Lecture 3: Problem Solving by Searching

Artificial Intelligence

CS-6364

Design of Agents for Problem-Solving

The **design of agents for problem-solving** methods may be broken down into:

a) **Problem formulation**

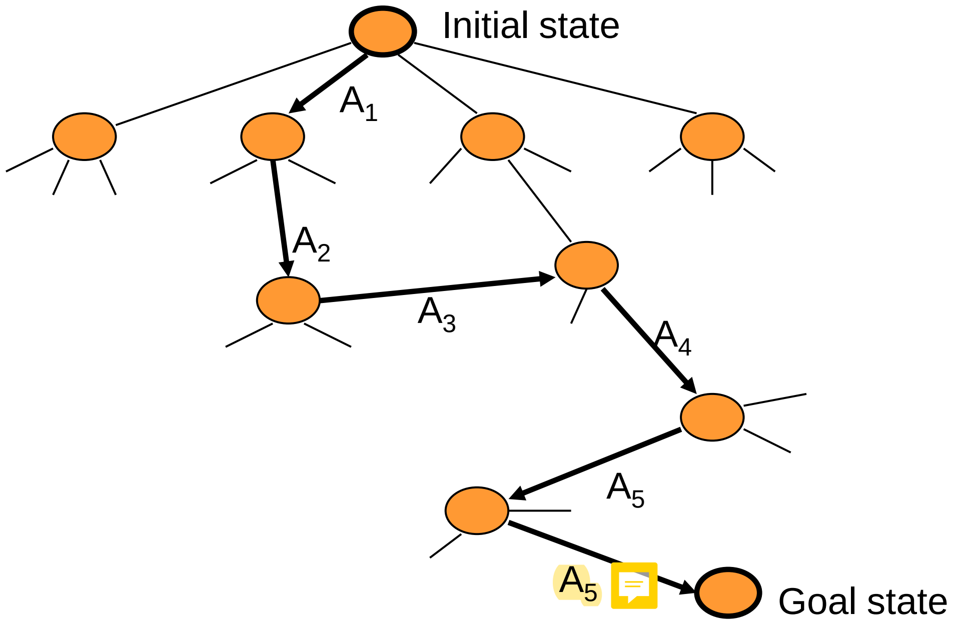
This is the process of transforming the problem to be solved into

- Set of possible states that completely describe all situations
- Actions to be taken, or transitions from a state to another.
- Goal state & the cost of obtaining a solution (solution path)

b) **Search algorithm** that takes the system from an initial state and provides a sequence of actions (and states) leading to goal. The solution is this sequence of actions.

c) **Execution** phase is the implementation of the sequence of actions

DETAILS: Agents for Problem-Solving



A_i—actions—that cause transitions from state to state.

Several possibilities of problem-solving systems:

$$\begin{tikzcd}[row sep=small] \text{Formulate} \arrow[d] \\ \text{Search} \arrow[d] \\ \text{Execute} \end{tikzcd} \qquad \begin{tikzcd}[row sep=small] \text{Formulate} \arrow[d] \\ \text{Search} \arrow[d, bend right] \\ \text{Execute} \arrow[u, bend right] \end{tikzcd}$$

Example 1 (Water jug puzzle)

There are two empty jugs, one of 4 gallons, one of 3 gallons. Fill the 4-gallon jug with 2 gallons of water.

(a) *Problem formulation*

Decide how to represent **states**

s_o: (*initial state*) (0,0)

$s_{i:}$: (*state i*) (x_i, y_i)

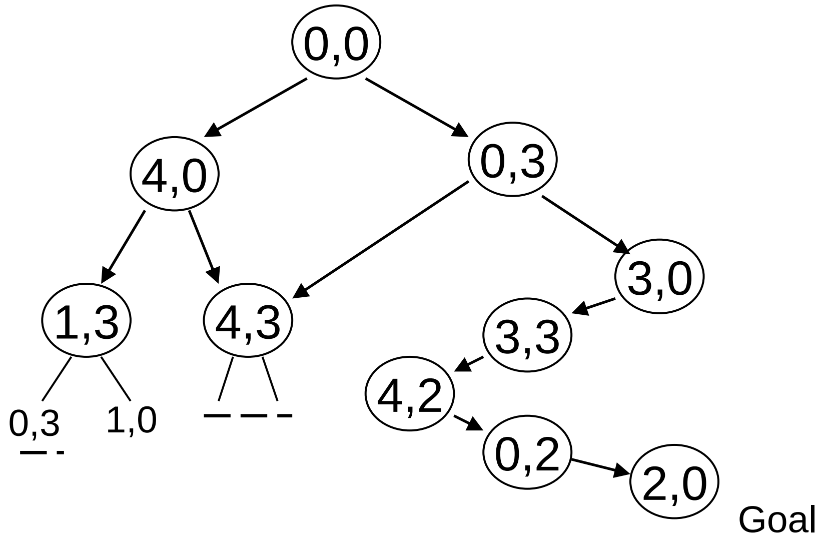
where: x_i—content of 4-gallon jug

y_i—content of 3-gallon jug

s_G: (*goal state*) (2,0)

Actions are to fill or empty the jugs

(b) *Search space*



Solution Path (in search space:

0 0

0 3

3 0

3 3

4 2

0 2

2 0

Production Rules for the Water Jug Puzzle

1	(x, y) if $x < 4$	$\rightarrow$ (4, y)	Fill the 4-gallon jug
2	(x, y) if $y < 3$	$\rightarrow$ (x, 3)	Fill the 3-gallon jug
3	(x, y) if $x > 0$	$\rightarrow$ (x-d, y)	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	$\rightarrow$ (x, y-d)	Pour some water out of the 3-gallon jug
5	(x, y) if $x > 0$	$\rightarrow$ (0, y)	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	$\rightarrow$ (x, 0)	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	$\rightarrow$ (4, y-(4-x))	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full

Production Rules: Continuation

8	(x, y) if $x + y \geq 3$ and $x > 0$	→ (x - (3 - y), 3)	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	→ (x + y, 0)	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	→ (0, x + y)	Pour all the water from 4-gallon jug into the 3-gallon jug
11	(0, 2)	→ (2, 0)	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	(x, 2)	→ (0, 2)	Empty the 4-gallon jug on the ground

Example 2: More complex Water jug puzzle

There are three empty jugs, one of 100 gallons, one of 4 gallons and one of 3 gallons. You are asked to design an intelligent agent that fills the 100-gallon jug with 5 gallons of water.

Specifically show:

The formulation of the problem by deciding how to represent an arbitrary state, the initial state and the goal state.

Solution:

An arbitrary state is represented as (X, Y, Z) where X,Y,Z are the volumes of water in 100 gallon jug, 4 gallon jug and 3 gallon jug respectively.

– *Initial state* (0,0,0)
– *Goal state* (5,X,X) here x is: don't care.

Example 2: continuation

Select a **set of actions** in the form of if-then rules that are used in your solution. Write the rules in plain English.

1) *Rule 1:* fill jug if empty
   if jug X is empty, fill jug X to the rim.
2) *Rule 2*: Transfer water from jug to jug
   if jug X is empty and jug Y has water, then move water from jug Y to jug X (with condition X > Y)

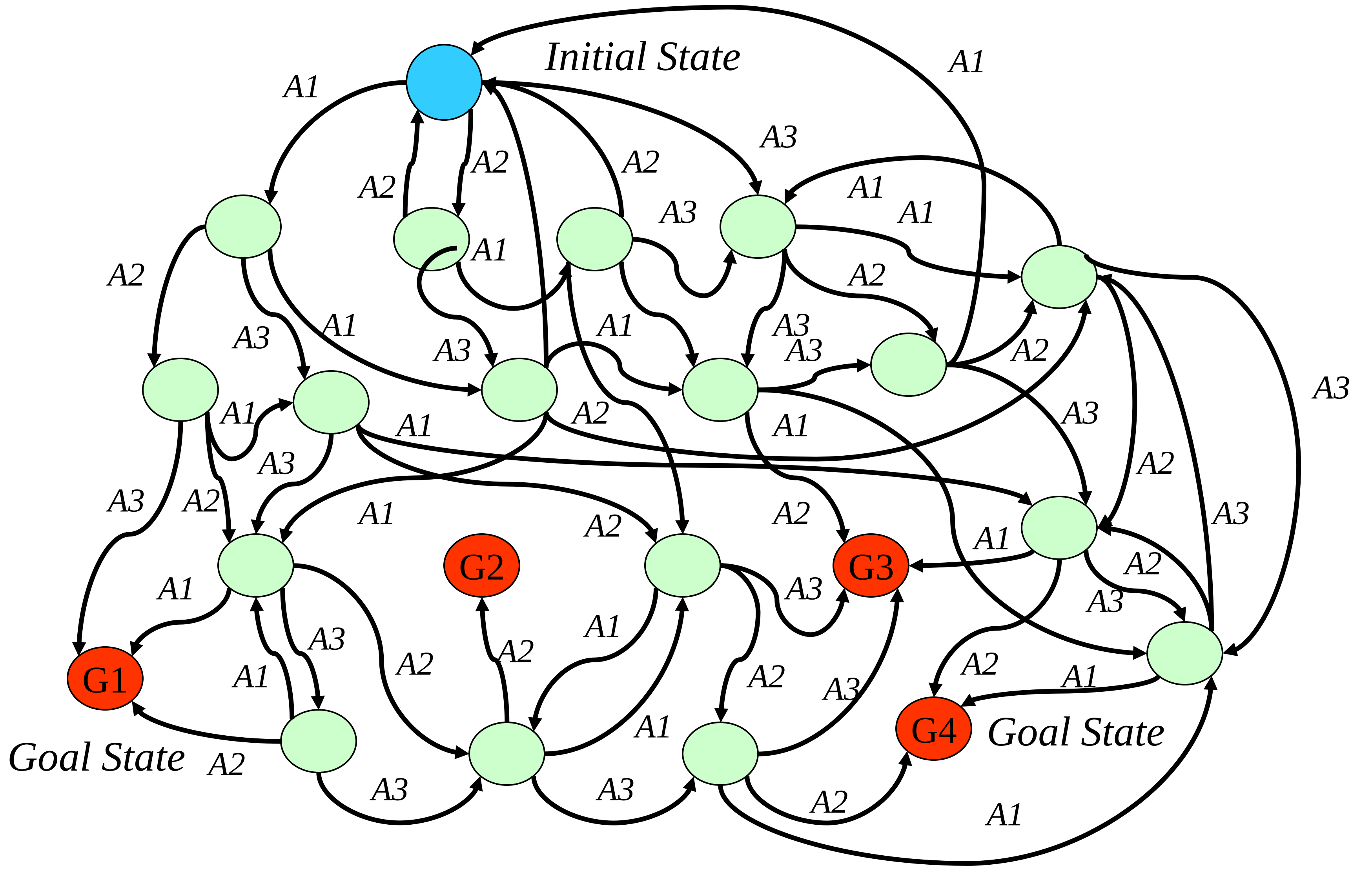
Example 2: continuation

Show the **search space** of your design <u>and the path to the solution</u>.

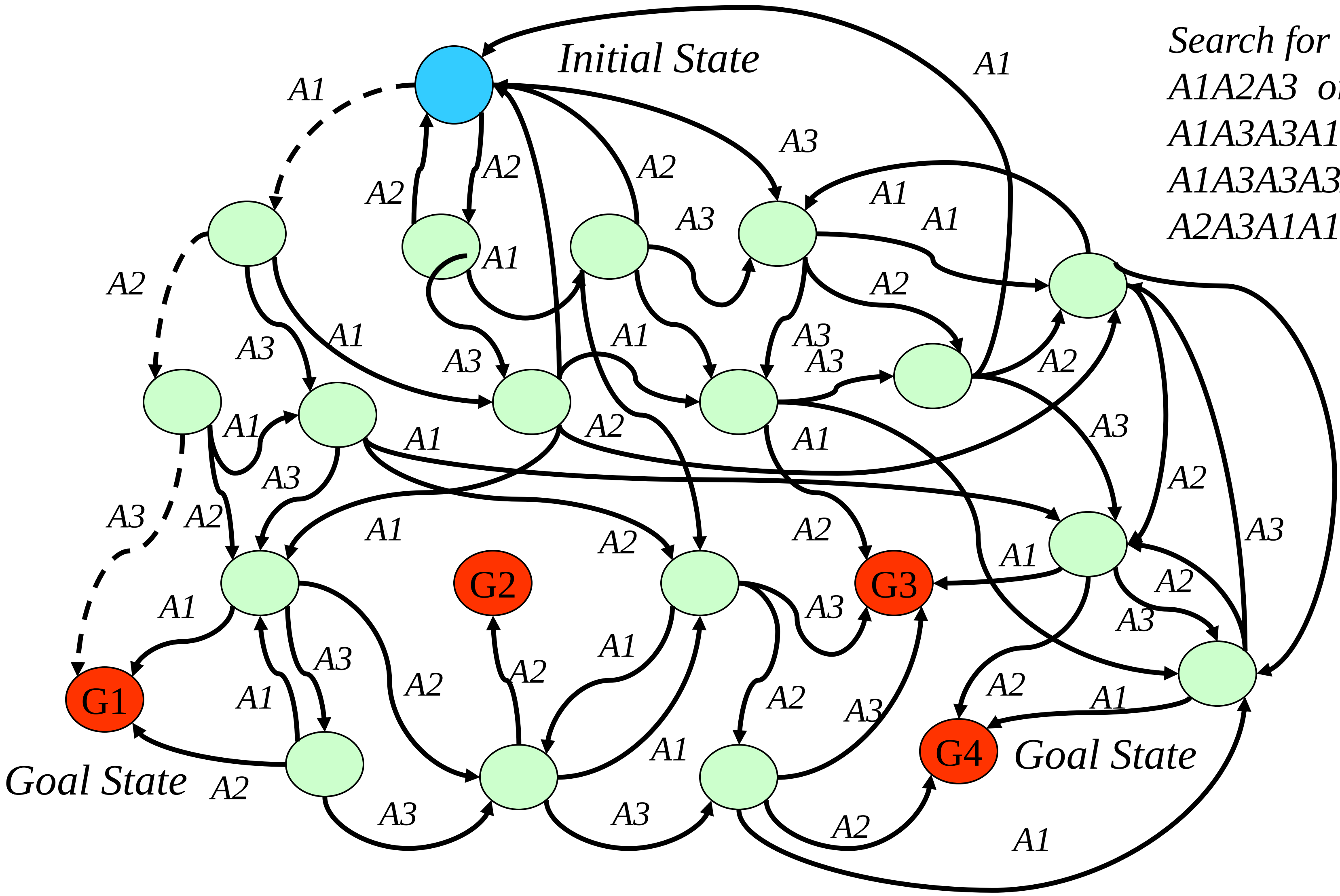
When showing the search space the first 2 levels are sufficient, but you need to show the complete path to the goal.

$$\begin{tikzcd}
 & & & & (0,0,0) \arrow[dlll, "R_1"'] \arrow[d, "R_1"] \arrow[drrr] & & & & \\
 & (0,4,0) \arrow[dl, "R_1"'] \arrow[d, "R_1"] \arrow[dr, "R_2"] & & & (100,0,0) \arrow[dl, "R_1"'] \arrow[d, "R_1"] & & & (0,0,3) \arrow[dll, "R_1"'] \arrow[dl, "R_1"] \arrow[d, "R_2"] \arrow[dr, "R_2"] & \\
(100,4,0) & (0,4,3) & (4,0,0) \arrow[d, "R_1"] & (100,4,0) & (100,0,3) & (100,0,3) & (0,4,3) & (0,3,0) & (3,0,0) \\
 & & (4,4,0) \arrow[d, "R_2"] & & & & & & \\
 & & (4,1,3) \arrow[d, "R_2"] & & & & & & \\
 & & (5,0,3) & & & & & &
\end{tikzcd}$$

Search Space for multiple goals



How can we reach G1?



Search for G1=
A1A2A3 or
A1A3A3A1 or
A1A3A3A3A2 or
A2A3A1A1

Solving problems by searching

- To solve goal-based problems, agents must search for a set of operators that move the agent from the current state to the goal state. The agent does this by looking at how it acts on its environment
- **Problem solving agents**
  - problem-solving agents are a specific class of goal-based agents. The first step in deciding how to solve a problem is to decide how the goal is to be represented. We call this **goal formulation**

Goal formulation

- What is a goal???
  - A set of world states where the goal is satisfied.
  - What determines the goal to be satisfied? A set of goal criteria.

