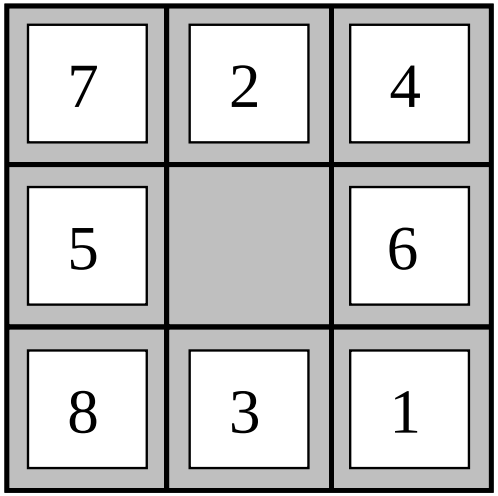
Problem solving and search

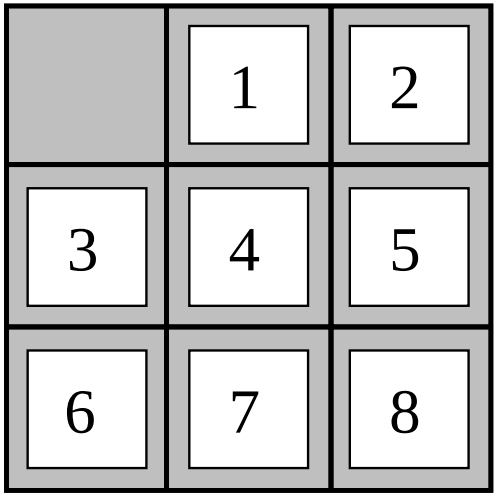
Chapter 3, Sections 3.1–3.4.5 - Part a: Introduction

Searching to Solve a Problem

♢ Consider some puzzles that you would want an AI to be able to solve:



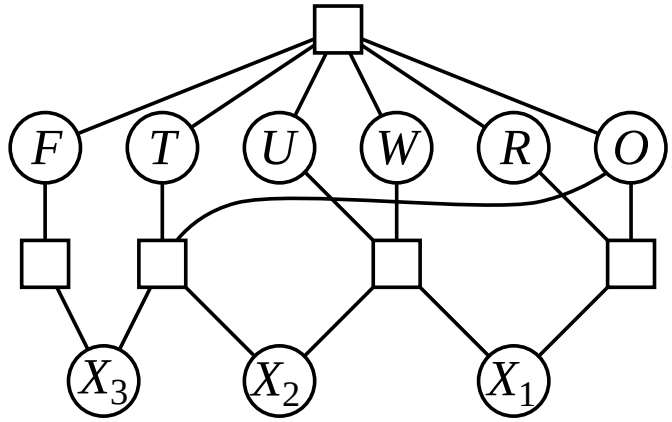
Start State



Goal State

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

(a)



(b)

Searching to Solve a Problem

♢ Simple-reflex agents directly maps percepts to actions. Therefore, they cannot operate well in environments where the mapping is too large to store or takes too much to specify; or where a model is needed.

♢ **Goal-based agents** can succeed by considering actions and desirability of their outcomes, in achieving their goal.

♢ **Problem solving agents** are goal-based agents that decide what to do by finding sequences of actions that lead to goal states

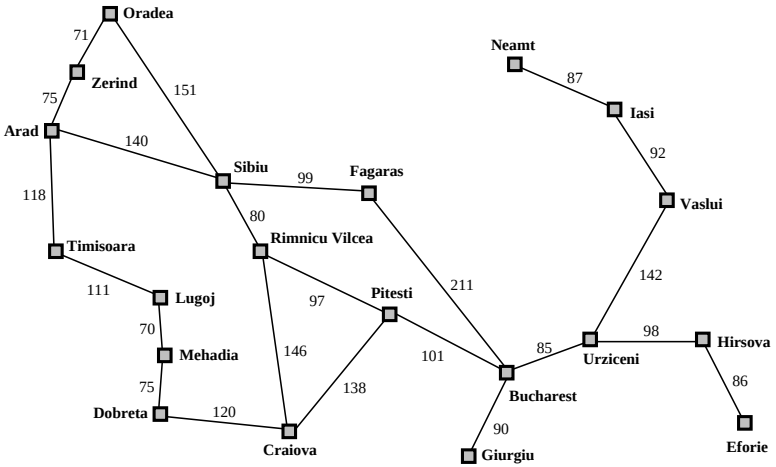
Introduction

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

♢ **Formulate goal**: Be in Bucharest

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

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



Introduction

♢ **Goal formulation**, based on the current situation and the agents performance measure, is the first step in problem solving and involves **finding the set of world states in which the goal is satisfied**.

