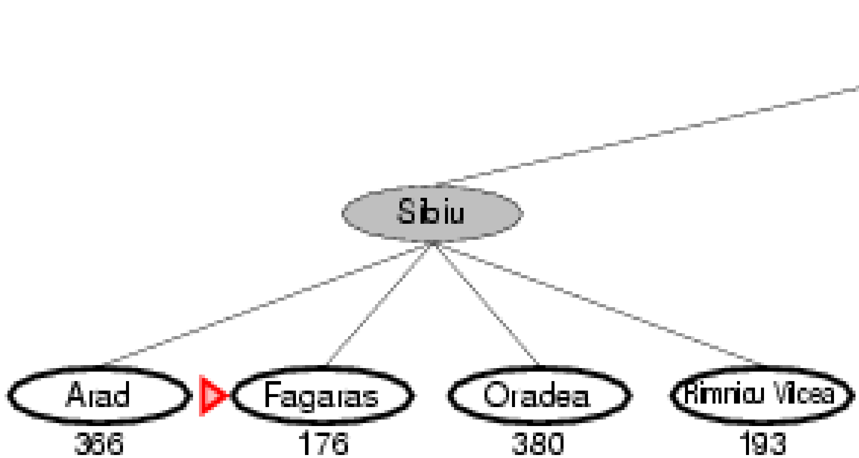


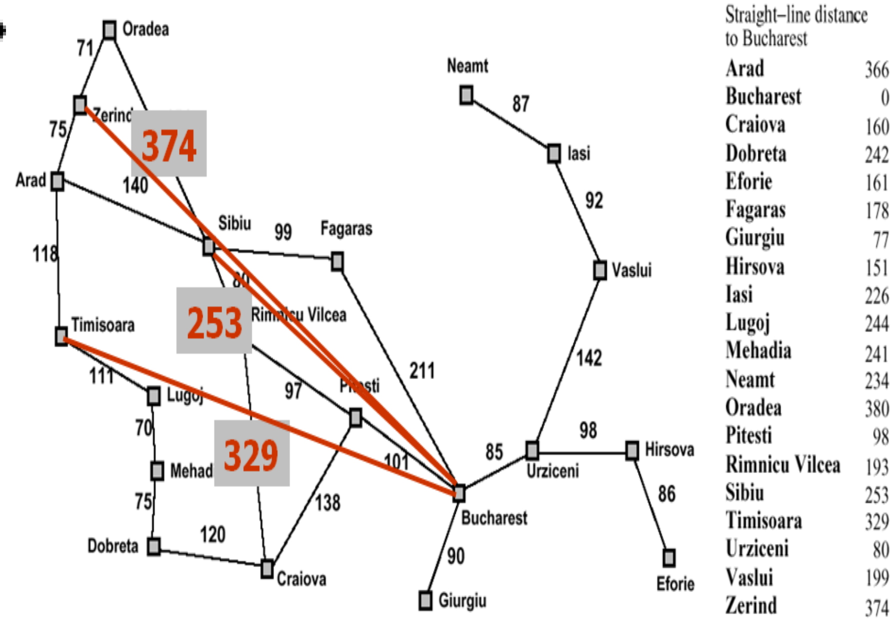


Straight–line distance
to Bucharest

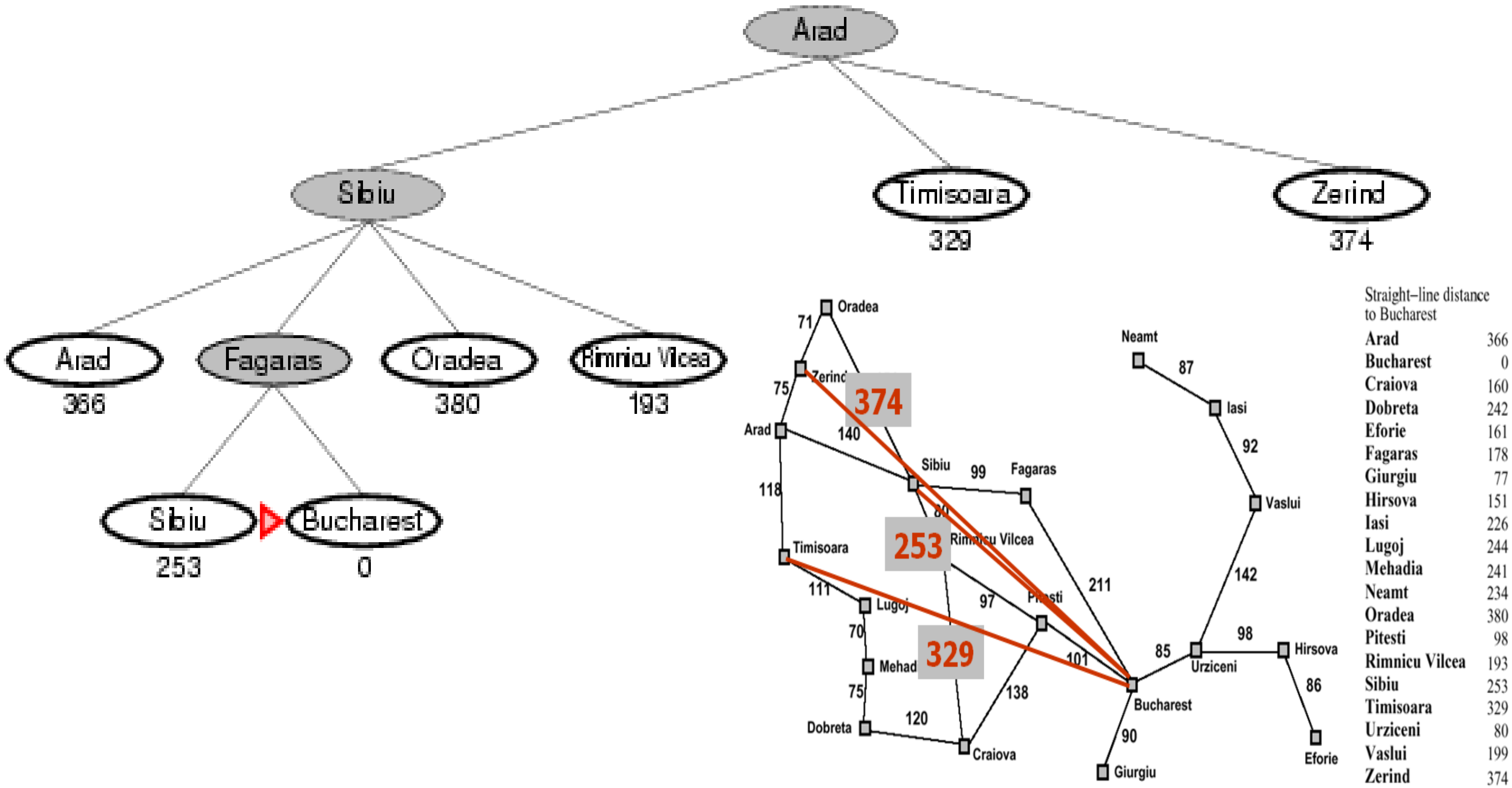
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