- Example: GOAL: I want to go on vacation somewhere exotic, e.g. Los Cabos, Hawaii, US Virgin Islands. My goal shall be satisfied if I arrive in such a location (criterion 1) and if I can vacation there (criterion 2).

- Goal formulation is based on the **initial state** (where you start, e.g. Dallas) and consists of a **set of world states** that satisfy the <u>goal criteria.</u>

Goals and Actions

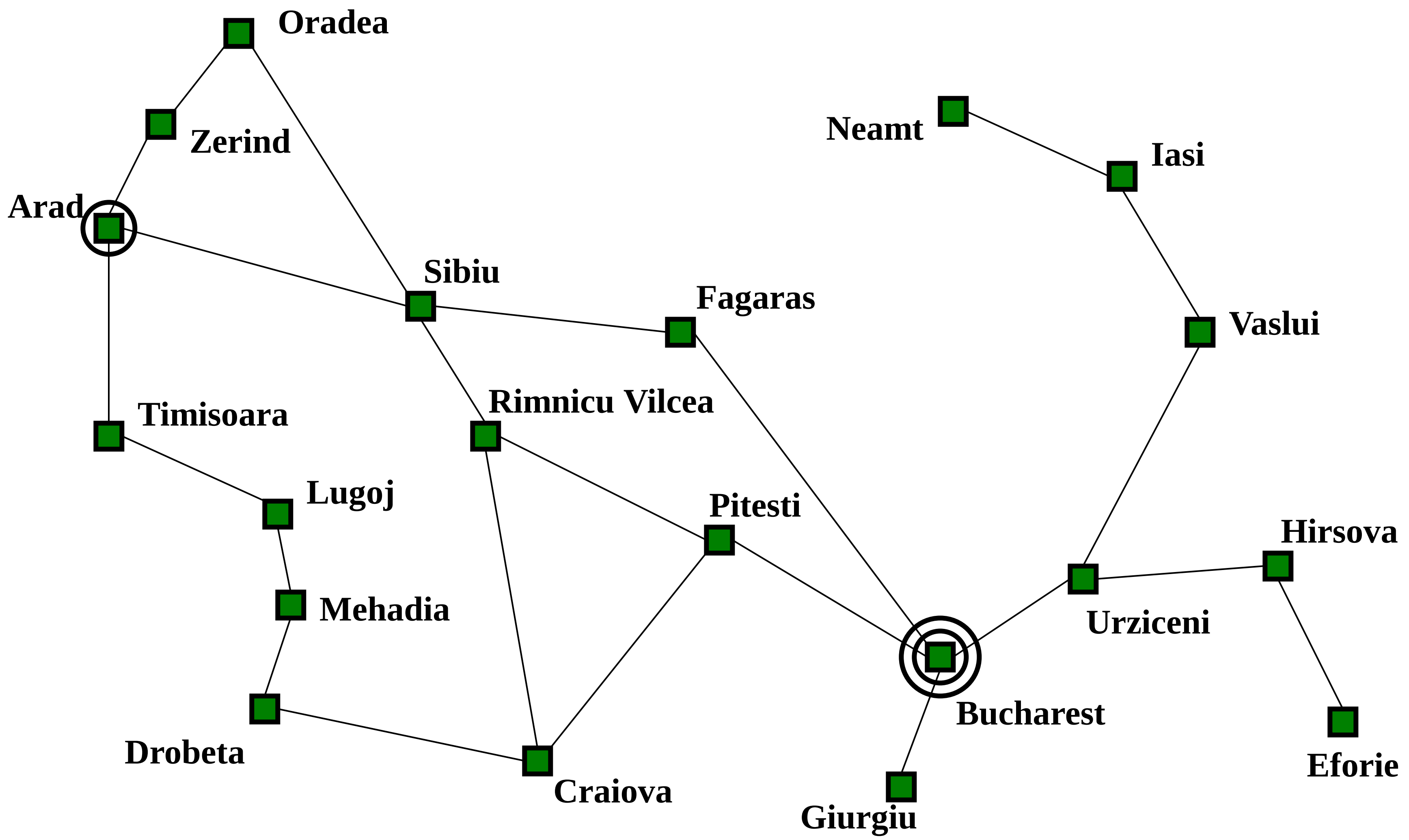
- We view **actions** as causing transitions between world states
  - The agent job is to decide what sequence of actions are needed to achieve the goal state. In doing this, we must decide how much of the world is necessary for solving the problem.
  - Example: being able to board a flight, taking a drive, etc.
  - Typically, a lot of information is thrown out. For instance, if you want to plan a driving route from one city to the next. You would like to use a map and measure distances between intervening cities. In your plan, you would not consider whether the car windows are going to be up or down, or whether the radio playing or not. This part of the world state doesn't change the task of moving between intervening cities. We call this process **abstraction**.

Selecting a state space

- Real world is absurdly complex

  $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) operator = complex combination of real actions
  - e.g. "Arad $\rightarrow$Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, <u>any</u> 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!

Example: Romania



Problem solving by search

- Problem formulation
  - The next step is to determine how to formulate the problem itself. This involves the search process to determine the move sequence for moving from the current state to a state of known value. This is typically the goal state, but may be an intermediate state.
- Search algorithm
  - you can use any search algorithm to solve a problem. The search algorithm has as input the problem you have formulated and it outputs an action sequence to reach the goal state
- Execution phase
  - in the execution phase, the agent carries out the action sequence found by the search algorithm

Formal Problem formulation

A *problem* is defined by five items:

1. *initial state*
   - e.g. *In(Arad)*
2. *actions* - given a state *s*, *Action(s)* returns the set of actions that can be executed in *s*
   - e.g. go(Zerind), go(Sibiu), etc.
3. *transition model* which describes what each action *a* does, and it is specified by a function *RESULTS (s, a)* that returns the state that results from doing action *a* in state *s*
   - e.g. *RESULTS(In(Arad), Go(Zerind)) = In (Zerind)*
   - *The state space is a directed graph in which the nodes are states and the links are actions.* ***A path*** *in this graph is a sequence of actions!*
4. *Goal test* which determines if a given state is a goal state, e.g. *In (Bucharest)*
5. *path cost* (additive) is a function which assigns a numeric cost to each path.

➢ A *solution* is a sequence of operators leading from the initial state to a goal state

Problem-solving agents

- Simple problem solving agent:

```
function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns an action
  persistent:  seq, an action sequence, initially empty
               state, some description of the current world state
               g, a goal, initially null
               problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     g ← 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
```

Note: the *environment is static, observable, discrete and deterministic.*

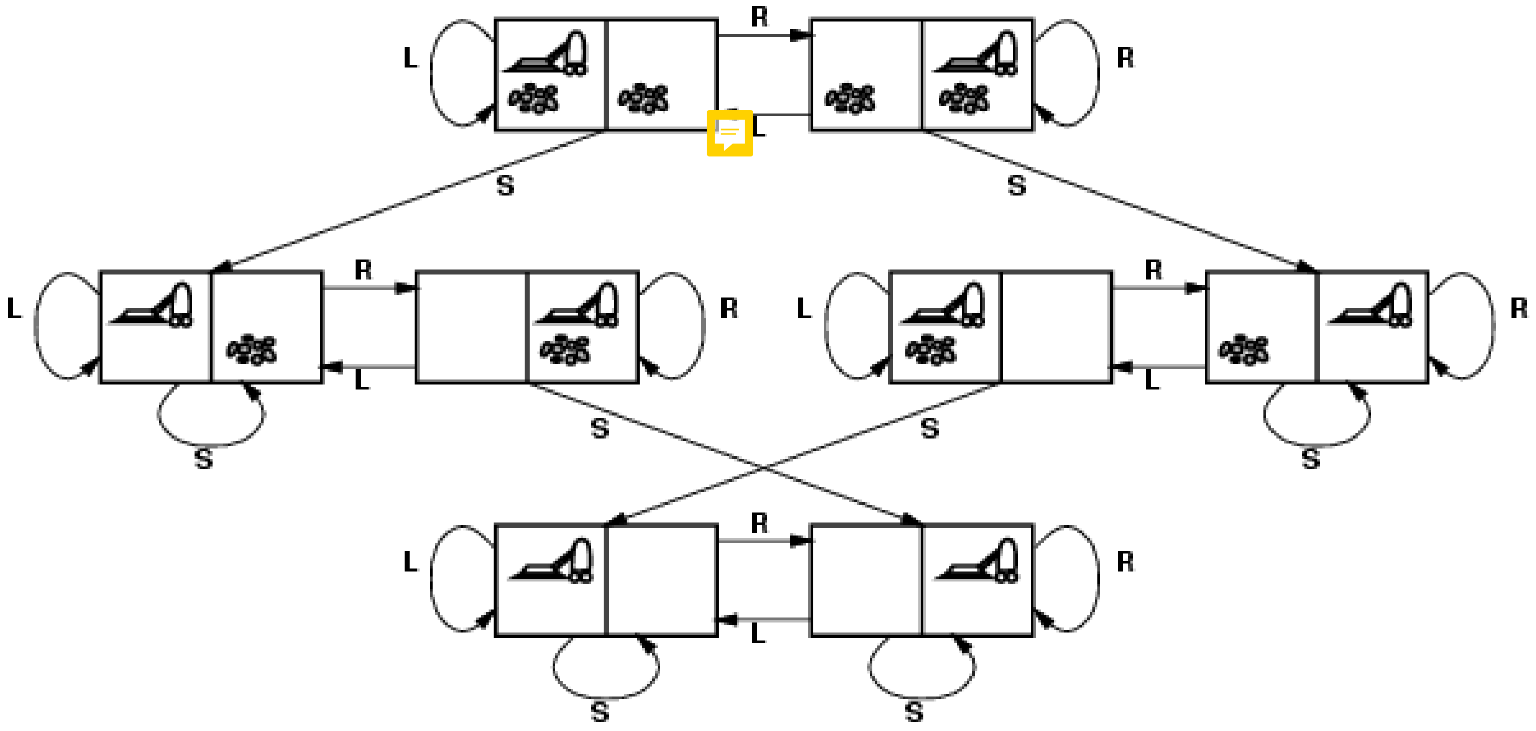
- *Formulates the goal and the problem, then searches for the sequence of actions that would solve the problem, and then executes actions, 1 at a time*

Example problems

- Toy problems are used to represent/demonstrate a concept, like a particular search algorithm.
- Real world problems, however, are much more difficult (and interesting) to solve. In addition, the formulation of the problem is not universally agreed upon. Different people would represent the problem in different ways.
- Toy problems
  - see the book for details: 8-puzzle, 8-queens (3 different problem formulations)

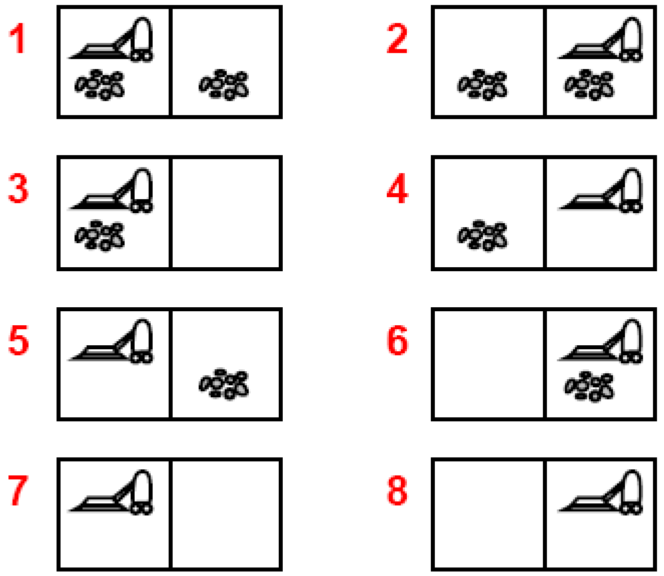
Example: vacuum world

- State space



Example: vacuum world

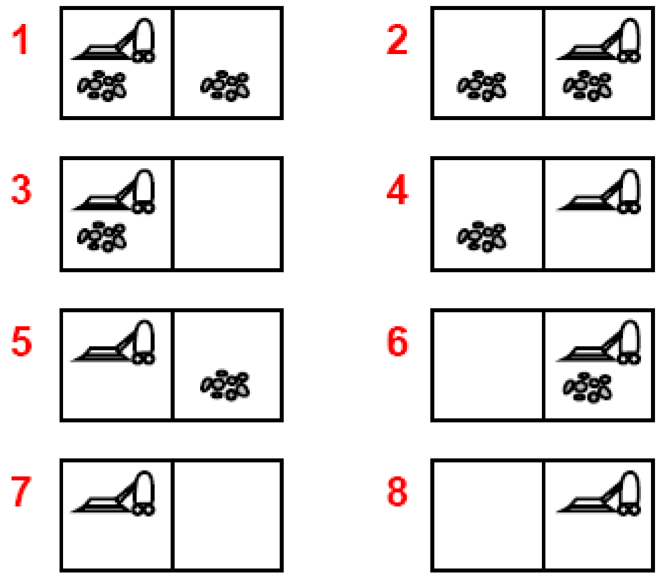
Single-state, start in #5. Solution??



Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??

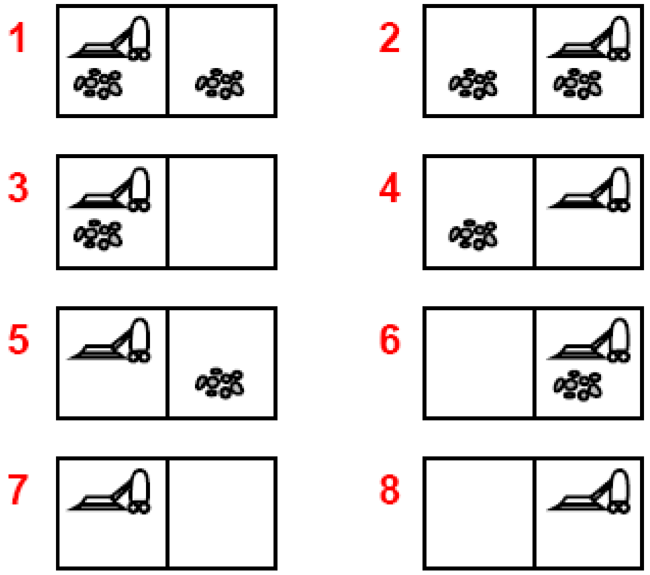


Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$

Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??