Problem formulation is the process of deciding **what actions and states to consider, given a goal.**

If it were to consider actions at the level of move the left foot forward an inch or turn the steering wheel one degree left, the agent would probably never find its way out of the parking lot, let alone to Bucharest, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution.

Introduction

♢ Simple goal-based agents can solve problems via **searching the state space** for a solution, starting from the initial state and terminating when (one of) the goal state(s) are reached.

♢ The search algorithms can be **blind** (not using specific info about the problem) as in Chp. 3 or **informed** (Chp. 4) using heuristics about the problem for a more **efficient search**.

Example: Romania

♢ While you may see the solution on the map easily, the solution will not be obvious to an agent or us if the graph is huge...

♢ The input may also be given to us as a list of roads from each city. This is in fact how a robot will see the map, as a list of nodes and edges between them, possibly with associated distances:

Arad to: Zerind, Sibiu, Timisoara

Bucharest to: Pitesti, Guirgiu, Fagaras, Urziceni

Craiova to: Dobreta, Pitesti

Dobreta to: Craiova, Mehadia

Oradea to: Zerind, Sibiu

Zerind to: Oradea, Arad

Example: Traveling in Romania

Formulate goal:

be in Bucharest

Formulate problem: What are the actions and states?

The states of the robot, abstracted for this problem, are "the cities where the robot is/may be at".

The corresponding operators taking one state to the other are "driving between cities".

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

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.

(Abstract) solution =

set of real paths that are solutions in the real world

Single-state problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

operators (or *successor function* $S(x)$)
e.g., Arad $\rightarrow$ Zerind Arad $\rightarrow$ Sibiu etc.

goal test, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., $NoDirt(x)$

path cost (additive)
e.g., sum of distances, number of operators executed, etc.

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

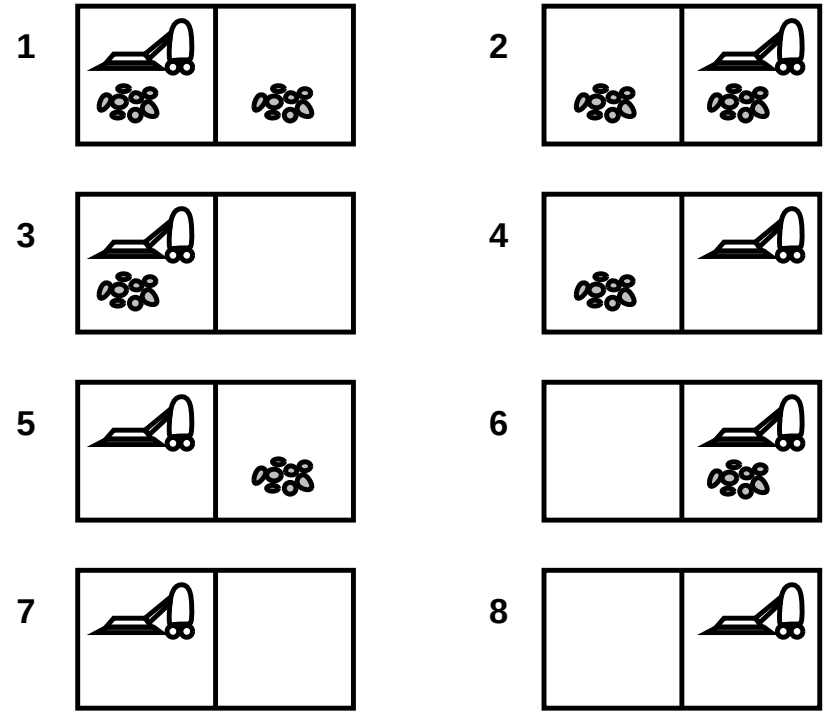
Example: vacuum world

Your robot needs to vacuum a two-room area. Each room may have dirt in it; the robot may be in one of the rooms and move left or right to go to the other room.