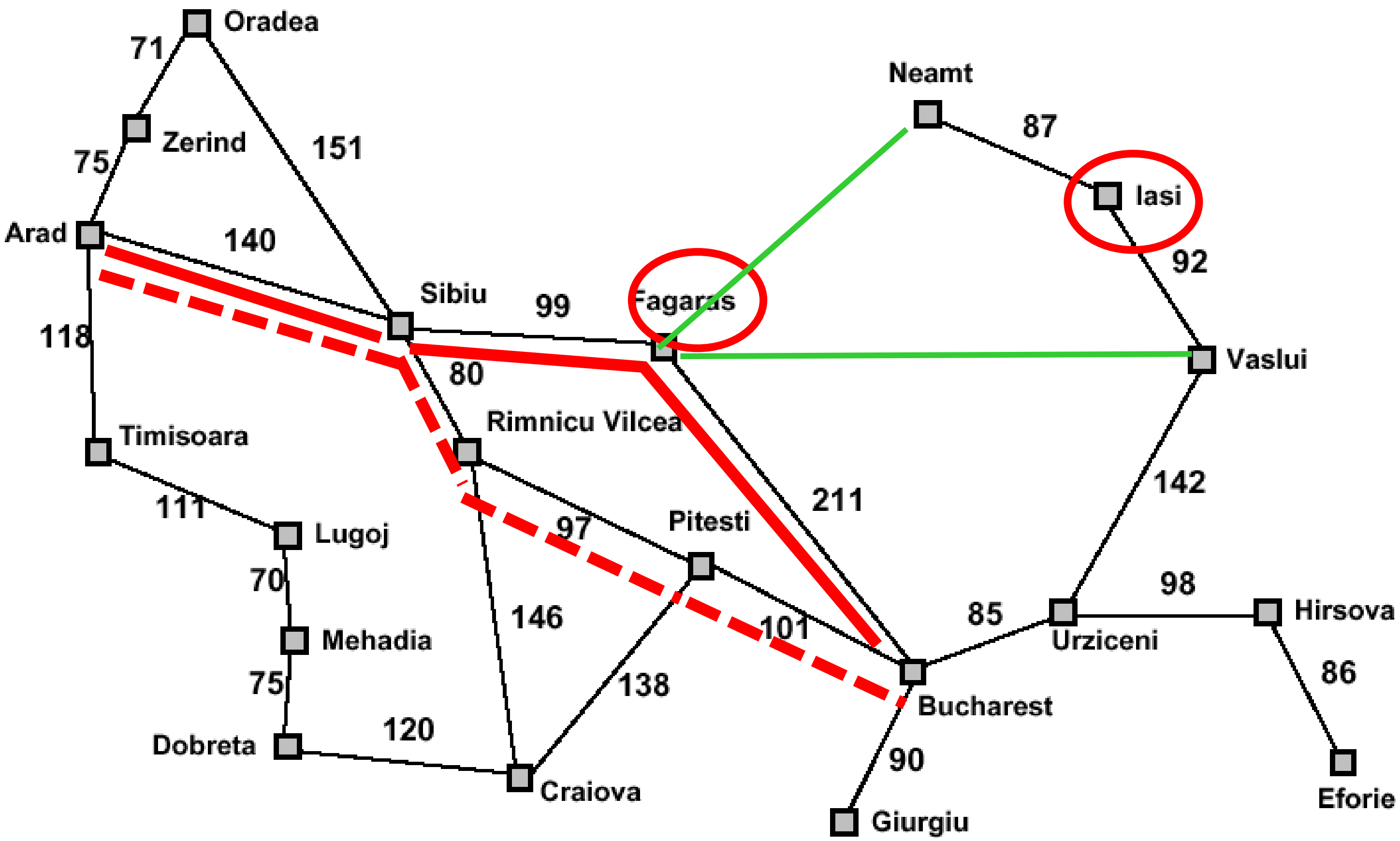




Greedy best-first search example



Not optimal (greedy)