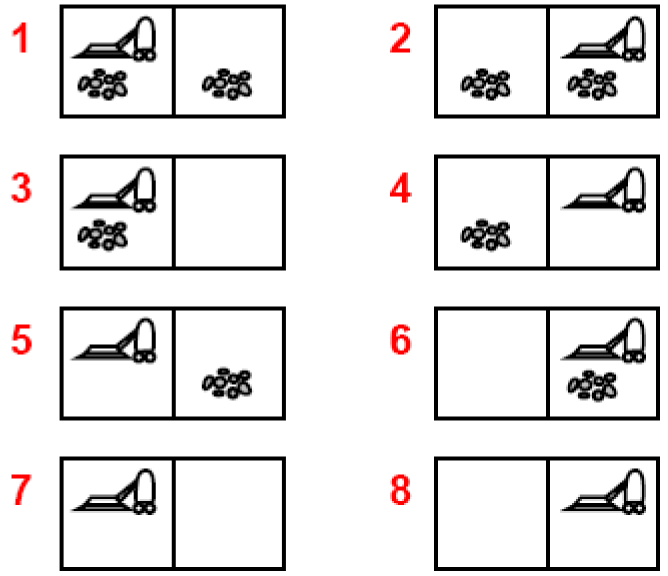


Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1,2,3,4,5,6,7,8\}$
e.g., $Right$ goes to $\{2,4,6,8\}$. Solution??
$[Right, Suck, Left, Suck]$

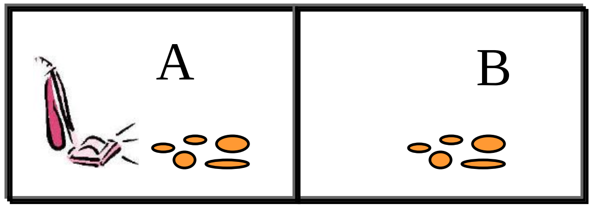
Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
$[Right, \textbf{if } dirt \textbf{ then } Suck]$



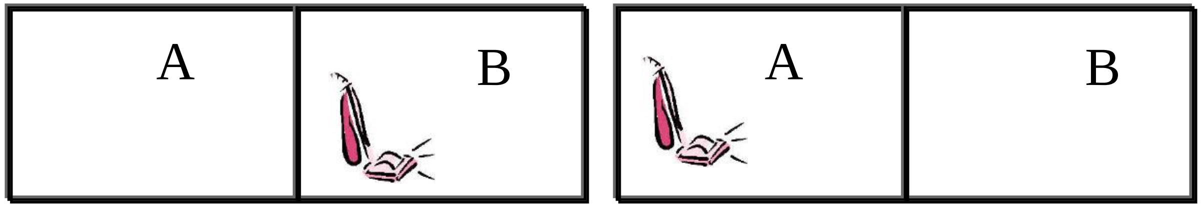
Formulation of vacuum world

- *States:* 2 locations, 4 conditions: dirt and vacuum in the same place, no dirt, dirt and vacuum in different places, dirt in both places:: #states = 2 × 4 = 8 states

1. *Initial state*

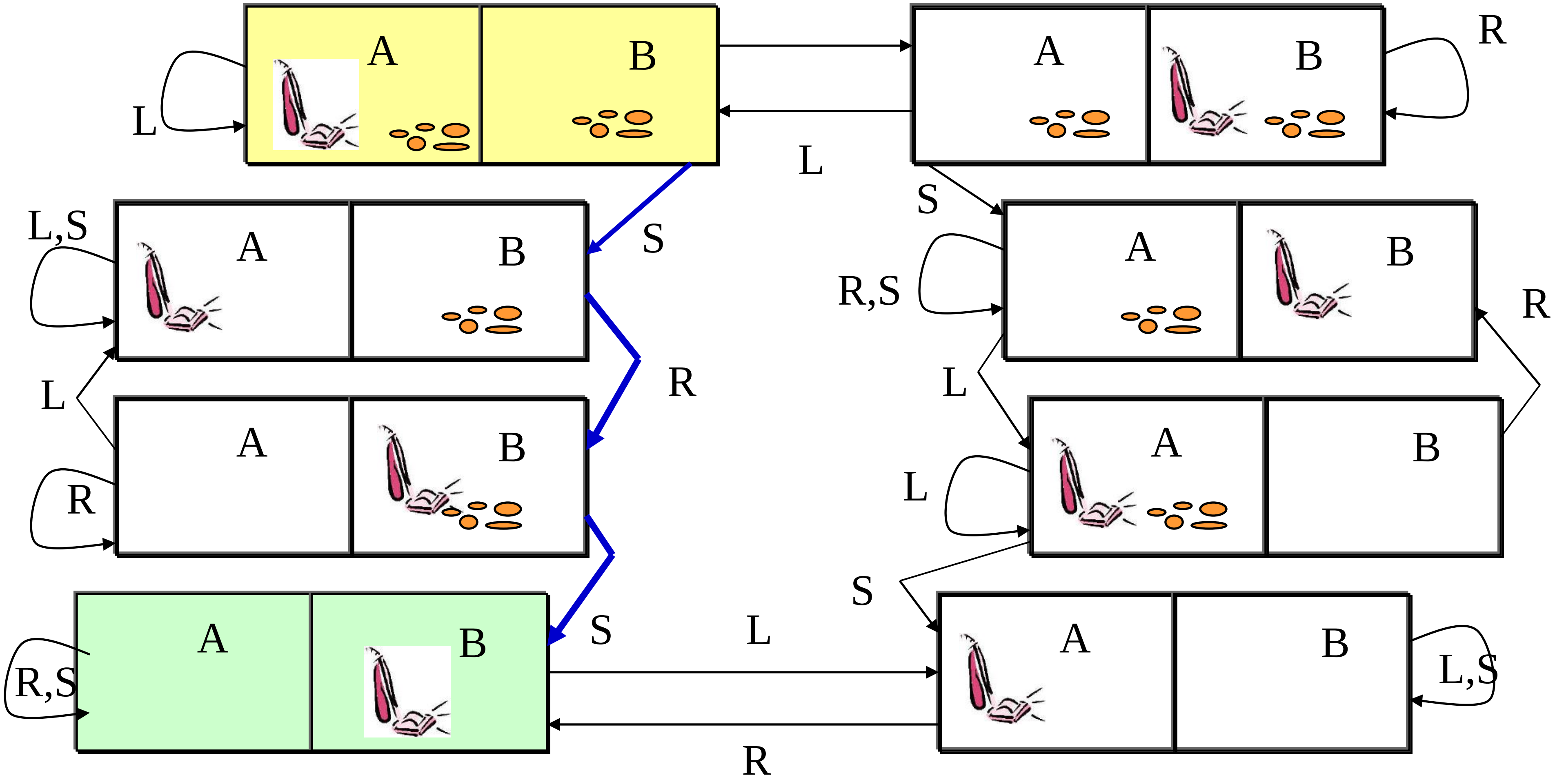


2. *actions* – {left, right, suck}
3. *Transition model:* *Result(A,left)=B,…*
4. *Goal test:*

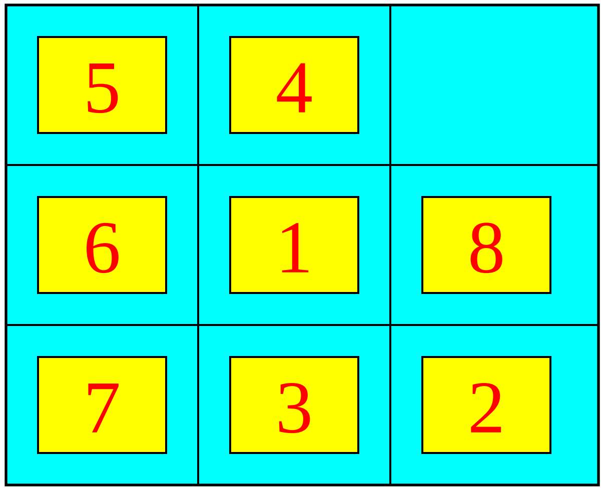


5. *Path cost* *– each step costs 1, path cost is # of steps*

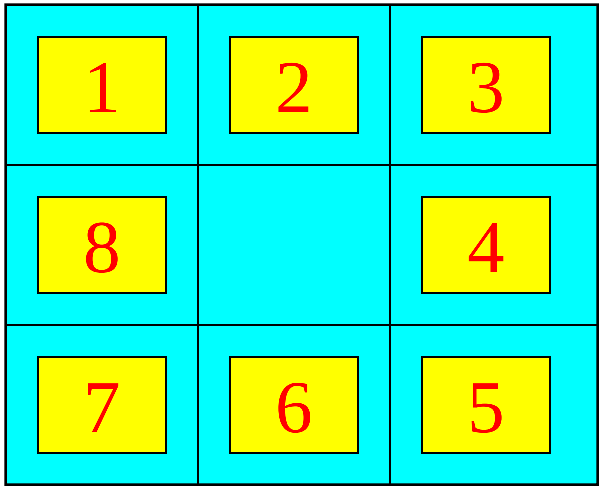
Path cost = 3



Example: the 8-puzzle



Start state



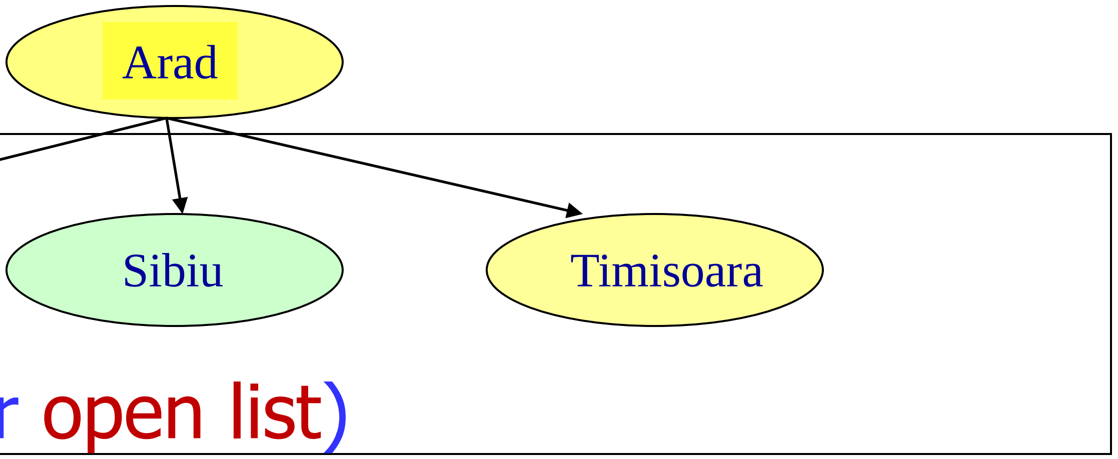
Goal state

- states??: integer locations of tiles (ignore intermediate positions)
- actions??: move blank left, right, up, down (ignore unjamming etc.)
- goal test??: - goal state (given)
- path cost??: 1 per move

[Note: optimal solution of n-Puzzle family is NP-hard]

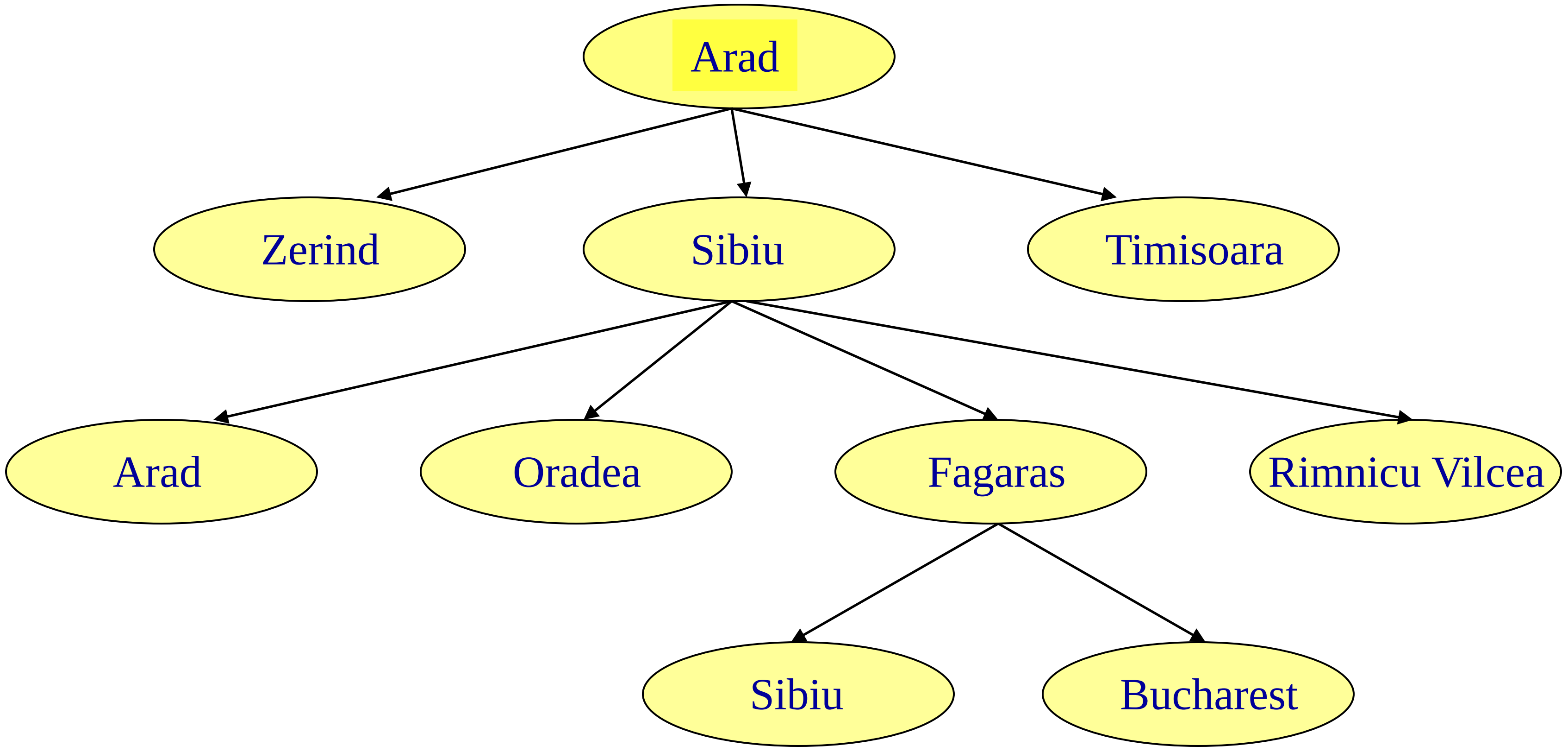
Searching *for solutions*

- *SOLUTION: an action sequence (from initial node to goal)*
- *Search Node* – the current node, initial node first
- *Expanding* the current state by applying each legal action and generating a new set of states
- *Search Strategy*: the choice of which of the new set of states to expand



The frontier is the collection of nodes that have been generated but not yet expanded!!!

General search example

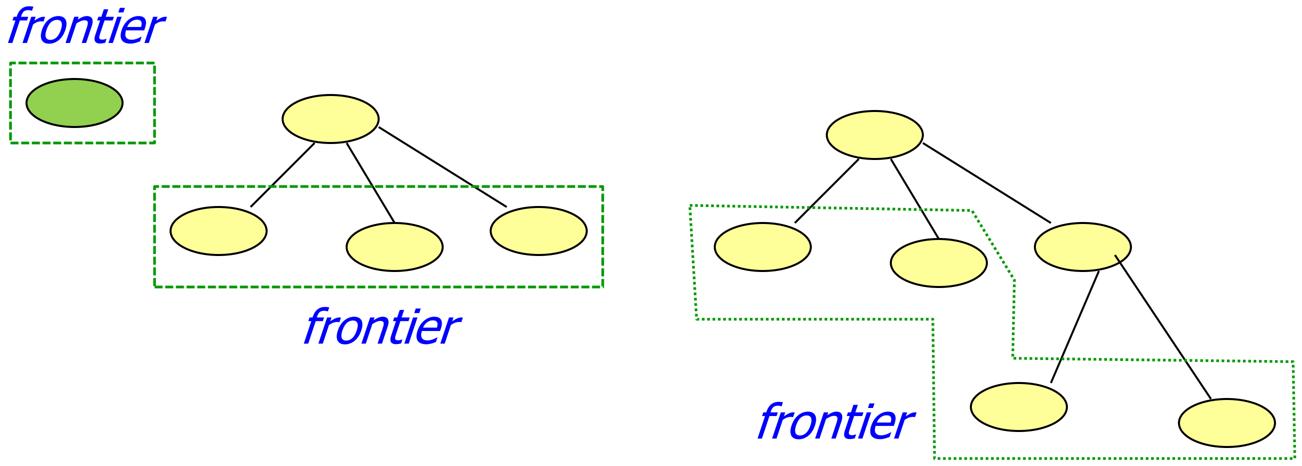


In(Arad) is a repeated state!!!!