What are the states of this vacuum world?.

Example: vacuum world

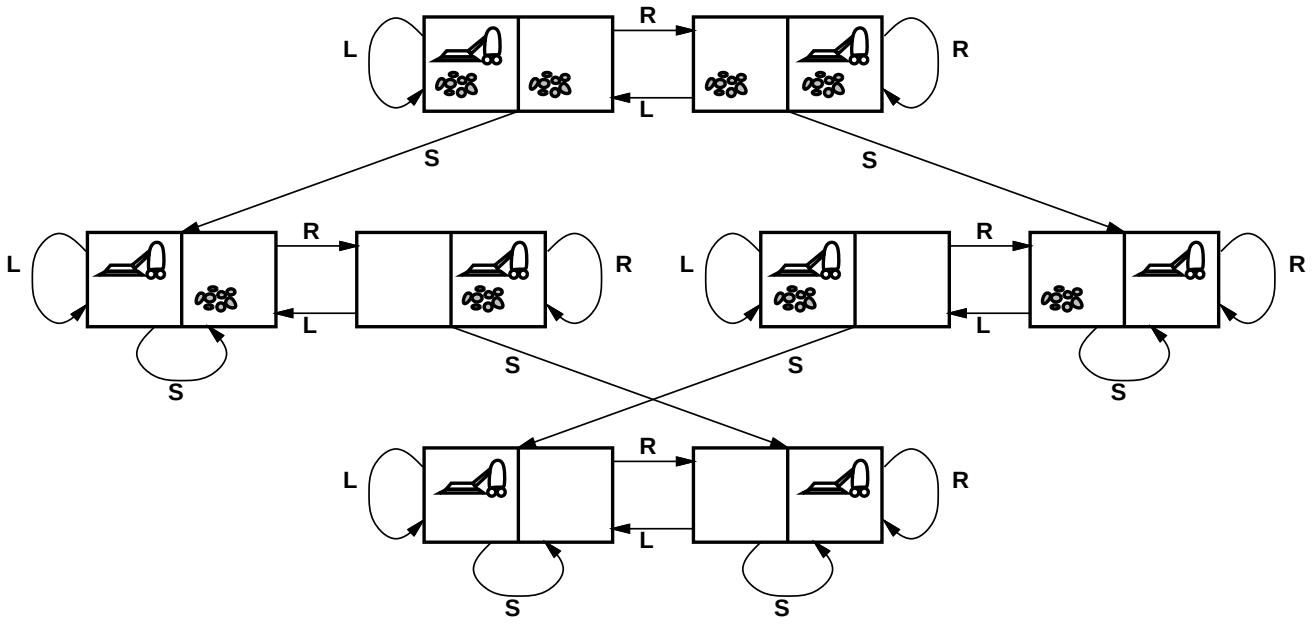
The 8 States:



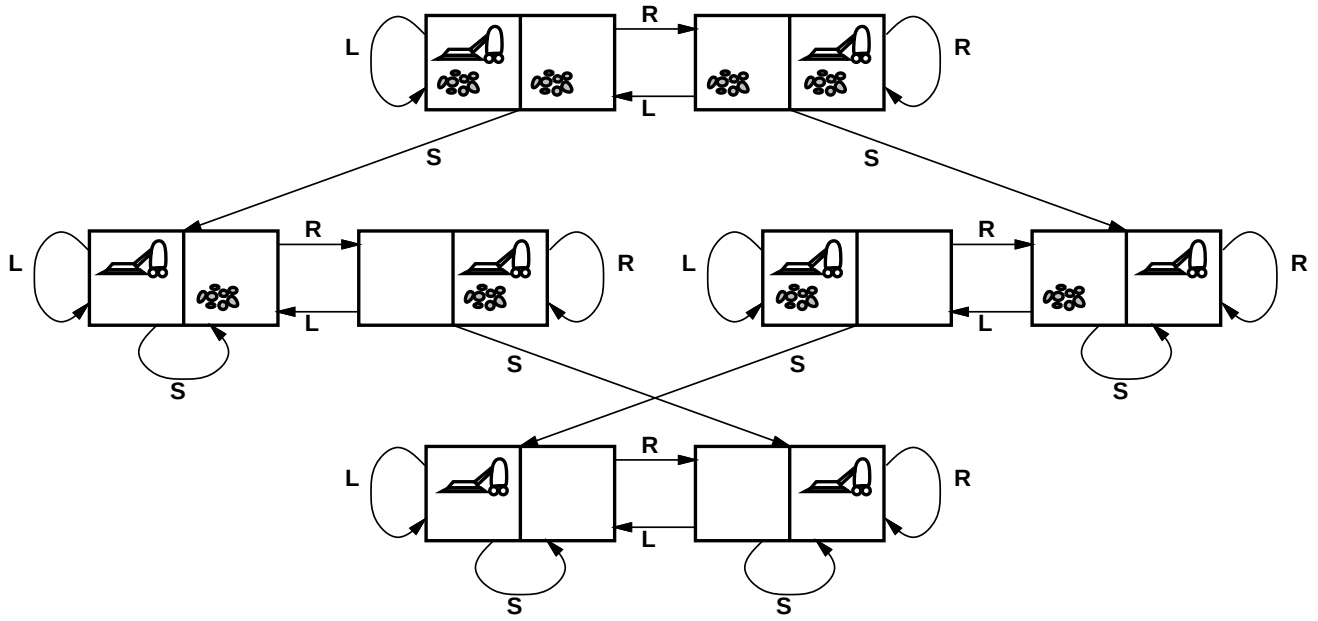
Example: vacuum world state space graph

In cases where some sensors (e.g. dirt sensor) are not working, we may ask if there is any solution (to clean the whole space).

The answer is yes, because even if we can't tell whether there is dirt or now, we can find a solution, as states transition from one to another in deterministic ways and after doing certain steps, we are sure to be in the goal state.