Straight–line distance
to Bucharest

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

Properties of greedy best-first search

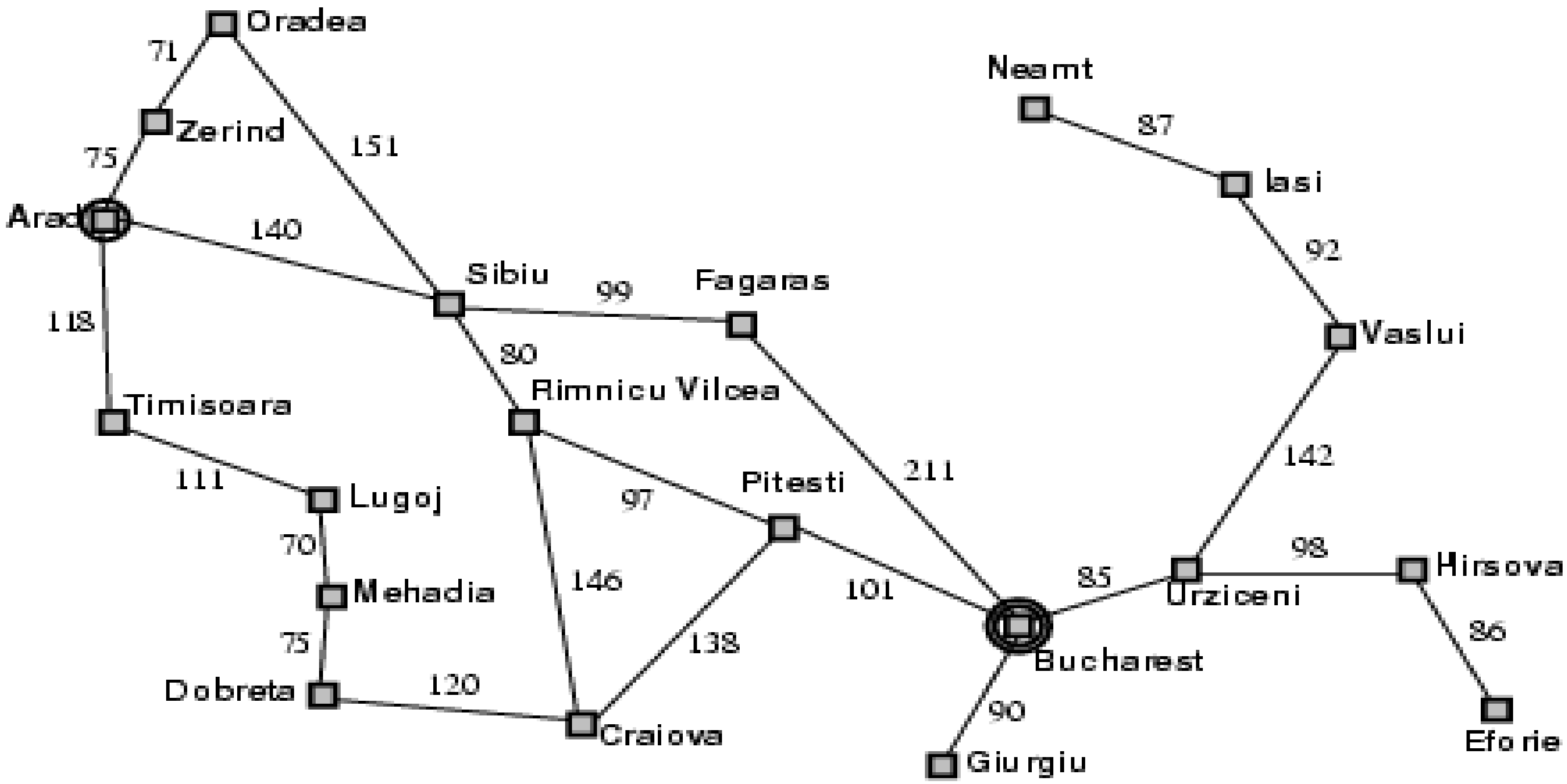
- Complete? No – **can get stuck in loops**, e.g., Iasi → Neamt → Iasi → Neamt → (to Fagaras)
  - Susceptible to false starts
  - (may be no solution)
  - May cause unnecessary nodes to expanded
  - Stuck in loop. (**Incomplete**)
- Time? ***$O(b^m)$***, but a good heuristic can give dramatic improvement
- Space? ***$O(b^m)$*** -- keeps all nodes in memory
- Optimal? No

A^* search

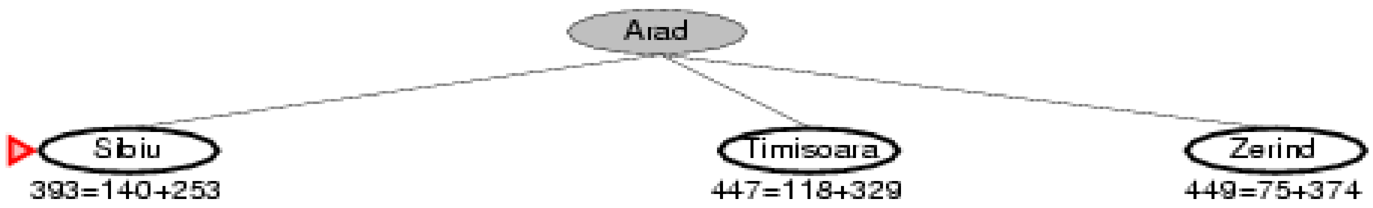
A^* search

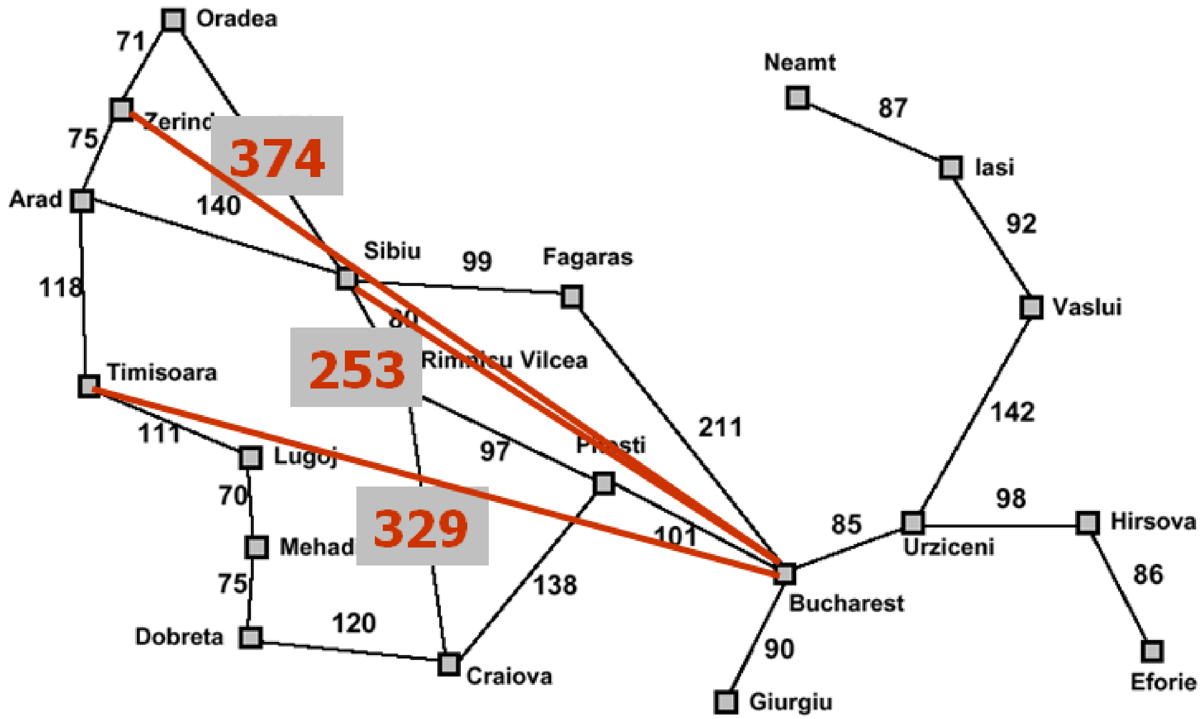
- Minimizing the total estimated solution cost
- Idea: avoid expanding paths that are already expensive
- **Evaluation function** $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated the cheapest cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal
- **Both complete and optimal**