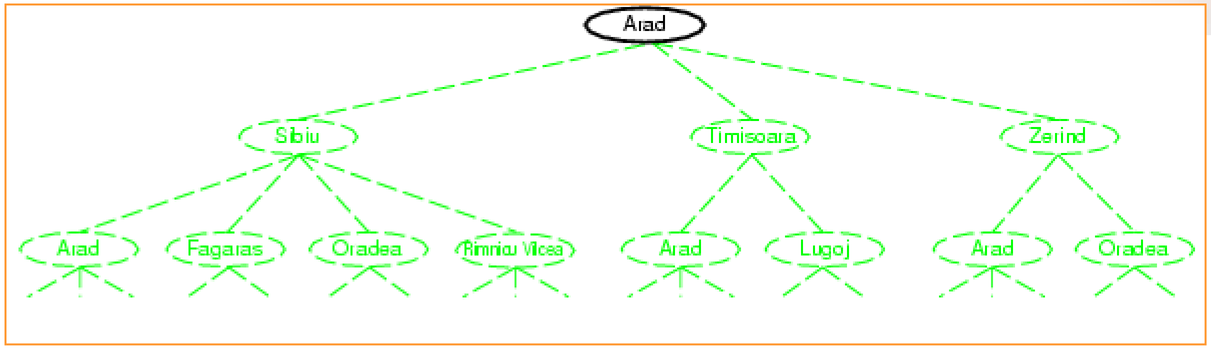
Search Trees

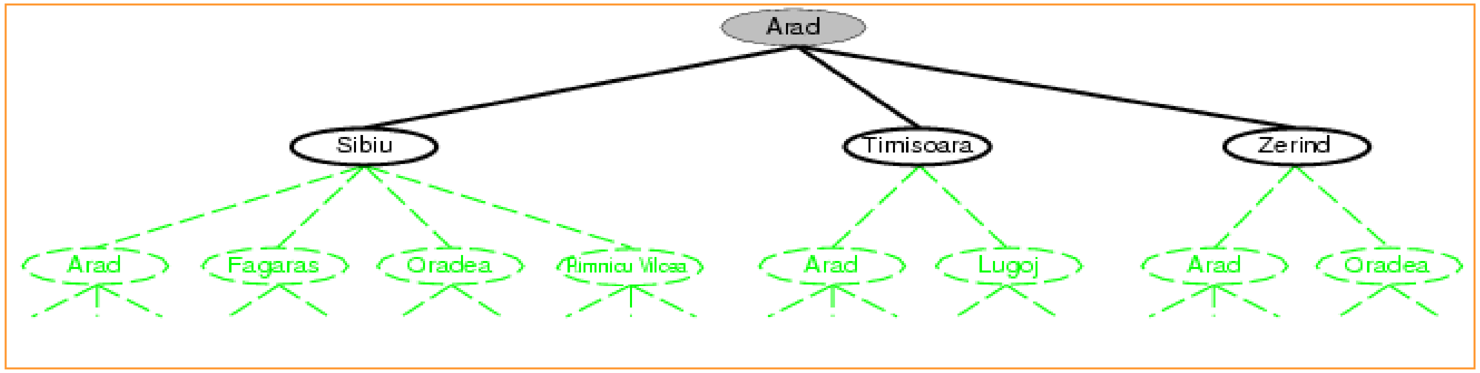
Algorithm 1

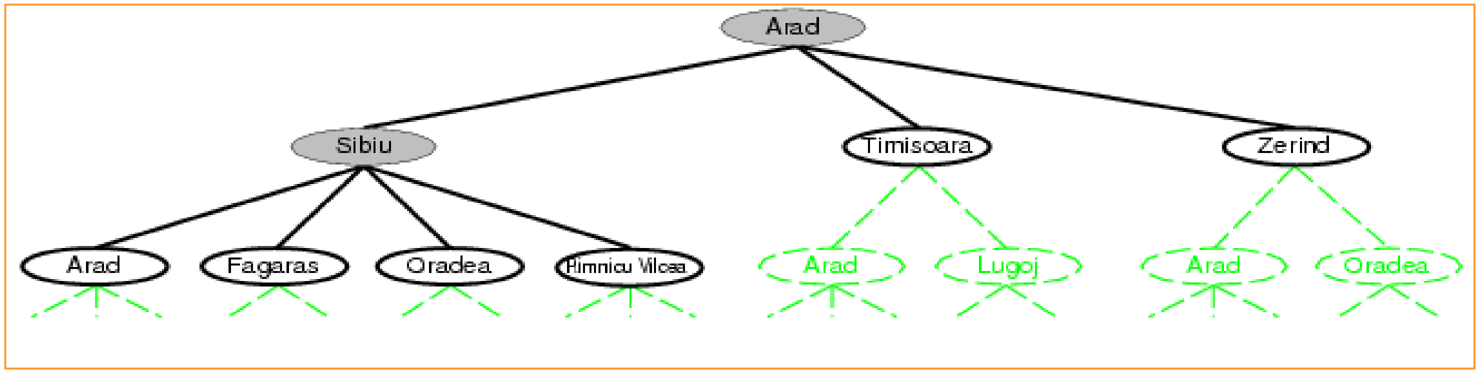
```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return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



Partial Search Trees







In(Arad) is a repeated state!!!

Repeated states !!!

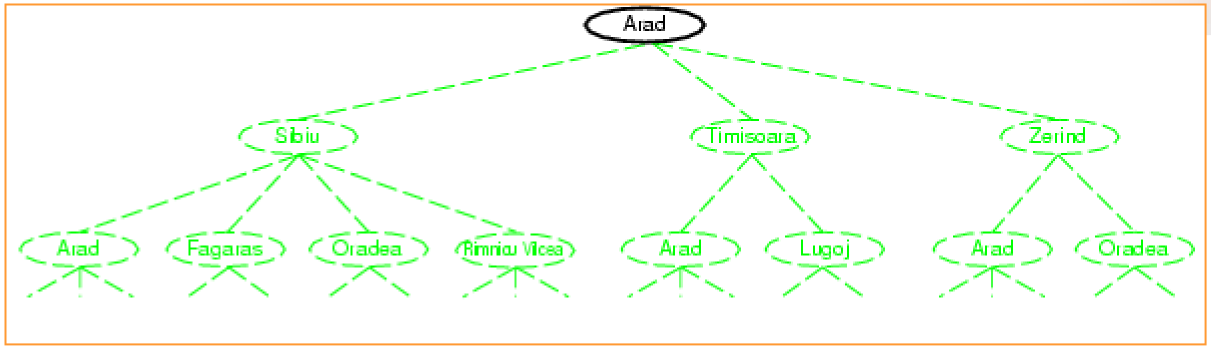
- Generate *loopy paths*!!!!
- Complete search becomes infinite!!!!
- Loopy paths are a special case of *redundant paths*, which exist whenever there is more than one way to get from a state to another.
- To AVOID loopy paths, we have to remember where we have been!!! How???
  - Augment the Tree-Search algorithm with a data structure called *explored set (a.k.a closed set)* which remembers every explored set.

Graph-Search algorithm

Algorithm 2

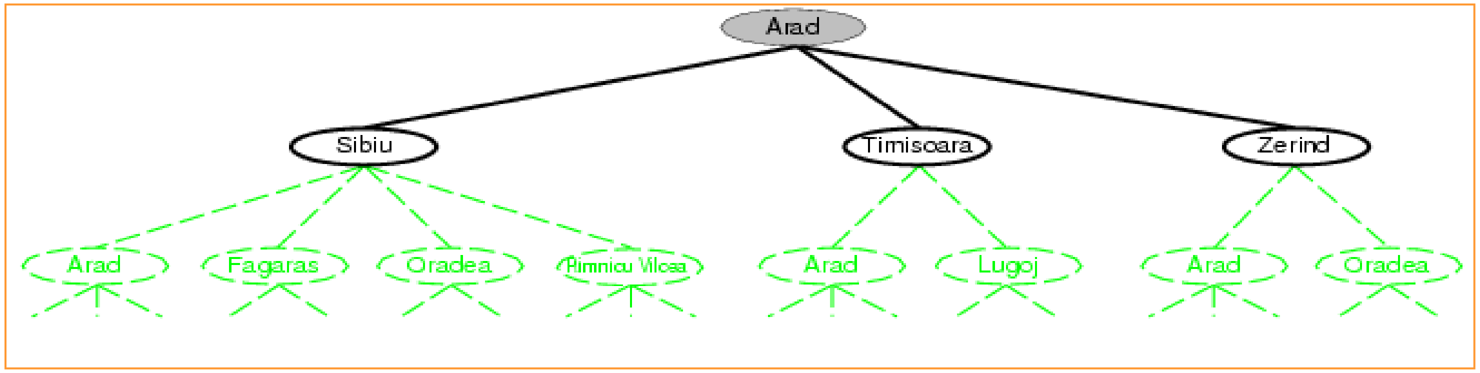
```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to empty
    loop do
        if frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if 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
```

Partial Search Trees



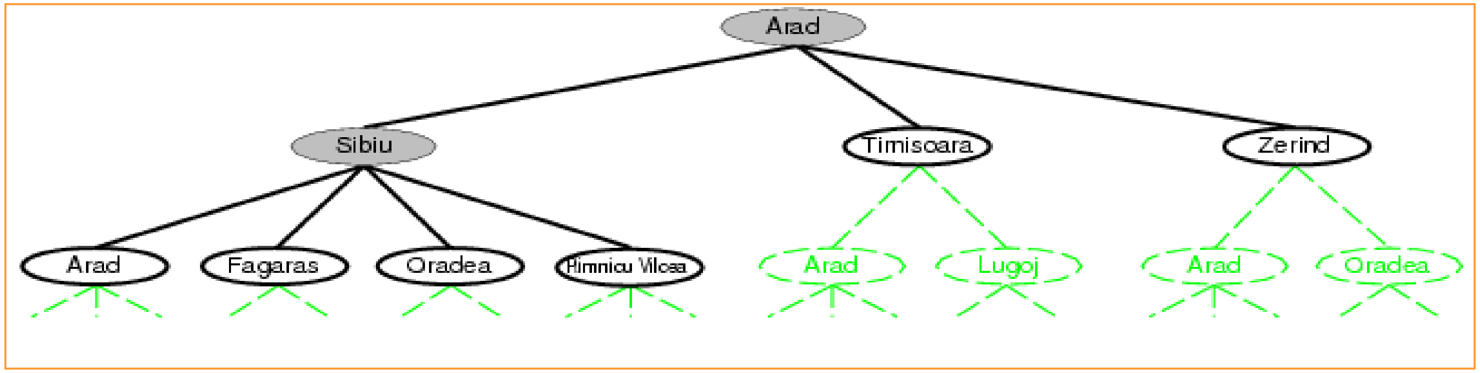
Explored = { }

Frontier = {Arad}



Explored = {Arad}

Frontier = {Sibiu, Timisoara, Zerind}



Explored = {Arad, Sibiu}

Frontier = {Fagaras, Oradea, Rimnicu Vilcea, Timisoara, Zerind}

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. In each node n of the tree, we have a structure that contains 4 components:

- **n.STATE** – to which state in the state space this *node* corresponds
- **n.PARENT** – the node in the search tree that generated this *node*
- **n.ACTION** – the action that was applied to the parent to generate this *node*
- **n.PATH-COST** – the cost g(n) of the path from the initial node to this *node*

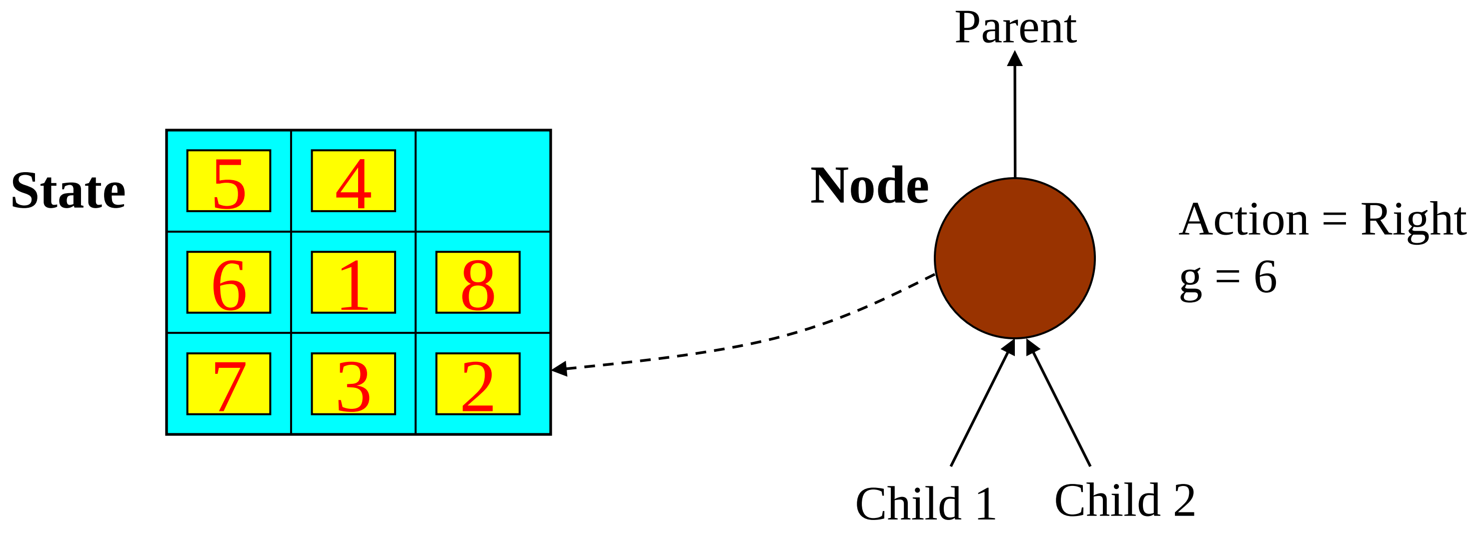
How we generate children nodes??

Given the components of the parent node:

```
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)
```

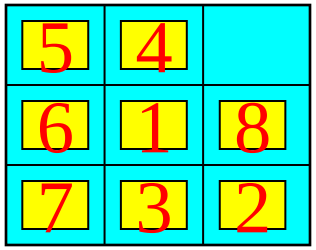
states vs. nodes

- A *state* belongs to the search space
- A *node* is the data structure from which the search tree is considered:
  - Each node has a *parent, children, path cost g(x), ….*



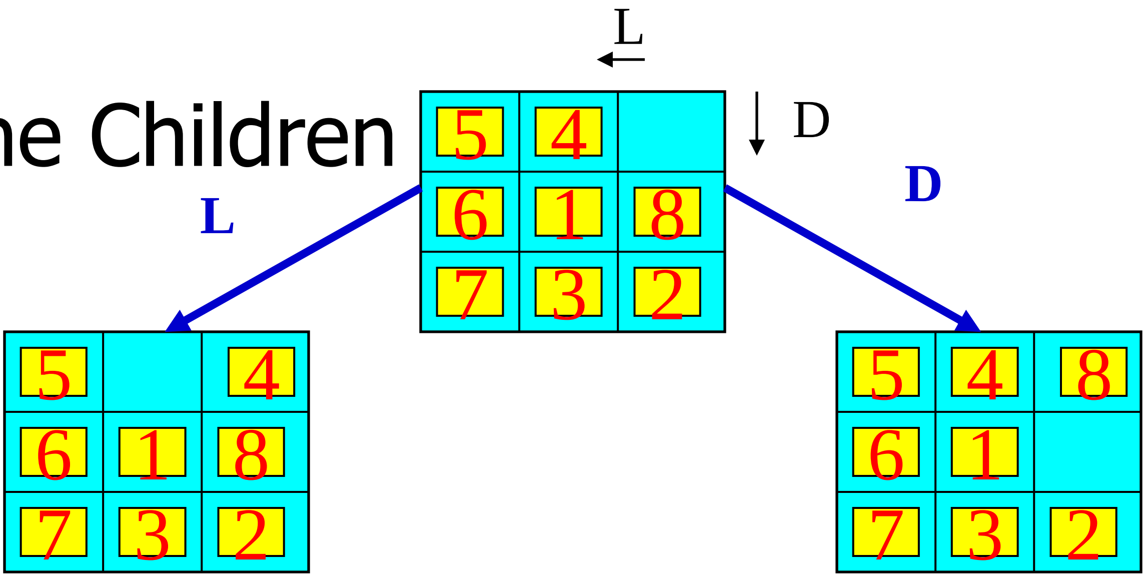
Parents and children

- The node:



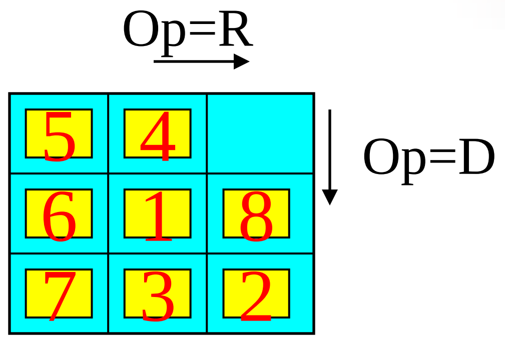
- The Father ???

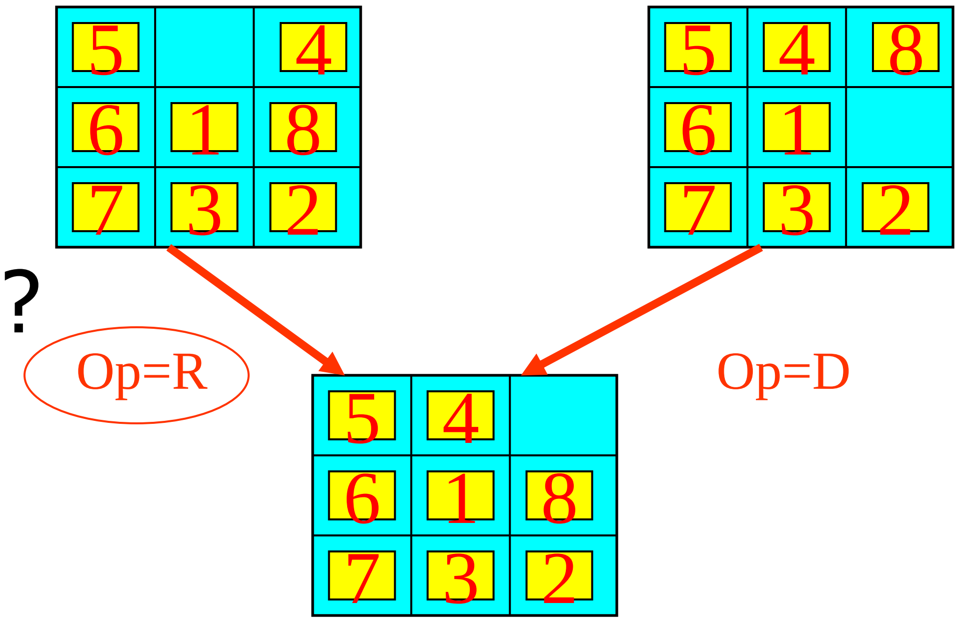
- The Children



How many fathers?

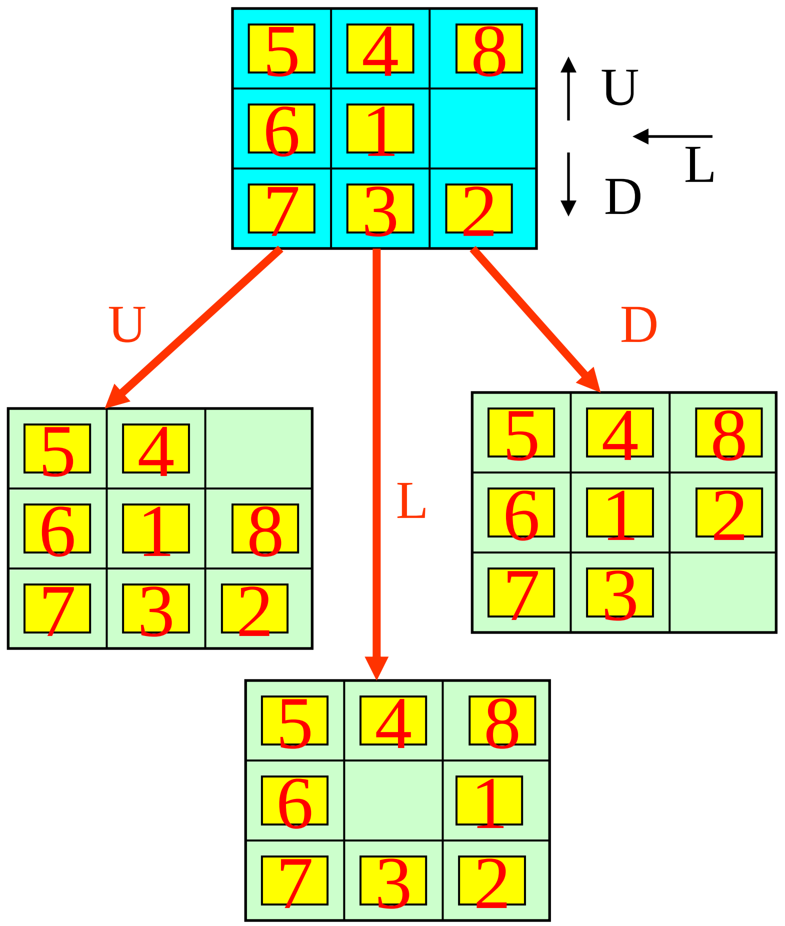
- The node
- Potential???
- Real ???

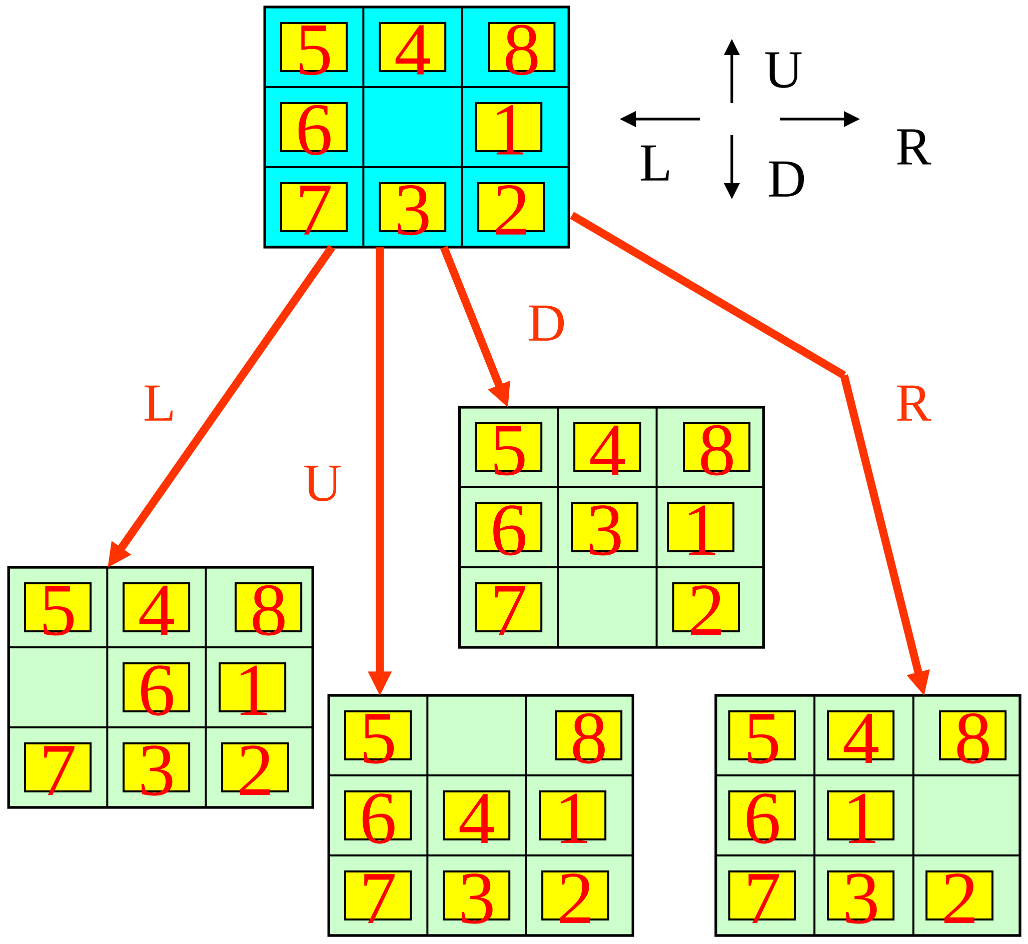




How many possible children?

- 2 or 3 or 4





Where do we put the nodes?

- The frontier needs to be stored in a way that allows the search algorithm to easily chose the next node to expand, according to the <u>preferred strategy</u>. The appropriate data structure is the *queue!*
- *Operations on a queue are:*
  - EMPTY?(*queue*)
  - POP(*queue*)
  - INSERT(*element, queue*)
- Queues are characterized by the order in which they store inserted nodes:
  - FIFO queue
  - LIFO queue
  - Priority queue

Search strategies

- 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?
  - time complexity – how long does it take to find a solution?
  - space complexity – how much memory is needed to find a solution?
  - 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

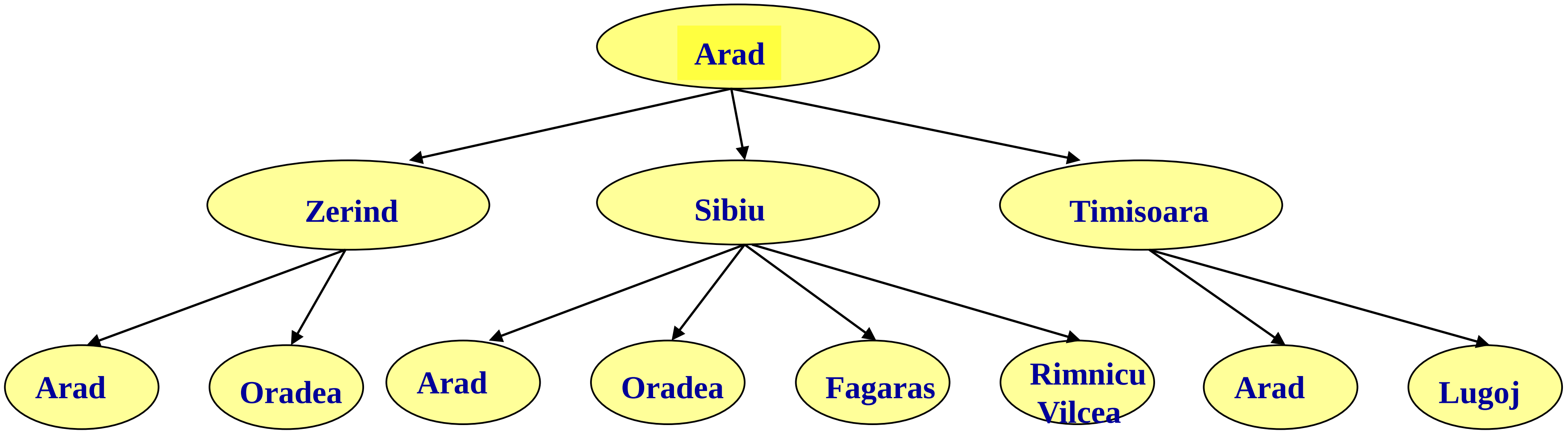
- *Uninformed* strategies use only the information available in the problem formulation
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Breadth-first search

- Expand the root node first, then all its successors, and their successors, etc. The <u>goal test </u>is applied to each node when it is generated!!!!!

<u>Implementation</u>:

- Start with the *frontier=null* and a *FIFO queue*
- *[1] expanded={Arad} frontier={Zerind, Sibiu, Timisoara}*
- *[2] expanded={Arad,Zerind} frontier={Sibiu,Timisoara,Arad,Oradea}*
- *[3] expanded={Arad,Zerind,Sibiu} frontier={Timisoara,Arad,Oradea,Arad,Oradea,Fagaras,Rm Vilcea}*



Breadth-first search

Expand shallowest unexpanded node

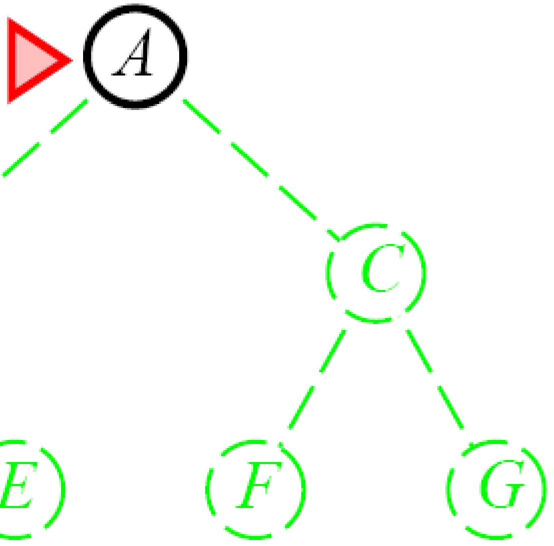
Frontier initially= NULL

Implementation:

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

Expanded={A}

Frontier = {B,C}



Breadth-first search

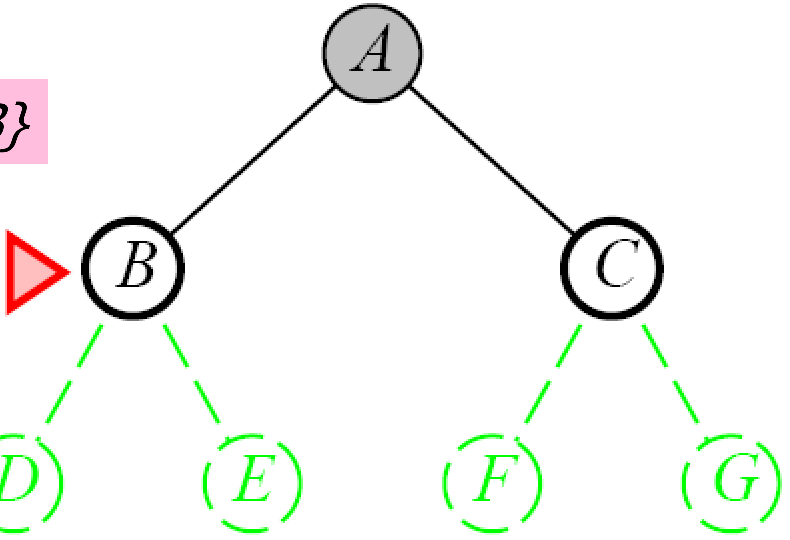
Expand shallowest unexpanded node

Implementation:

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

Expanded={A,B}

Frontier = {C, D, E}



Breadth-first search

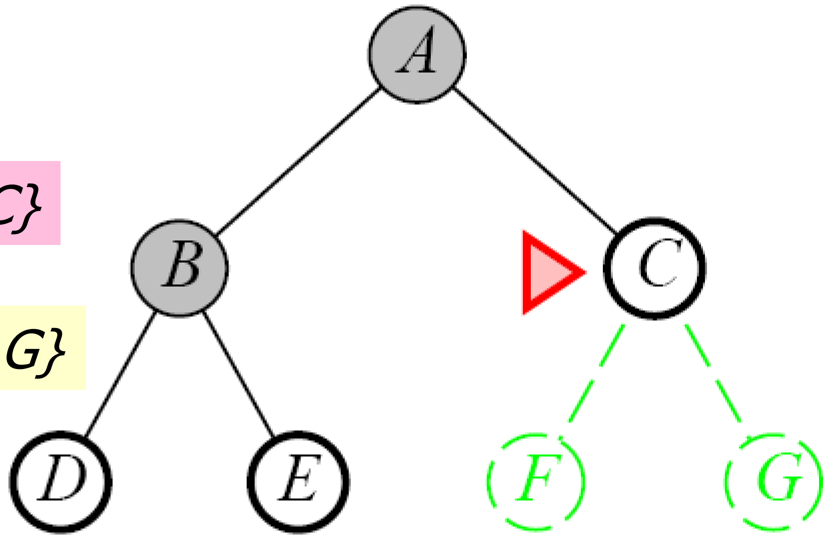
Expand shallowest unexpanded node

Implementation:

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

Expanded={A,B,C}

Frontier = {D, E, F, G}



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)
```

Properties of breadth-first search

- Complete ?? Yes (if the shallowest goal is at some depth *d)*
- Space ?? $b+b^2+b^3+\ldots+b^d+(b^d-b)=O(b^{d+1})$, i.e. exponential in *d – (keeps every node in memory)*
- Time ?? 10000 nodes per second
- Optimal ?? Yes if the path-cost is a non-decreasing function (if cost = 1 per step); not optimal in general

- *Memory requirements are a bigger problem than time requirements;*

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
*C** - the cost of the optimal solution