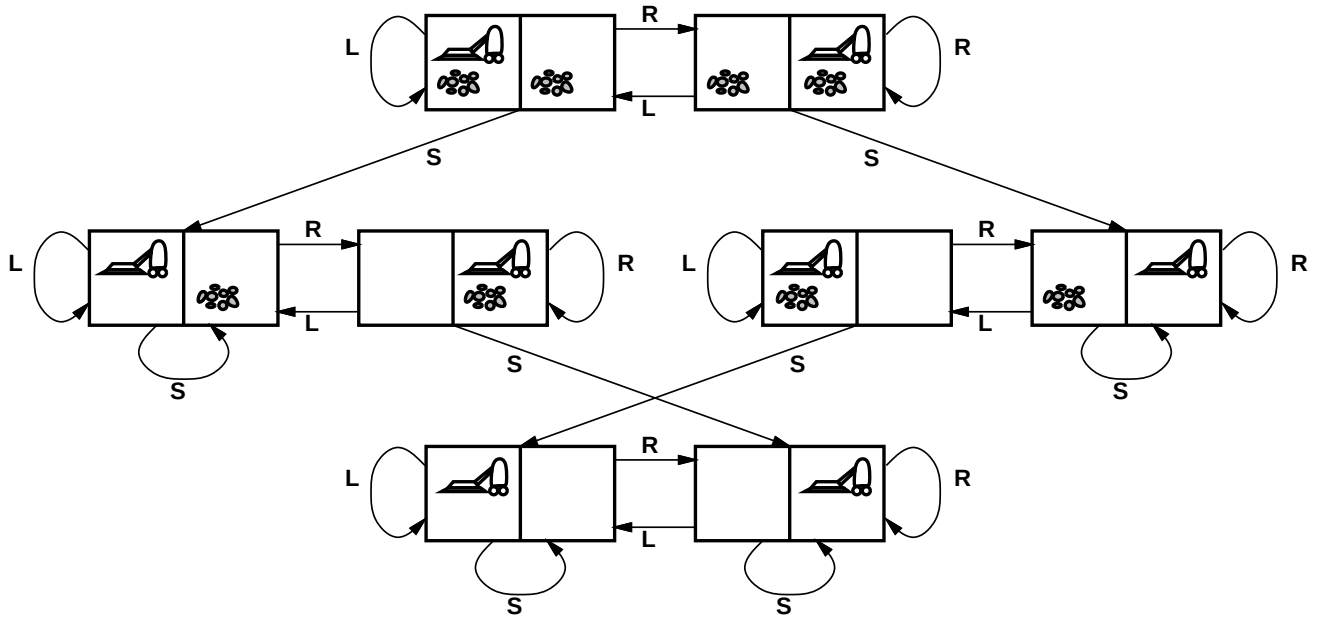


Example: vacuum world state space graph



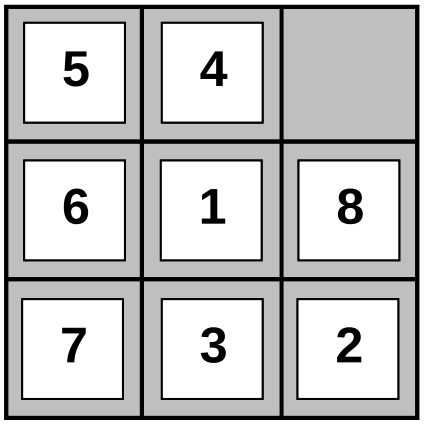
states??: operators??: goal test??: path cost??:

Example: vacuum world state space graph

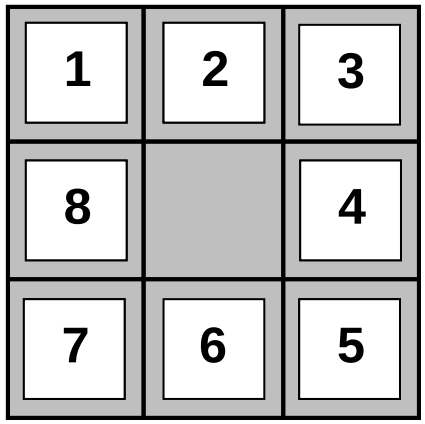


states??: integer dirt and robot locations (ignore dirt *amounts*)
operators??: $Left$, $Right$, $Suck$
goal test??: no dirt
path cost??: 1 per operator

Example: The 8-puzzle



Start State



Goal State

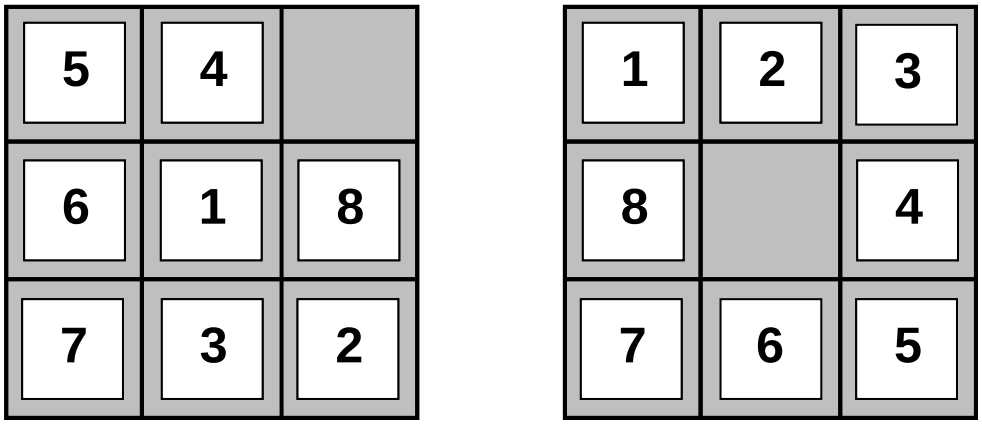
states??

operators??

goal test??

path cost??

Example: The 8-puzzle

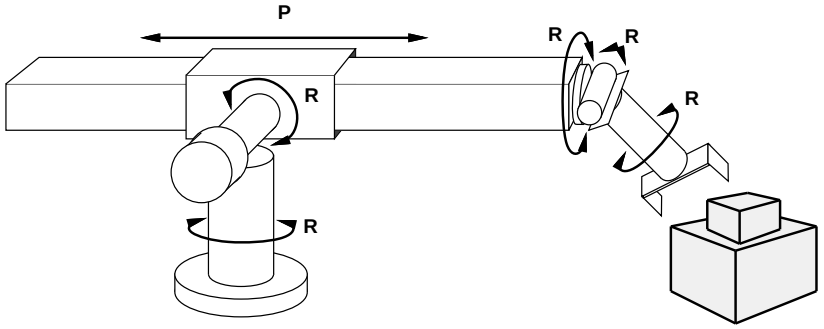


Start State **Goal State**

states??: integer locations of tiles (ignore intermediate positions)
operators??: 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]