A* search example



A* search example

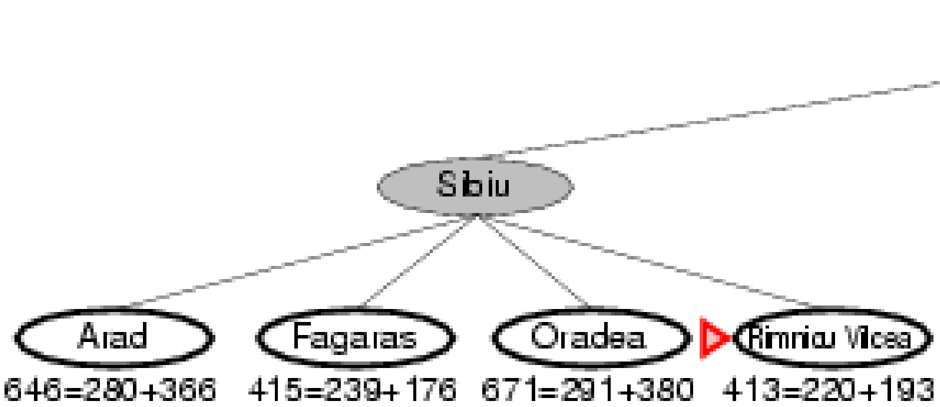


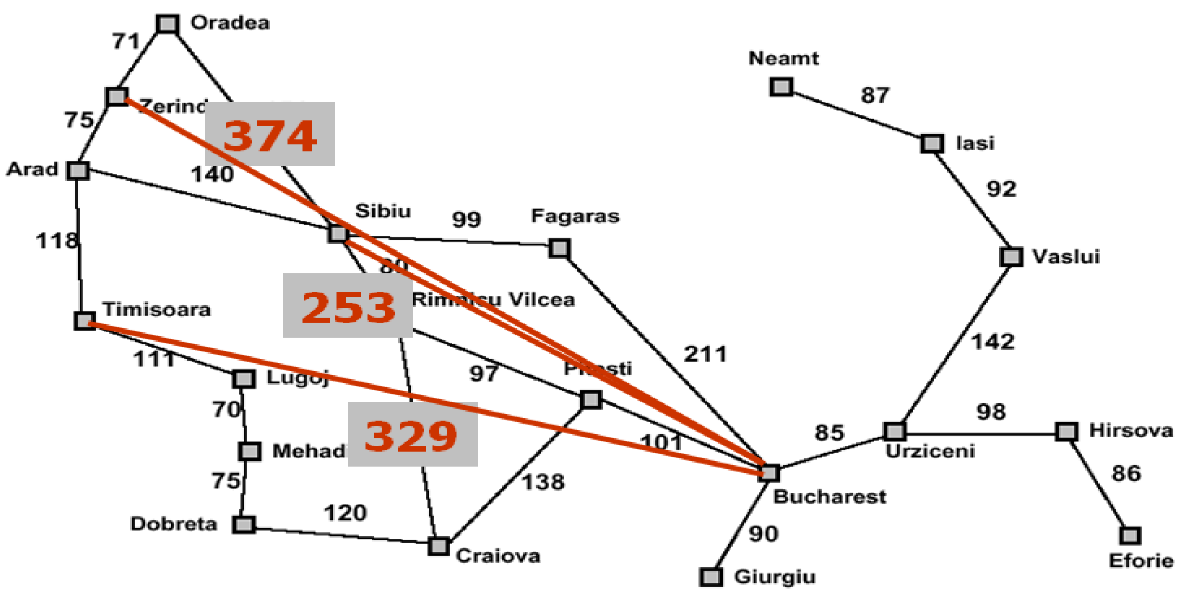


Straight–line distance to Bucharest

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

A^* search example

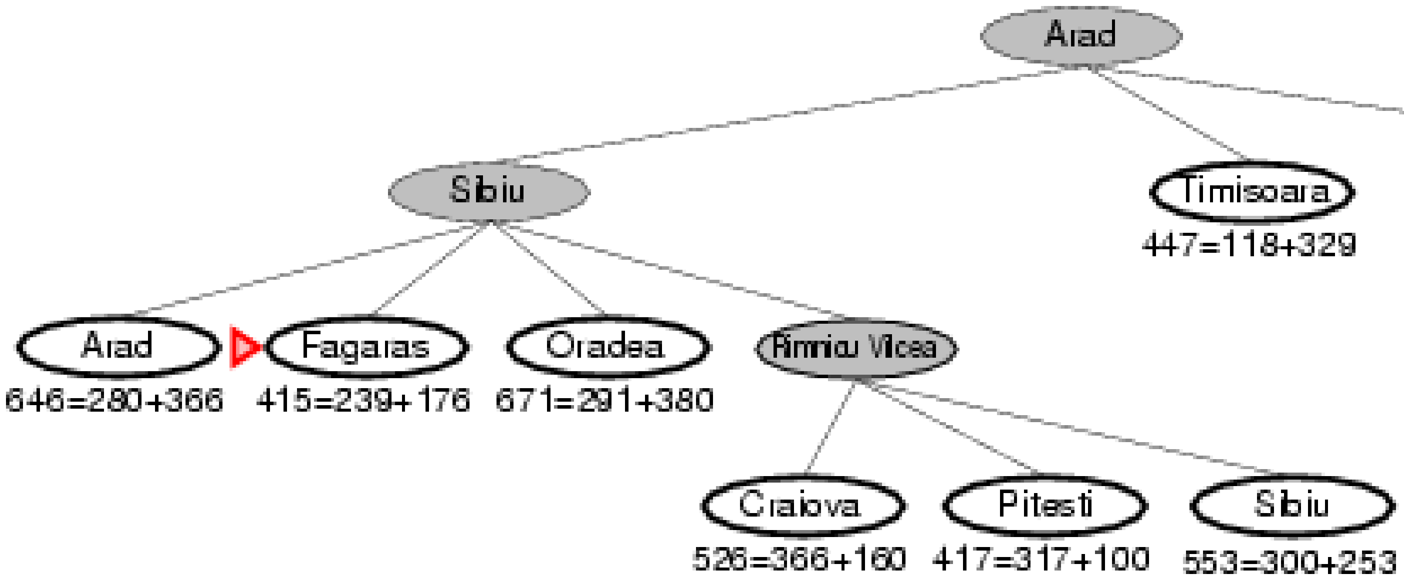


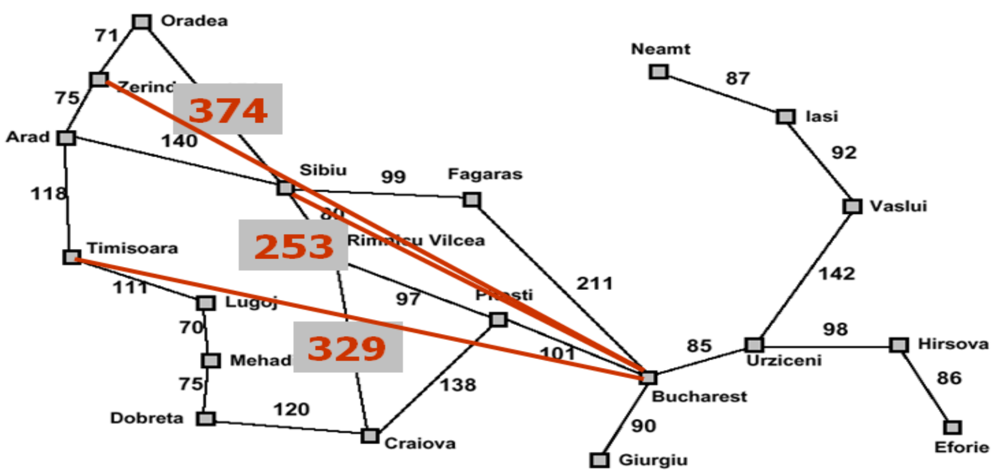


Straight–line distance
to Bucharest

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

A* search example

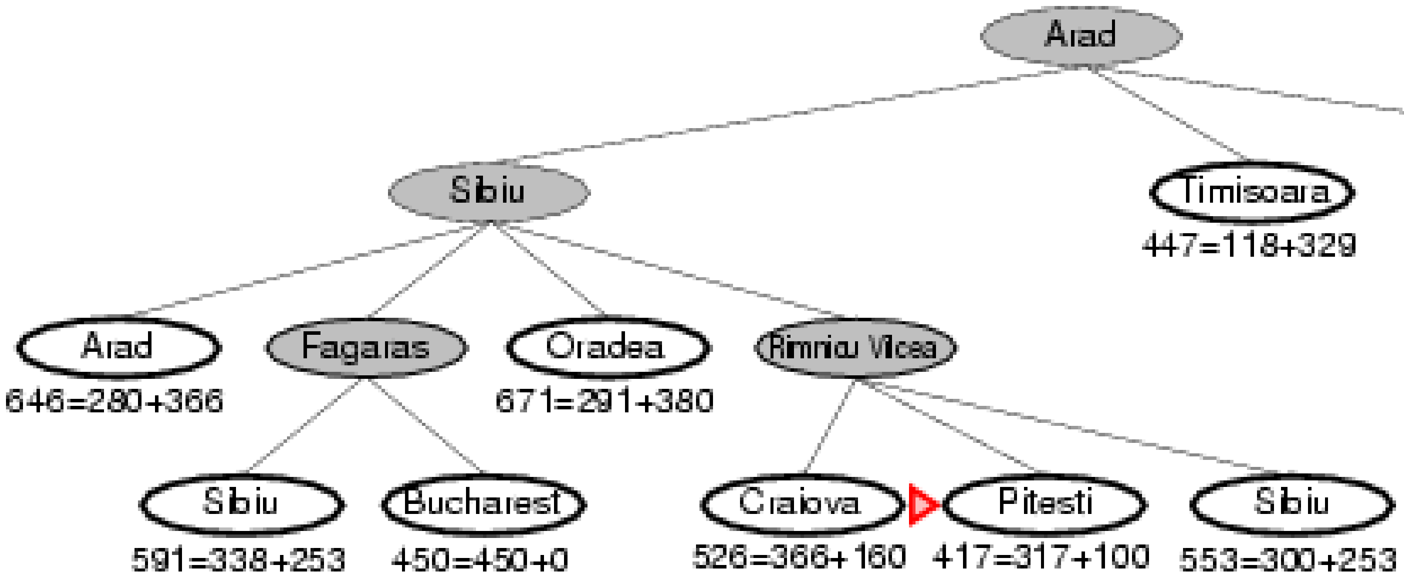


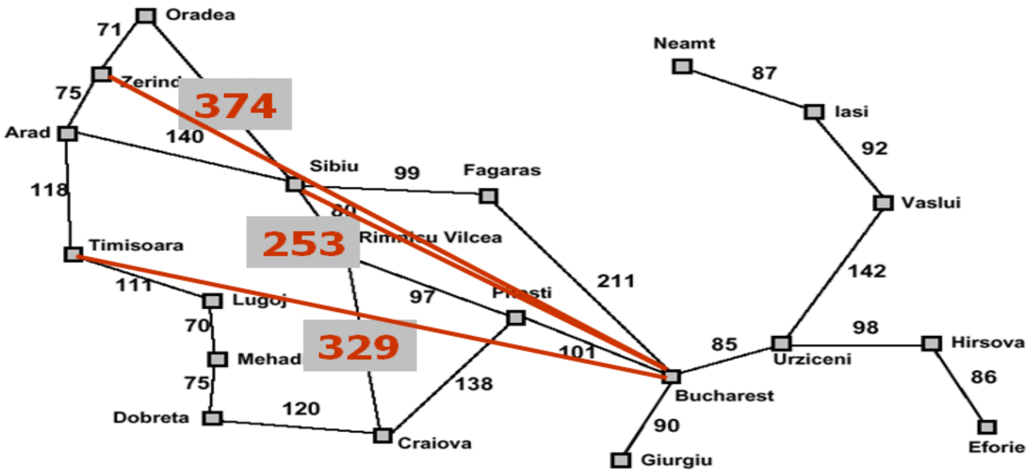


Straight−line distance
to Bucharest