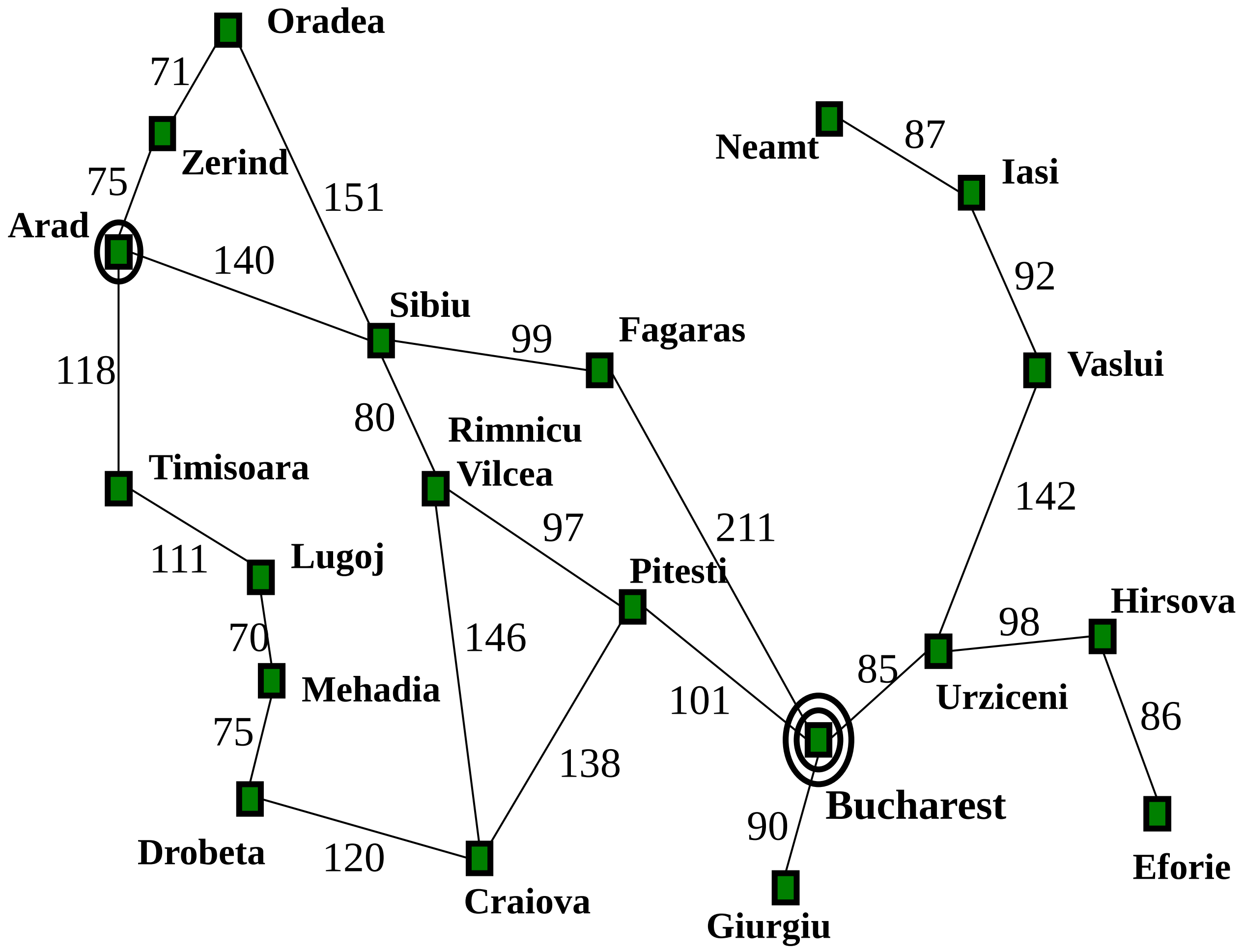
Uniform-cost search

- If *not all step costs are equal*, a simple improvement is possible
  - Always select the node with the *lowest path cost (g)* $\rightarrow$ *the frontier queue will be ordered by cost!!! (in increasing order)*
  - *2 significant differences from breadth-first search:*
    1. *The goal test is applied to a node when it is selected (not when generated!!!)*
    2. *A test is added in case a better path is found to a node currently on the frotntier.*

Uniform-Cost Search

```
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
```

Romania with step costs in km

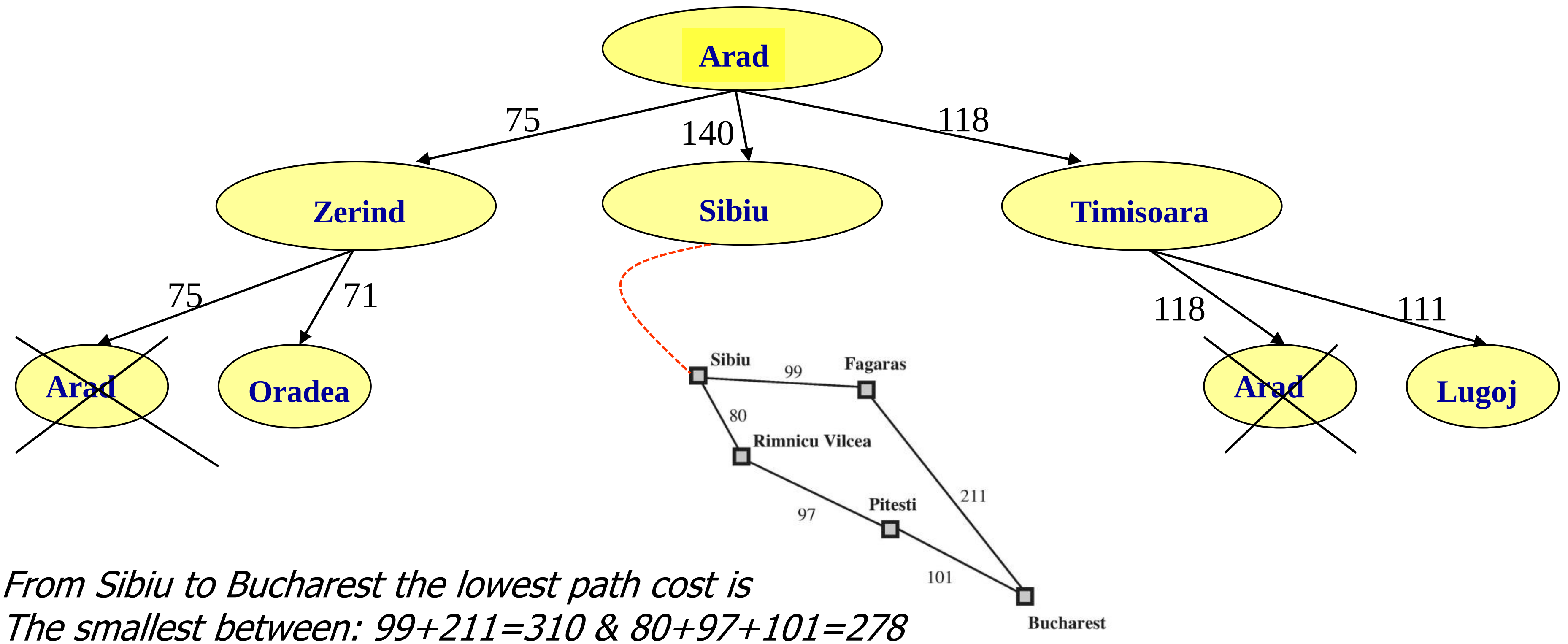


Straight-line distance to Bucharest

Arad	**366**
Bucharest	**0**
Craiova	**160**
Drobeta	**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**

Uniform-cost search

- Expand node with lowest path cost



From Sibiu to Bucharest the lowest path cost is
The smallest between: 99+211=310 & 80+97+101=278
From Arad to Bucharest, the lowest path cost is
140+278=418.

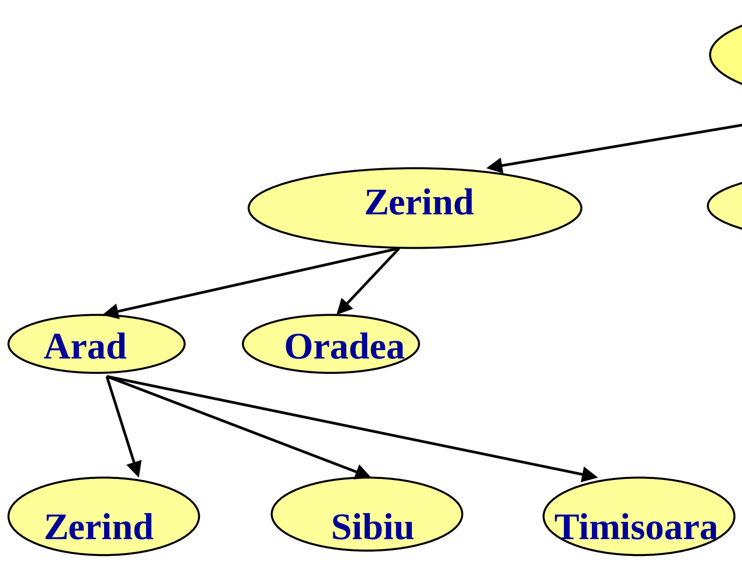
Properties of uniform-cost search

- Complete ?? Yes, if step cost $\geq \varepsilon$
- Time ?? If C* is the cost of the optimal solution, $O(b^{\lceil C/\varepsilon \rceil}) >> b^d$
- Space ?? Same as time complexity
- Optimal ?? Yes

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
*C** - the cost of the optimal solution

Depth-first search

- Expand deepest node in current frontier



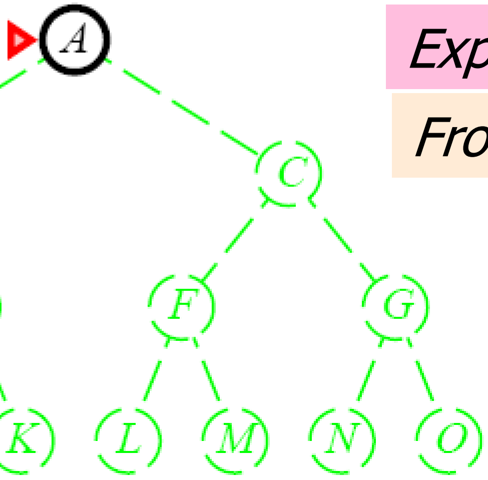
- I.e. depth-first search performs infinite cyclic excursions
- Need a finite, non-cyclic search space (or repeated state-checking)

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A}

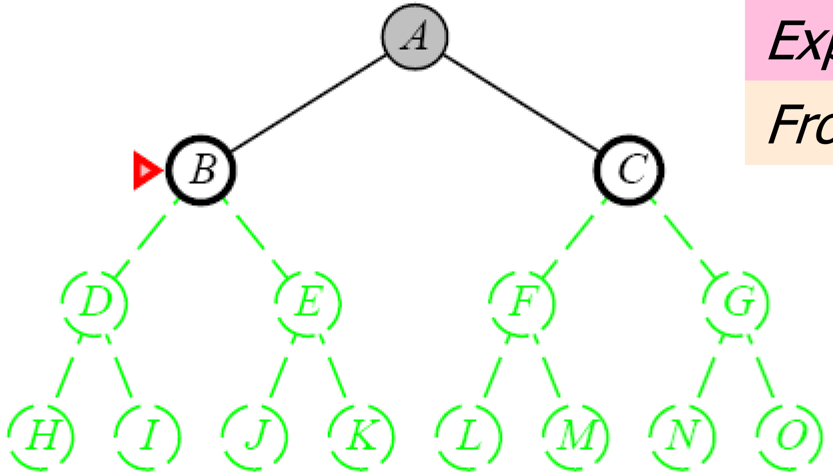
Frontier = {B,C}

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B}

Frontier = {D,E,C}

Depth-first search

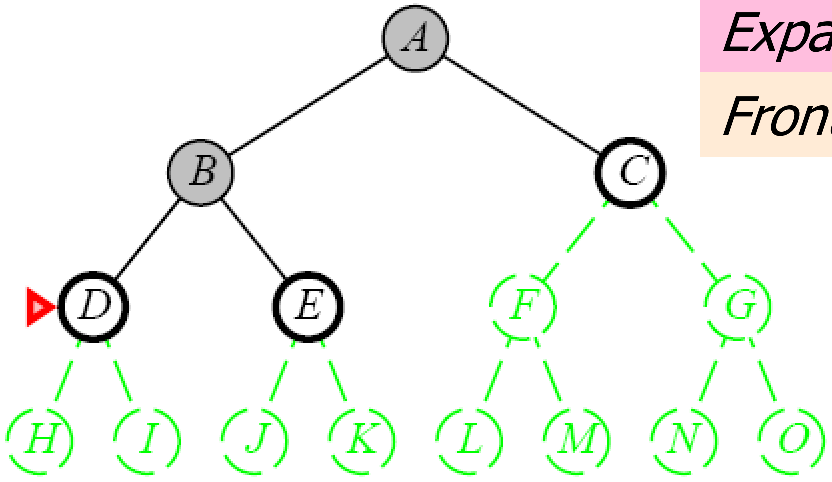
Expand deepest unexpanded node

Implementation:

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

Expanded={A,B,D}

Frontier = {H,I,E,C}

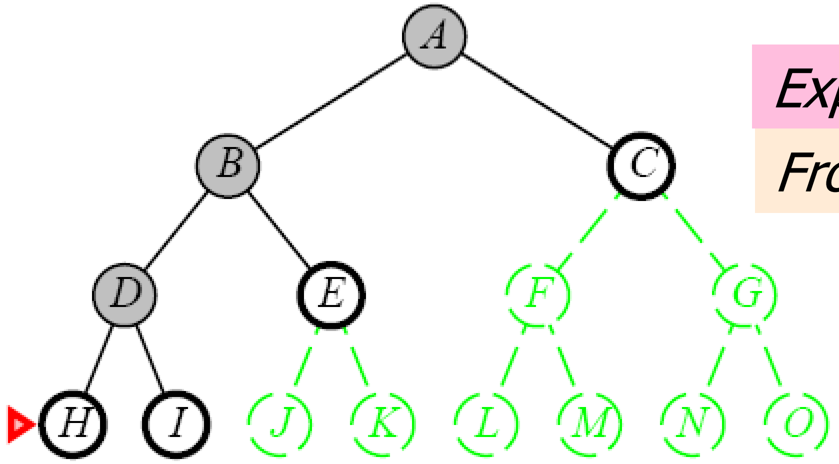


Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B,D,H}

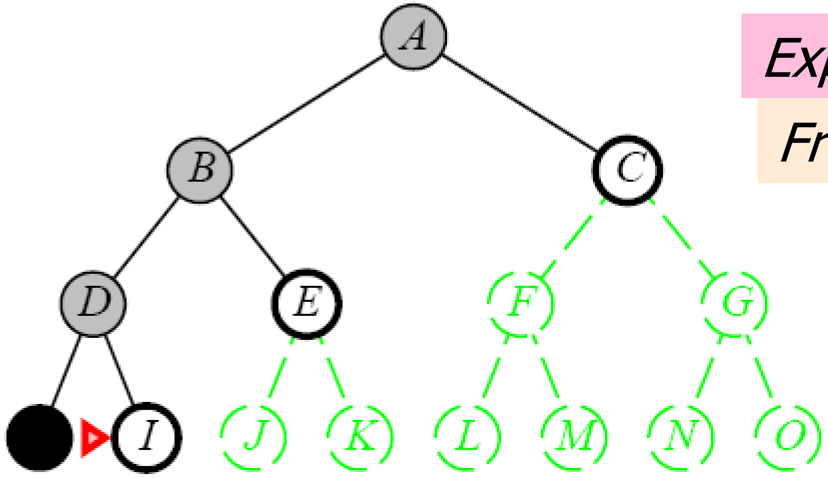
Frontier = {I,E,C}

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B,D,H,I}

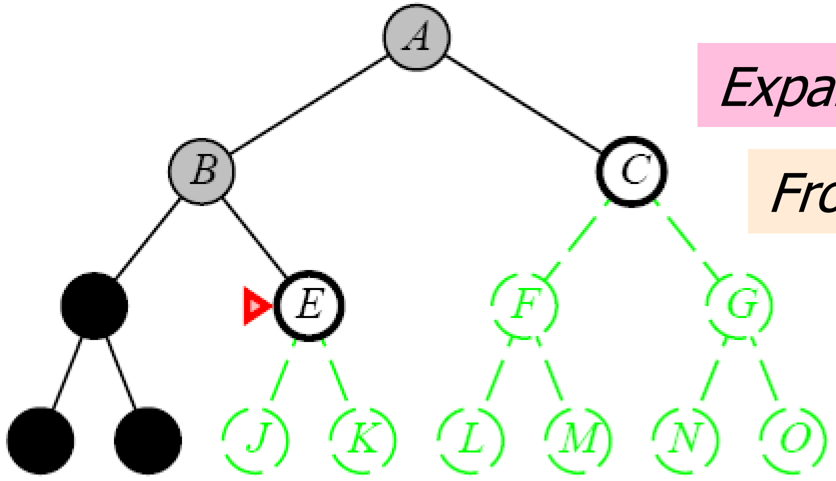
Frontier = {E,C}

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B,D,H,I,E}

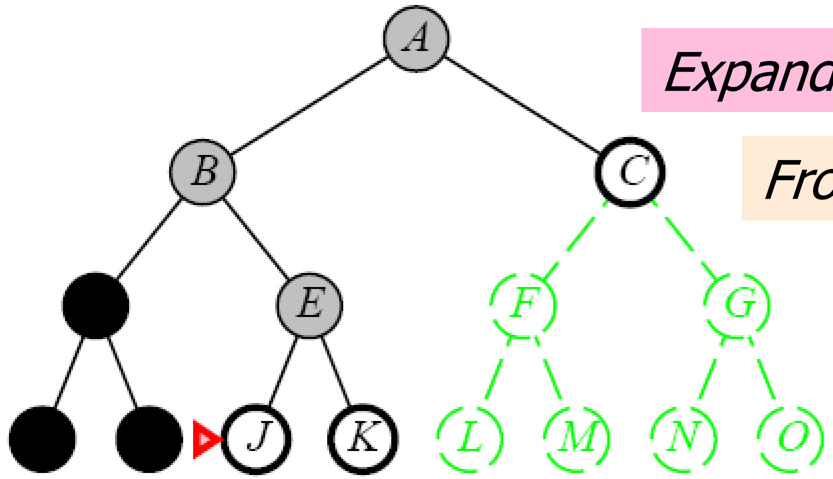
Frontier = {J,K,C}

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B,D,H,I,E,J}

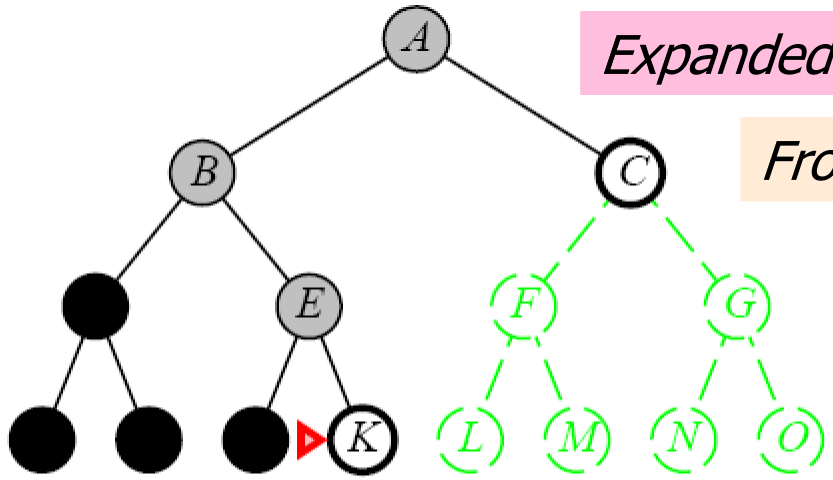
Frontier = {K,C}

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B,D,H,I,E,J,K}

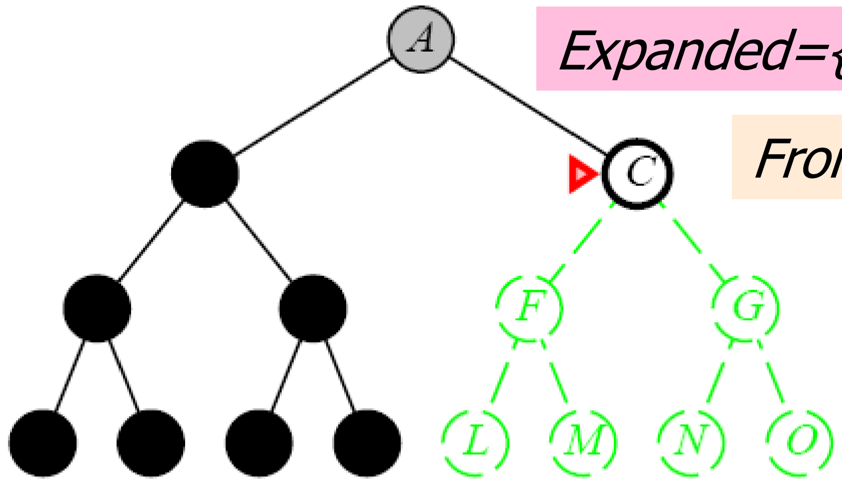
Frontier = {C}

Depth-first search

Expand deepest unexpanded node

Implementation:

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



Expanded={A,B,D,H,I,E,J,K,C}

Frontier = {F,G}

Depth-first search

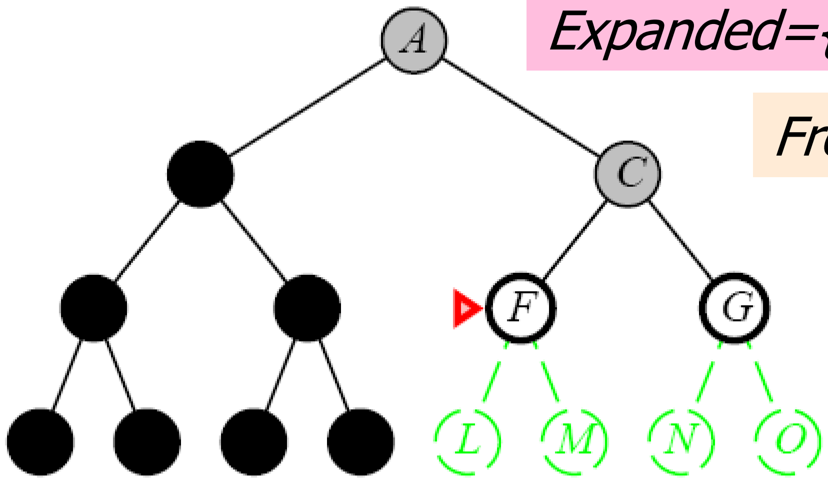
Expand deepest unexpanded node

Implementation:

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

Expanded={A,B,D,H,I,E,J,K,C,F}

Frontier = {L,M,G}



Depth-first search

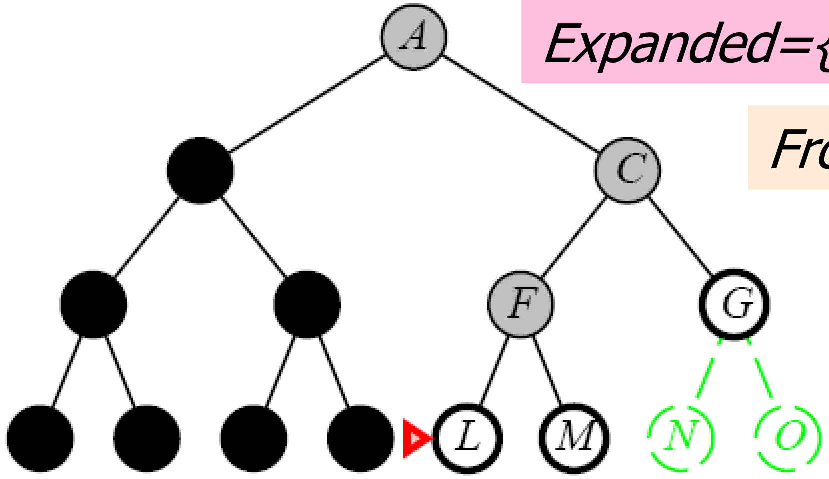
Expand deepest unexpanded node

Implementation:

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

Expanded={A,B,D,H,I,E,J,K,C,F,L}

Frontier = {M,G}



Depth-first search

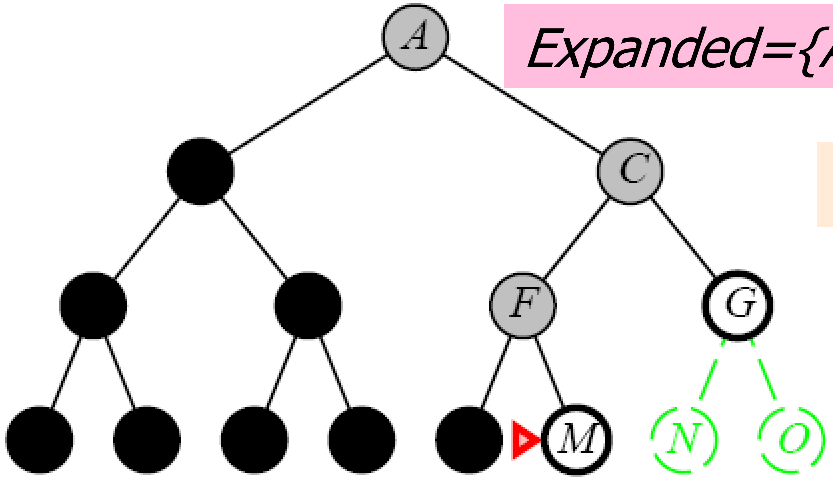
Expand deepest unexpanded node

Implementation:

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

Expanded={A,B,D,H,I,E,J,K,C,F,L,M}

Frontier = {G}



Properties of depth-first search

- Complete ?? No: fails in infinite-depth spaces, spaces with loops
  - modified to avoid repeated states along path $\Rightarrow$ complete in finite spaces

- Time ?? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first

- Space ?? $O(bm)$, i.e. linear space!

- Optimal ?? No

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
*C** - the cost of the optimal solution

Depth-limited search

- = depth-first search with depth limit /
- <u>Implementation</u>:
  - Nodes at depth / have no successors
  - Sometimes depth limits can be suggested by the knowledge of the problem.
  - <u>Example:</u> The Romania problem : there are 20 cities! If there is a solution, it must be of length 19!
    - But if we study the map carefully, we discover that any city can be reached from any other city in maximum 9 steps. This is the diameter of the state space – gives a good depth limit

Depth-Limited Search

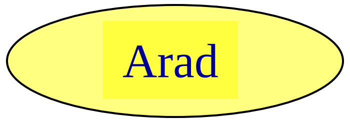
```
function Depth-Limited-Search(problem, limit)
                                                  returns a solution, or failure/cutoff
  return Recursive-DLS(MAKE-NODE(problem, INITIAL-STATE), problem, limit)

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
```

Iterative deepening search

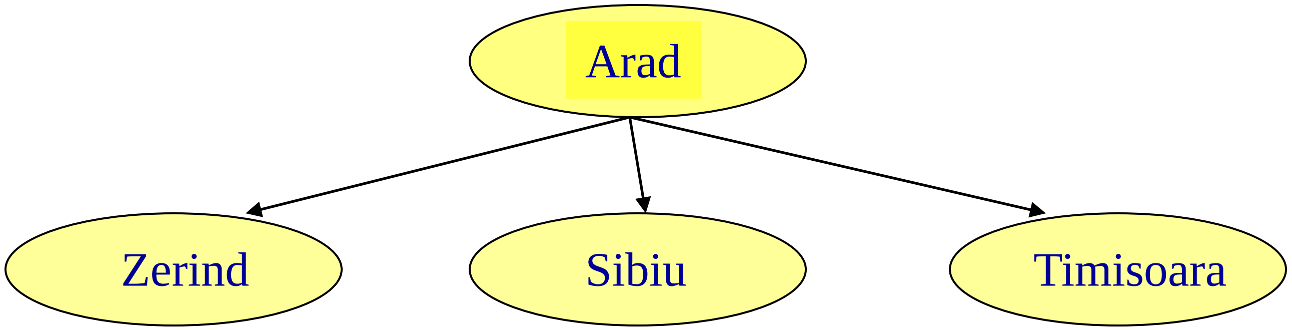
It is a general strategy used on combination with depth-first search, which finds the best depth limit!