Example: robotic assembly



states??: real-valued coordinates of
robot joint angles
parts of the object to be assembled

operators??: continuous motions of robot joints

goal test??: complete assembly

path cost??: time to execute

Searching to Solve a Problem

Now that we understand how search can be used for problem solving, let's look at how we can get an agent to solve problems

$\Rightarrow$ implementation of search algorithms

Implementation of search algorithms

♢ **Offline, simulated exploration** of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

Note that when thinking about the goal of going to Bucharest, we are not actually taking all those trips, but simulating it in our minds (offline search).

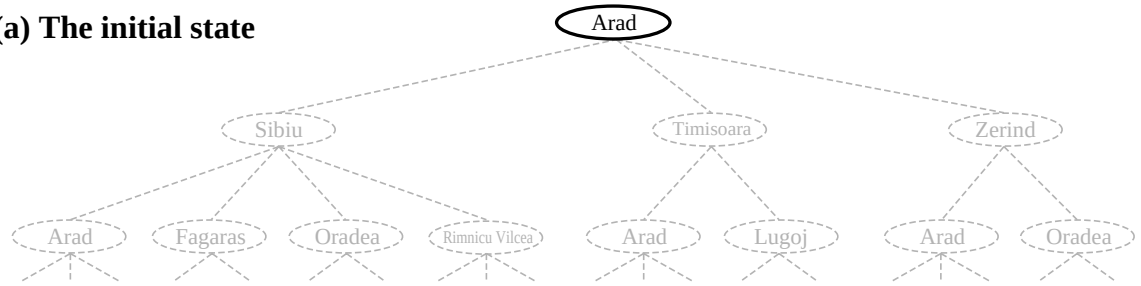
♢ Need to **keep track of the partial work**: use a **search tree**

We need a systematic search and keep track of what we have search so far.

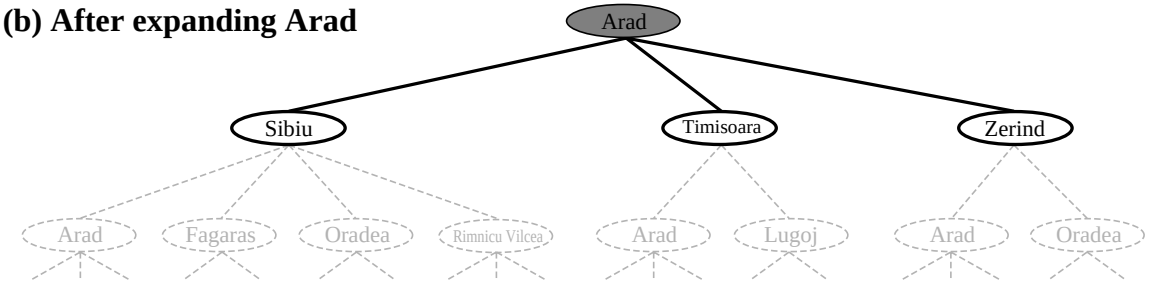
Tree search

We will see how we can use a search tree to keep track of our partial search process.

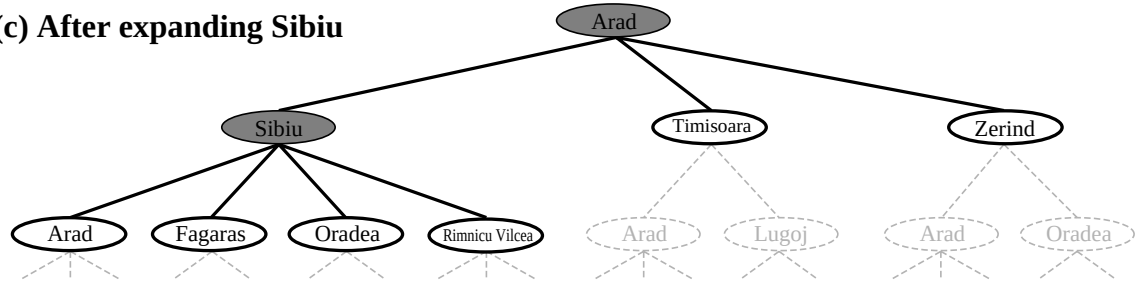
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu

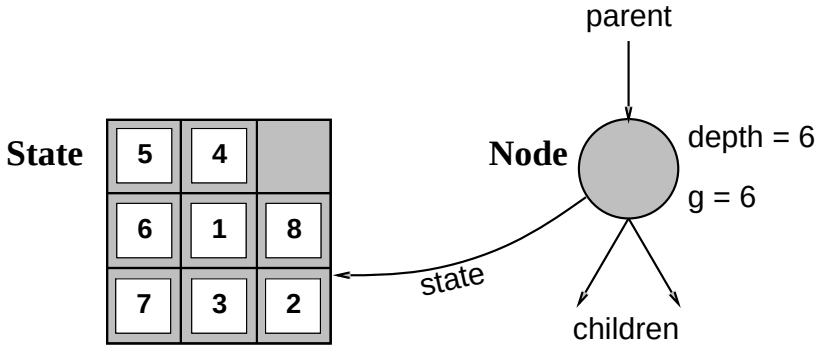


Tree search Algorithm

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Tree search Implementation: states vs. nodes

A **state** is a (representation of) a physical configuration
A **node** is a data structure constituting part of a search tree
includes *state*, *parent*, *children*, *operator*, *depth*, *path cost* $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

Note that there are pointers (links) from Parent to Child, as well as from Child to parent, so as to be able to recover a solution once the goal is found.

Terminology

♢ **depth of a node**: number of steps from root (starting from depth=0)

♢ **path cost**: cost of the path from the root to the node

♢ **expanded node**: node pulled out from the queue, goal tested (not goal) and its children (resulting from alternative actions) are added to the queue. It 'is often used synonymous to **visited** node.

Only difference is that we could visit a node and see that it corresponds to the goal state and not expand it. So visited can be one more than expanded (you can use them the interchangeably, unless asked otherwise).

♢ **generated nodes**: nodes added to the fringe. different than nodes expanded!

Terminology

♢ **Leaf node**: A node with no children in the tree

♢ **Fringe**: The set of all leaf nodes available for expansion at any given time. Fringe is also called **Frontier**.

♢ Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next, determned by their **search strategy**.

Implementation of search algorithms

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

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

Data structures for the implementation

♢ We need to be able to implement the **fringe** and **explored / visited** lists efficiently.

♢ Fringe should be implemented as a **queue**.
Queue operations:

- Empty?(Fringe) should return True only if there are no elements in the Fringe.
- RemoveFront(Fringe) removes and returns the "first" element of the Fringe
- InsertElement(Fringe,Element) inserts Element and returns the Fringe

Notice that we will always take a node from the front of the Fringe, so insertion of the expanded nodes (depending on the Queueing Function) is what distinguishes between different search strategies.

Implementation of search algorithms

♢ Insertion at the End or the Front would take constant time and result in FIFO (first-in first-out) queue or LIFO queue (last-in first-out, which makes it a stack).

♢ Certain search algorithms (e.g. Uniform-Cost search) requires the insertion to be done not to the front or end, but according to some priority (e.g. total cost etc). In that case, we need to use a **priority queue**.

♢ The Explored set can be implemented as a **hash table** and roughly have constant insertion and lookup time regardless of the number of states.

Implementation of search algorithms

SUMMARY in case you have not taken a data structures course:

♢ You should assume the search tree implementation is done with appropriate data structures as efficiently as possible.

♢ **As the programmer choosing the appropriate search strategy, you are responsible in selecting the appropriate search strategy, so as to reduce the time and/or space complexities for the particular problem at hand**.

→ reduce the number of explored and generated nodes which affects both time and space complexity

Summary

♢ Know the names and rules of the problems mentioned in these slides

♢ Understand searching, search tree, search strategy concepts

♢ Know terminology: goal test, visited, expanded, generated nodes; branching factor, depth of the solution, maximum depth of the tree; path cost, operator; online and offline search and tree terminology (root and leaf nodes, child and parent links, backtracking...)

♢ Know the basic TreeSearch algorithm (slide 23)