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

A^* search example

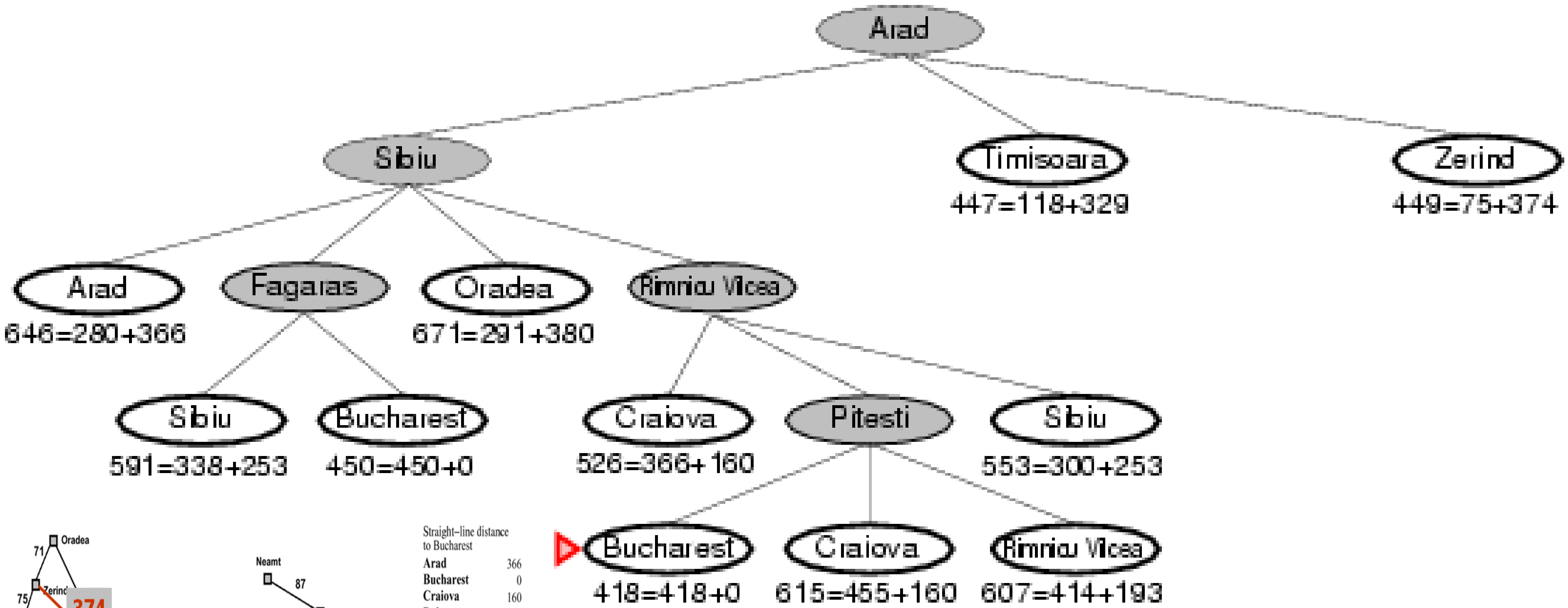


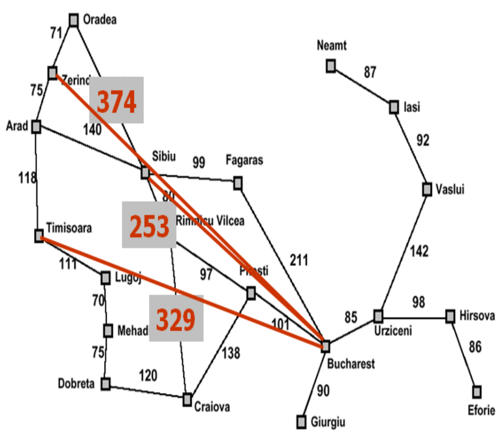


Straight–line distance to Bucharest

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

A* search example





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

Admissible heuristics

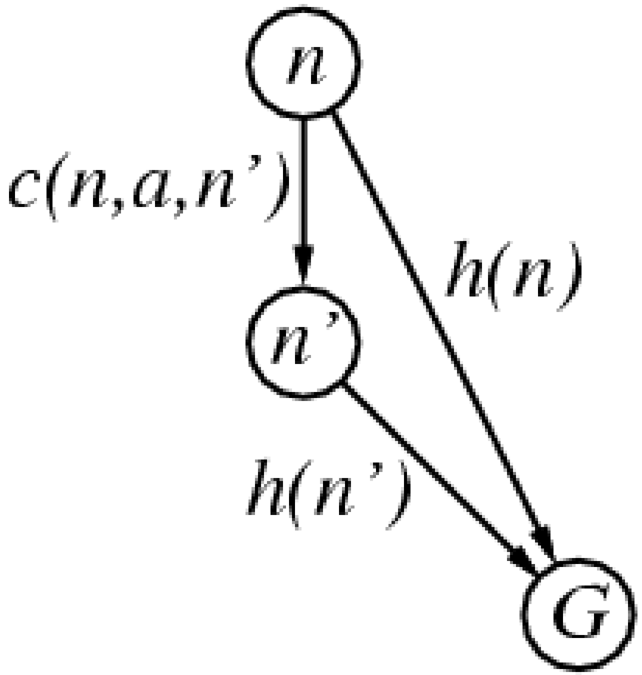
- **Conditions for optimality: Admissibility (可採納的) and consistency (一致性)**
- A heuristic $h(n)$ is **admissible** if for every node n,

  $h(n) \leq h^*(n)$, where $\boldsymbol{h^*(n)}$ is the true cost to reach the goal state from n.
- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is optimistic.
- Example: $\boldsymbol{h_{SLD}(n)}$ (never overestimates the actual road distance)
- **Theorem**: **If $h(n)$ is admissible, A* using TREE-SEARCH is optimal**

Consistency (monotonicity) heuristics

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

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



- If *h* is consistent, we have

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

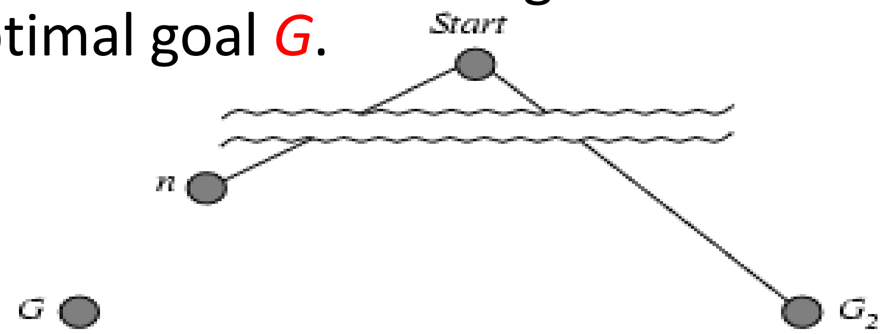
- i.e., *f(n)* is **non-decreasing** along any path.