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

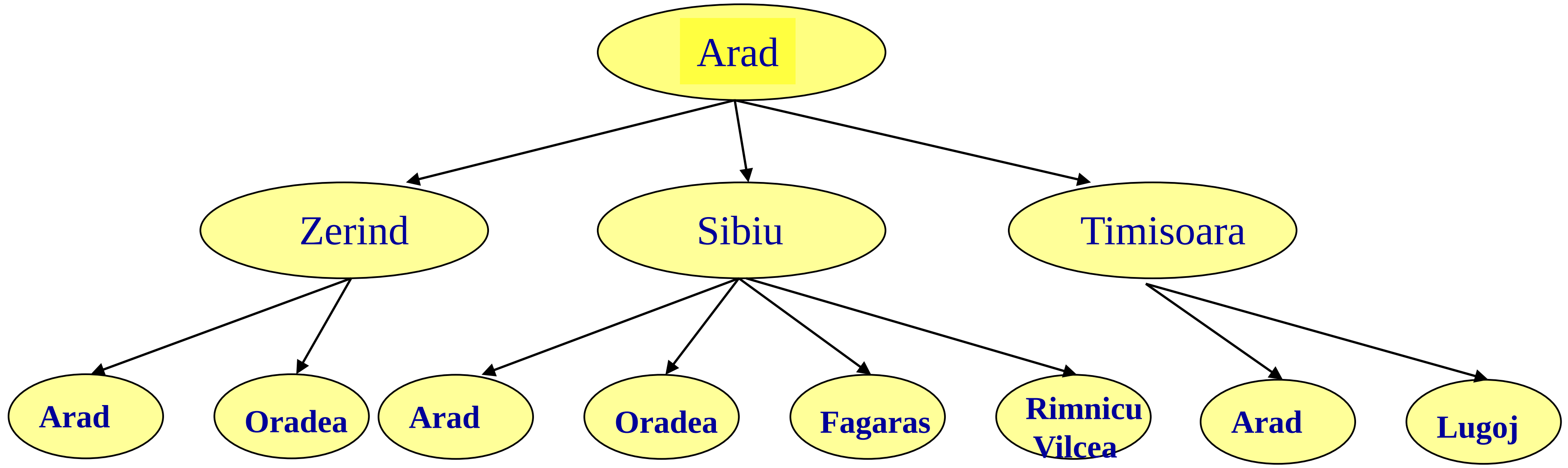
Iterative deepening search l = 0



Iterative deepening search $l = 1$

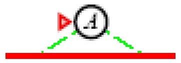


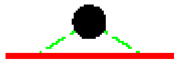
Iterative deepening search l = 2



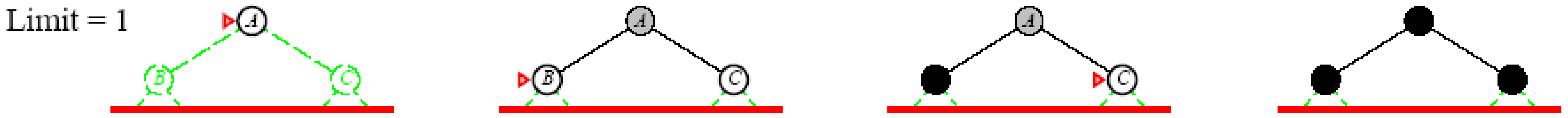
Iterative deepening search $l = 0$

Limit = 0

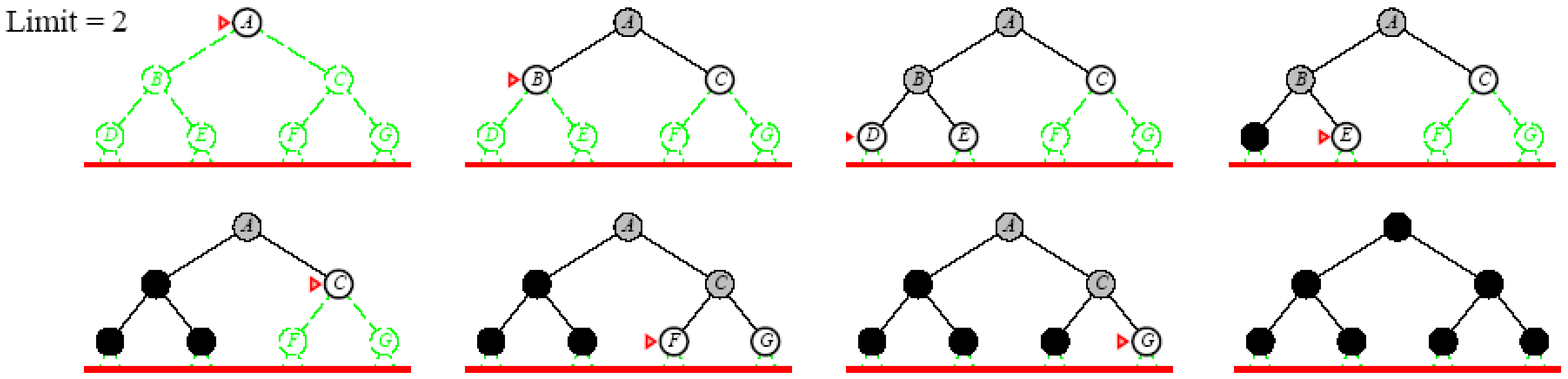




Iterative deepening search $l = 1$

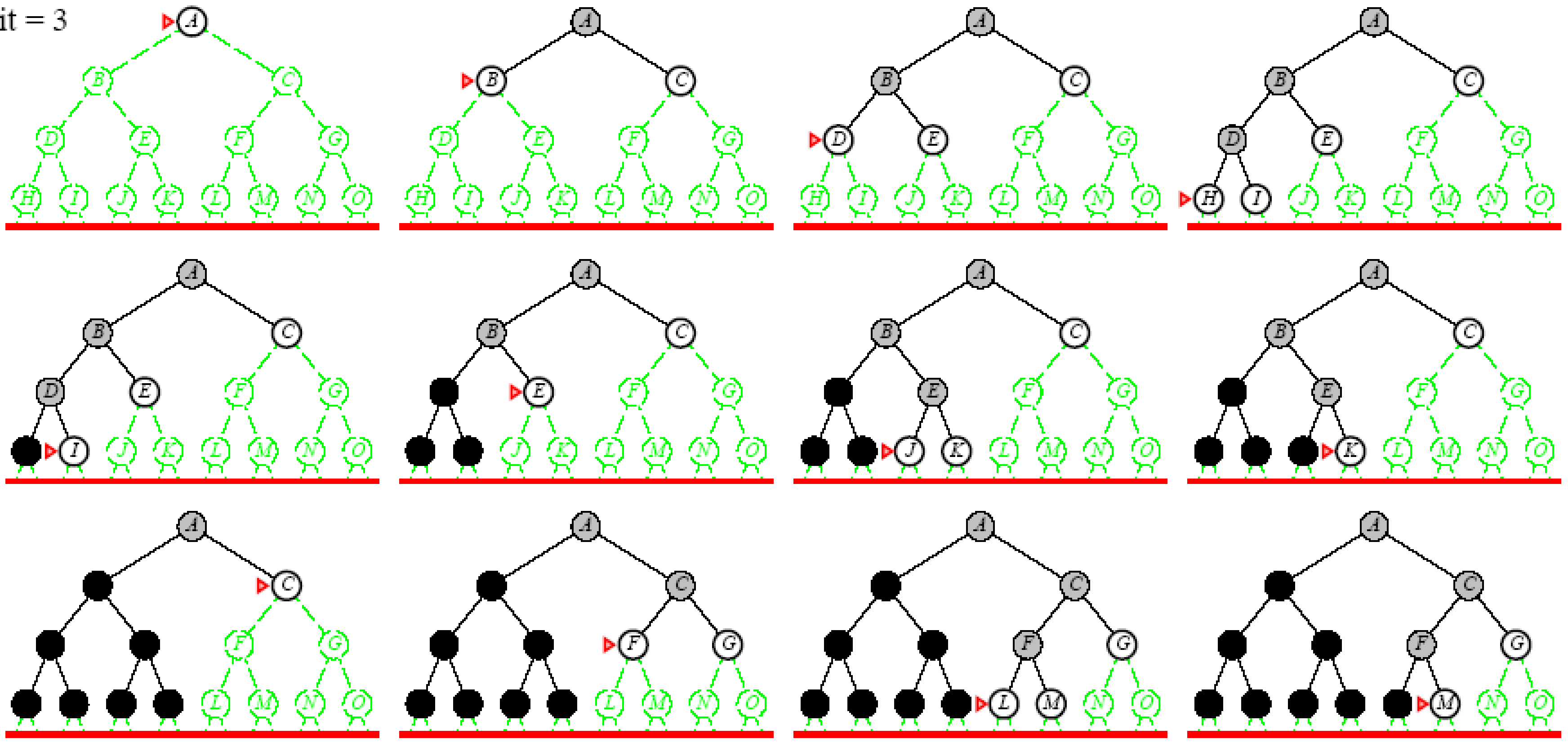


Iterative deepening search $l = 2$



Iterative deepening search $l = 3$

Limit = 3



Properties of iterative deepening search

- Complete ?? Yes
- Time ?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space ?? $O(bd)$
- Optimal ?? Yes, if step cost = 1
  - can be modified to explore uniform-cost tree

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
*C** - the cost of the optimal solution

Summary

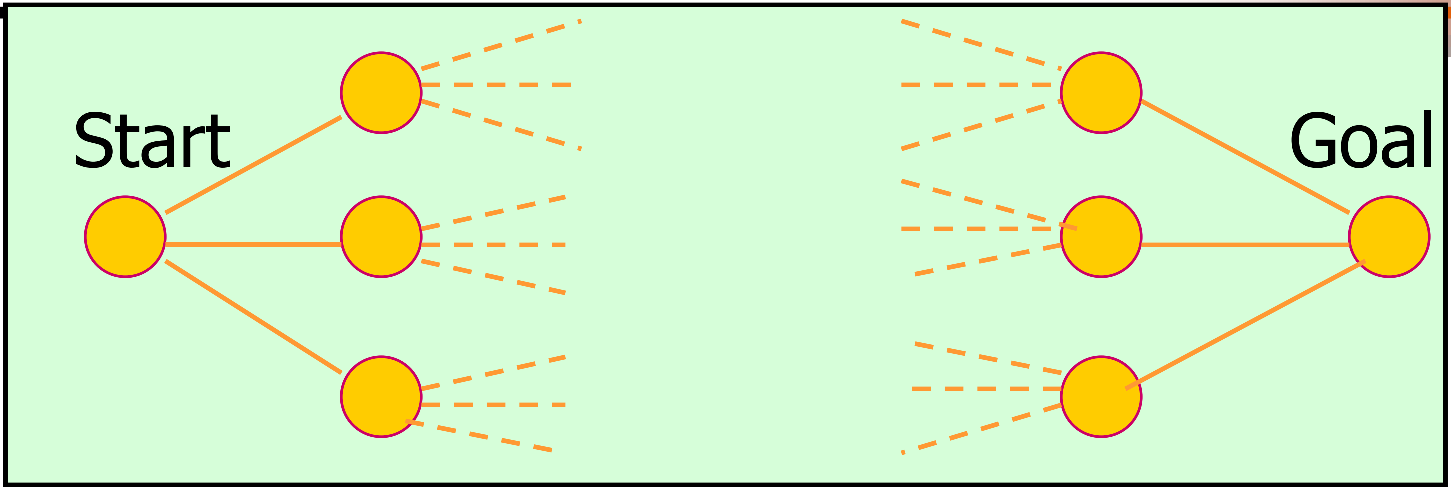
- Problem formulation usually requires abstracting real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Summary of algorithms

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

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
*C** - the cost of the optimal solution

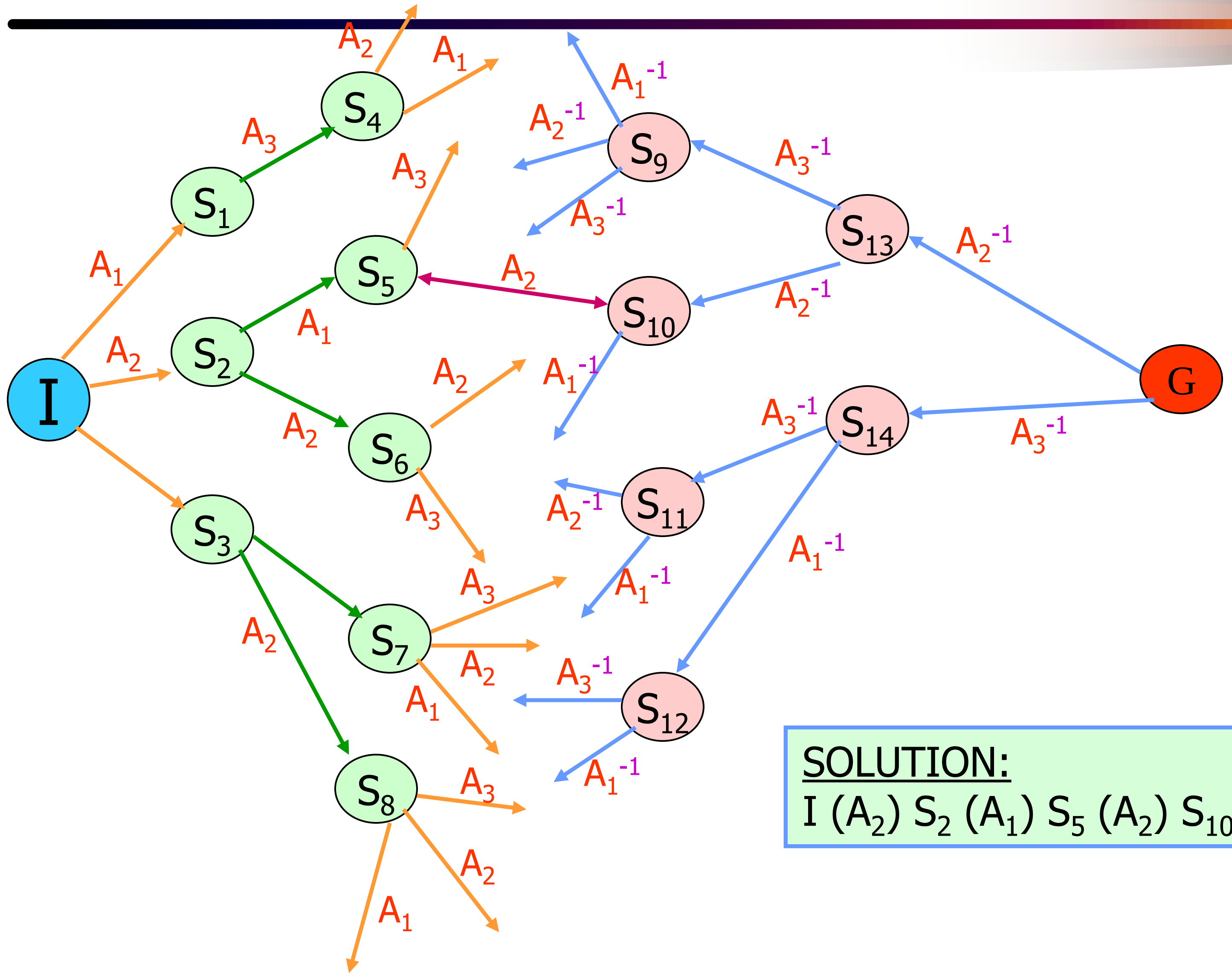
Bi-directional search



The idea is to run two simultaneous searches: one forward from the start(initial) node, one backwards from the goal node, hoping that the two searches meet in the middle!!!

- It is implemented by replacing the goal test with a check to see if the frontiers of the two searches intersect. If they do, a solution was found!!!!!

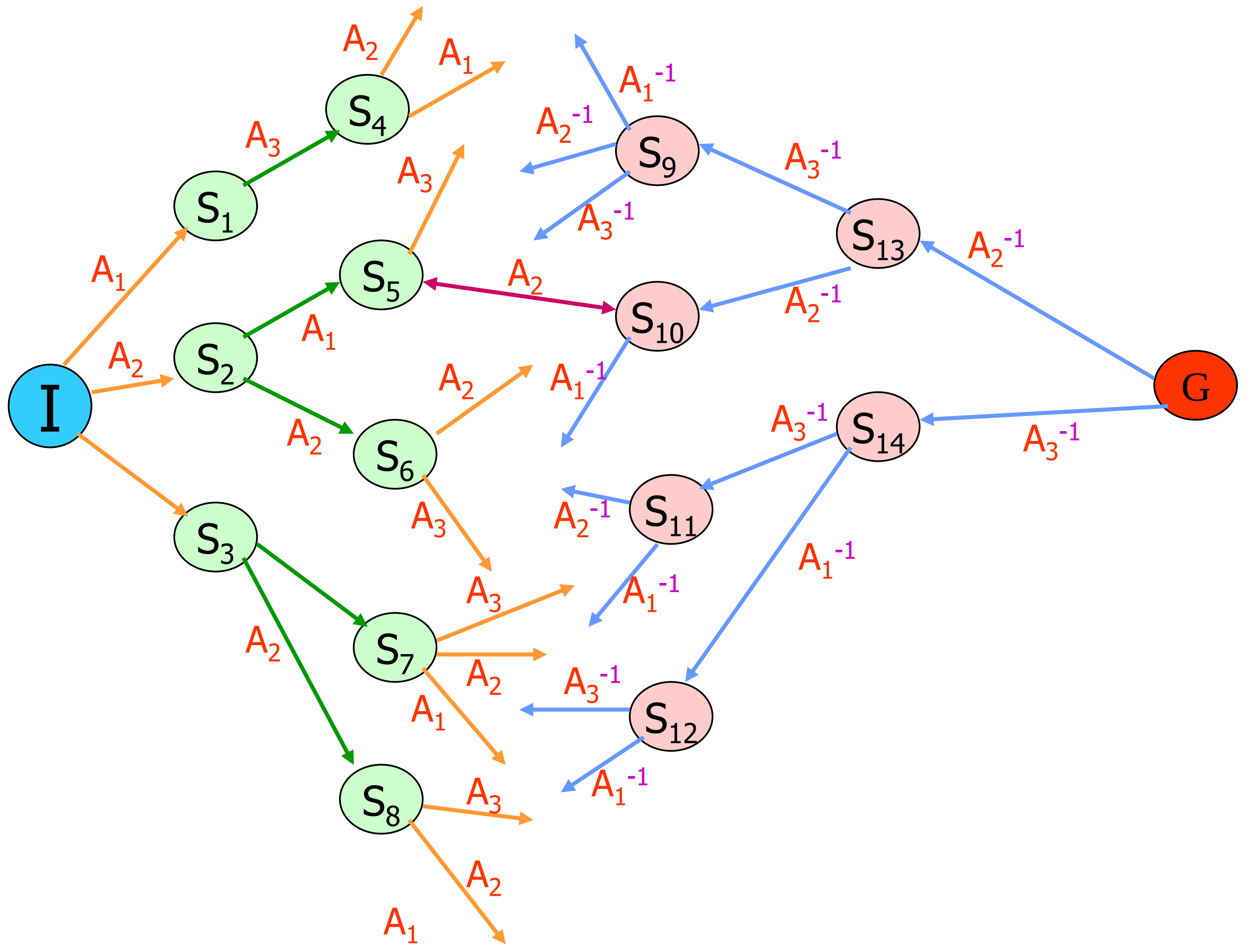
Example



<u>SOLUTION:</u>

I (A_2) S_2 (A_1) S_5 (A_2) S_{10} (A_2) S_{13} (A_2) G

The algorithm needs a mechanism that works in a constant amount of time, regardless of the number of nodes expanded, to determine if the two searches have achieved a common node. This is most often done by a hash table.



SEARCH 1
Expanded nodes:
1] I
2] S_1 S_2 S_3
3] S_4 S_5 S_6 S_7 S_8
4] ... S_{10} ....

SEARCH 2
Expanded nodes:
1] G
2] S_{13} S_{14}
3] S_9 S_{10} S_{11} S_{12}
4] ... S_5 ....

Bi-directional search

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
*C** - the cost of the optimal solution

Summary of algorithms

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

Since the depth is the exponent in the time complexity, it reduces the time requirement to $O(b^{d/2})$. For a BFS problem with b=10 and d=5, the number of expanded nodes is 111,111. For a bi-directional search, the number of expanded nodes is 2,222.