Triangle inequality

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

Optimality of A* (TREE-search)

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

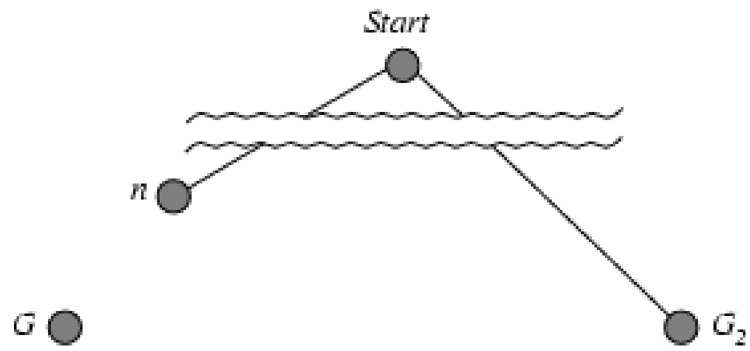


G_2 and n in fringe

- $f(G_2) = g(G_2)+h(G_2)=g(G_2) > C^*$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- If h(n) does not overestimate the cost of completing the solution path ($h(n) \leq h^*(n)$)
- $f(n) = g(n)+h(n) \leq g(n) + h^*(n) \leqq C^*$
- $f(n) \leqq C^* < f(G_2)$
- So, **G_2 will not be expanded and A* must return an optimal solution**.

Optimality of A^* (proof)

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

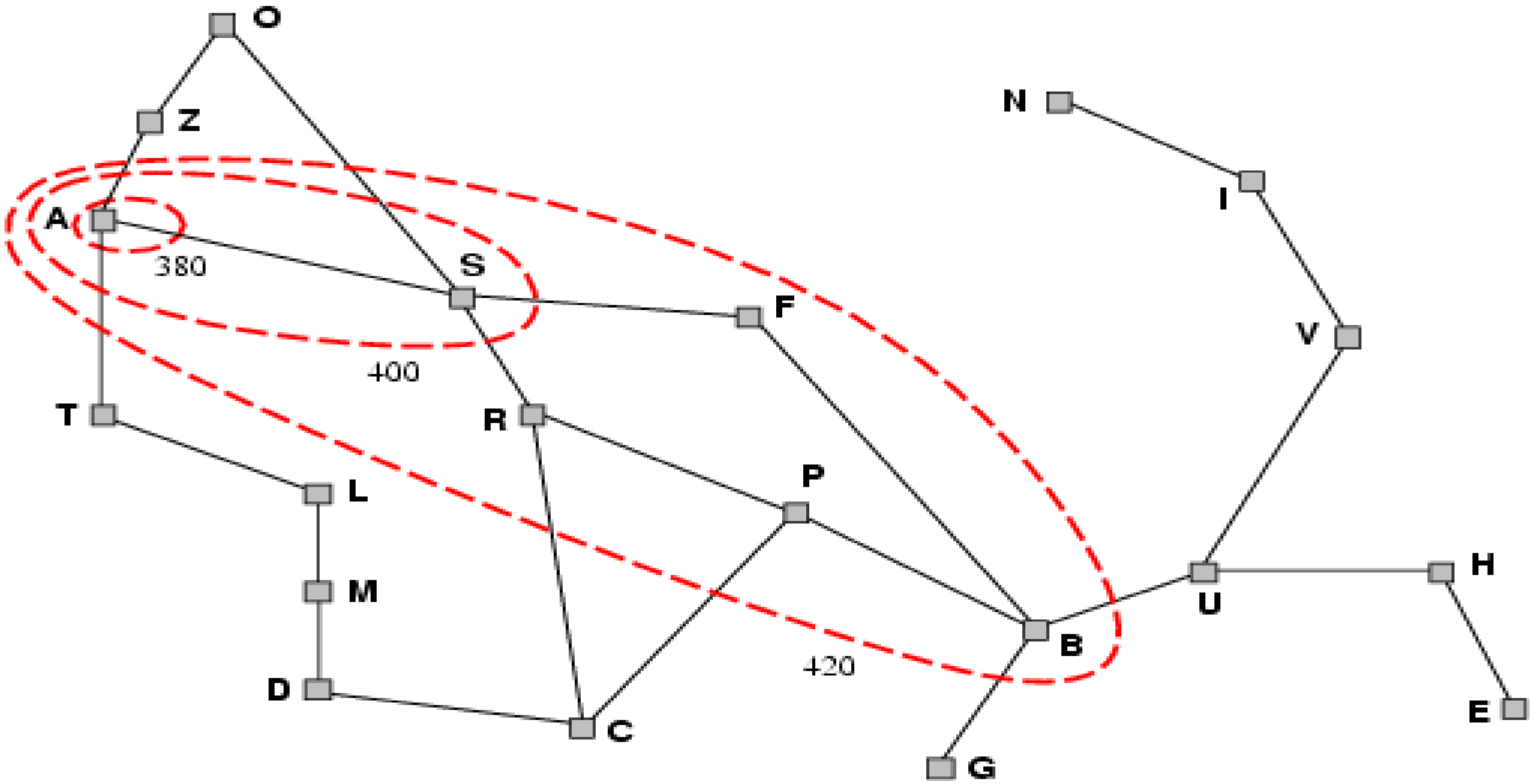


- $f(G_2) > f(G)$ from above
- $h(n) \le h^*(n)$ since h is admissible
- $g(n) + h(n) \le g(n) + h^*(n)$
- $f(n) \le f(G)$

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

Optimality of A^*

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



http://aima.cs.berkeley.edu/figures.html

Properties of A*

- A* expands all nodes with f(n) < C*
- A* might then expand some of the nodes right on the "goal contour" (f(n) =C*) before selecting a goal state.
- The solution found must be an optimal one.
- Complete? **Yes** (unless there are infinitely many nodes with f ≤ *f(G)*)
- Time? **Exponential**
- Space? Keeps all nodes in memory, before finding solution it may run out of the memory.
- Optimal? **Yes**
- Optimal Efficient: for any given heuristic function, on other optimal algorithm is guaranteed to expand fewer nodes than A*. Since A* expand no nodes with f(n) >C*.

Heuristic Functions

Heuristic Functions

E.g., for the 8-puzzle:

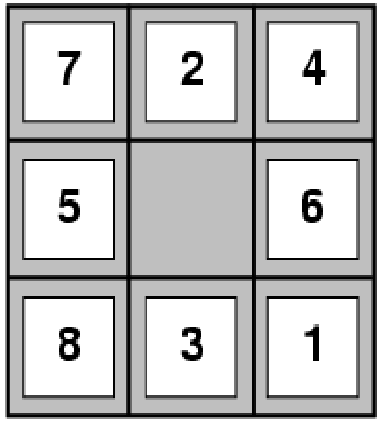
Average solution cost ~22 steps

Branching factor =3

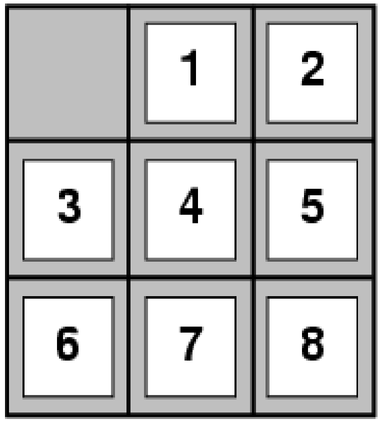
Space: $3^{22} = 3.1 * 10^{10}$

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

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



Start State



Goal State

Dominance

- **If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1**
- Domination translates directly into efficiency: A∗ using h_2 will never expand more nodes than A∗ using h_1.
- Effective branching factor b*:

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

- The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems.
- A well designed heuristic would have a value of **b* close to 1**, allowing fairly large problems to be solved at reasonable computational cost.

1200 tests

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

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH 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.

http://aima.cs.berkeley.edu/figures.html

Generating admissible heuristics from Relaxed problems

- A problem with fewer restrictions on the actions is called a **relaxed problem**
- If the rules of the 8-puzzle are **relaxed so that a tile can move anywhere**, then $h_1(n)$ gives the shortest solution.
- If the rules are **relaxed so that a tile can move to any adjacent square**, then $h_2(n)$ gives the shortest solution.

- **The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.**
- Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent.**

Construct heuristic from relaxed problem

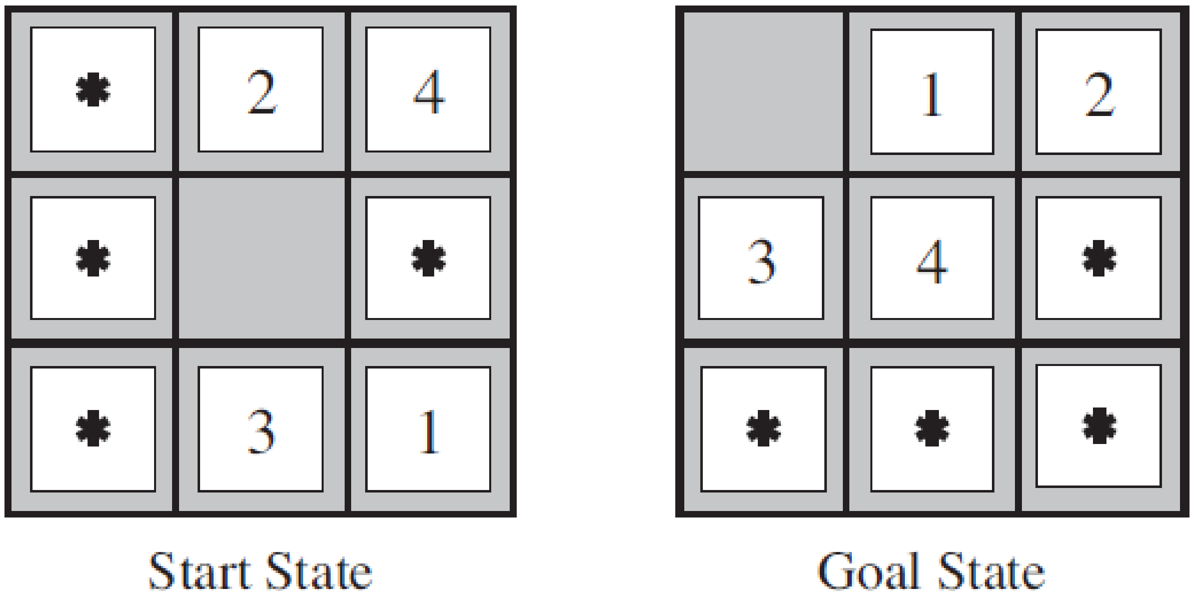
- If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.
- For example, if the 8-puzzle actions are described as
- A tile can move from square A to square B if A is horizontally or vertically adjacent to B **and** B is blank, three relaxed problem
  - (a) A tile can move from square A to square B if A is adjacent to B.
  - (b) A tile can move from square A to square B if B is blank.
  - (c) A tile can move from square A to square B.
- **ABSOLVER (1993)**
  - Generate heuristic automatic from problem definition.
  - Generate a new heuristic for 8-puzzle problem better than any-existing heuristic.
  - Found the first useful heuristic for the famous Rubik's cube puzzle(魔術方塊).

Combination of heuristics/ Drive from subproblem

- If a collection of admissible heuristics $h_{1, \ldots,} h_m$ is available for a problem and none of them dominates any of the others, which should we choose?
- We can have the best of all worlds, by defining $h(n)=\max\{h_1(n), h_2(n), \ldots, h_m(n)\}$.
- This composite heuristic uses whichever function is most accurate on the node in question.
- Because the component heuristics are admissible, *h* is admissible; it is also easy to prove that *h* is consistent. Furthermore, *h* dominates all of its component heuristics.

Generate heuristics from subproblem: Pattern databases

- **The optimal solution of the subproblem is a lower bound on the cost of the complete problem.**
- It turns out to be more accurate than Manhattan distance in some cases.
- The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem.



Learning heuristic from experience

- **Pattern database**
  - Then we compute an admissible heuristic h_{DB} for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database.
  - The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.
  - the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000.
- **disjoint pattern databases/** reduced by a factor of 10,000
- **Inducting learning feature**

Summary

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