Solving Problems by Searching

CS 3600
Intro to Artificial Intelligence

Simplifying the domain

Our first intelligent agent will be restricted to work on environments that are

- **Known**
- **Fully observable**
- **Single-agent**
- **Deterministic**
- **Episodic***
- **Static**
- **Discrete**

Extensions that relax these assumptions

- **Unknown** (Learning agents)
- **Partially observable** (POMDPs)*
- **Multi-agent** (min-max search)
- **Stochastic** (probabilistic reasoning)
- **Sequential** (Hierarchical Planning)*
- **Dynamic** (Replanning)
- **Continuous** (RRTs and Controls)*

Search-based agent

```
def __init__(self,init_state):
    self.state = init_state
    self.problem = None
    self.plan = list()
    self.goal = None
def search_based_agent(self,percept):
    self.state = self.update_state(percept)
    if len(self.plan)<=0:
        self.goal = self.make_goal(self.state)
        self.problem = self.make_problem(self.state,self.goal)
        self.plan = self.search(self.problem)
    action = self.plan.pop(0)
    return action
```

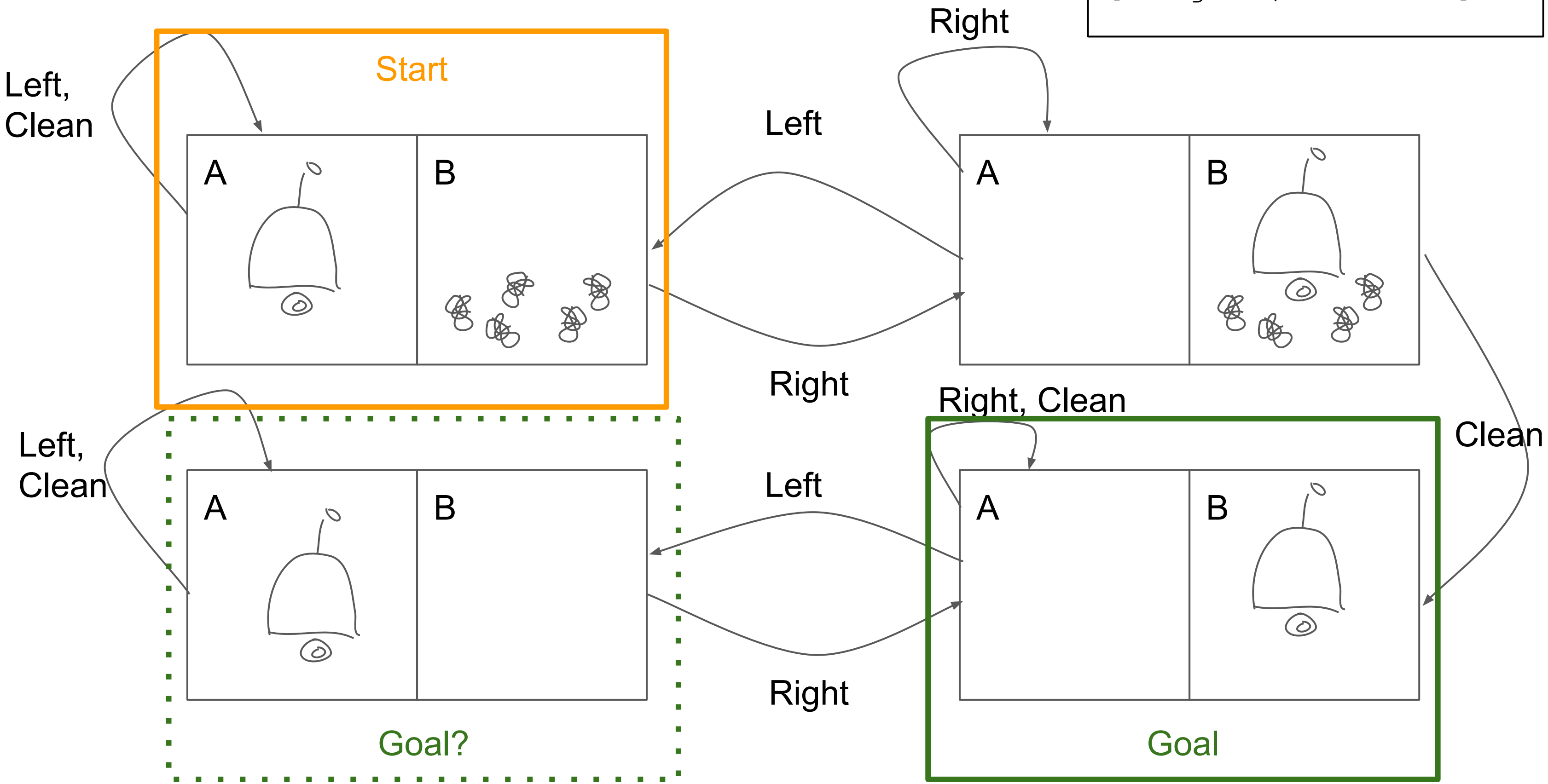
This agent is **offline**: it decides on a full plan of action before taking a single step

Components of a Search-based agent

- **`update_state(percept)`** - Construct a state representation from a given percept.
  `{‘loc’:‘A’, status:‘clean’} →`
  `{‘loc’:‘A’, ’A-clean’:True, ‘B-clean’:self.state[‘B-clean’]}`
- **`choose_goal(state)`** - Define success (goal state? goal test?)
  `make_problem(state,goal)` - set up actions, construct state space (explicit? successor function?), initialize book-keeping
- **`search(problem)`** - returns a sequence of actions that take the agent from the start state to the/a goal state

Example problem: Vacuum World

Solution:
`['Right', 'Clean']`



Components of a problem

For non-trivial problems, we need a way to **generate** the state space without explicitly representing every node/edge

- **Start state**: The initial state, where the agent starts
- **Successor function**: S(state) returns a set of (action, successor state) pairs
- **Goal test function**: Goal(state) returns true if the state is a goal
- **Step cost**: c(s1, a, s2) returns the cost of moving from s1 to s2 using action a

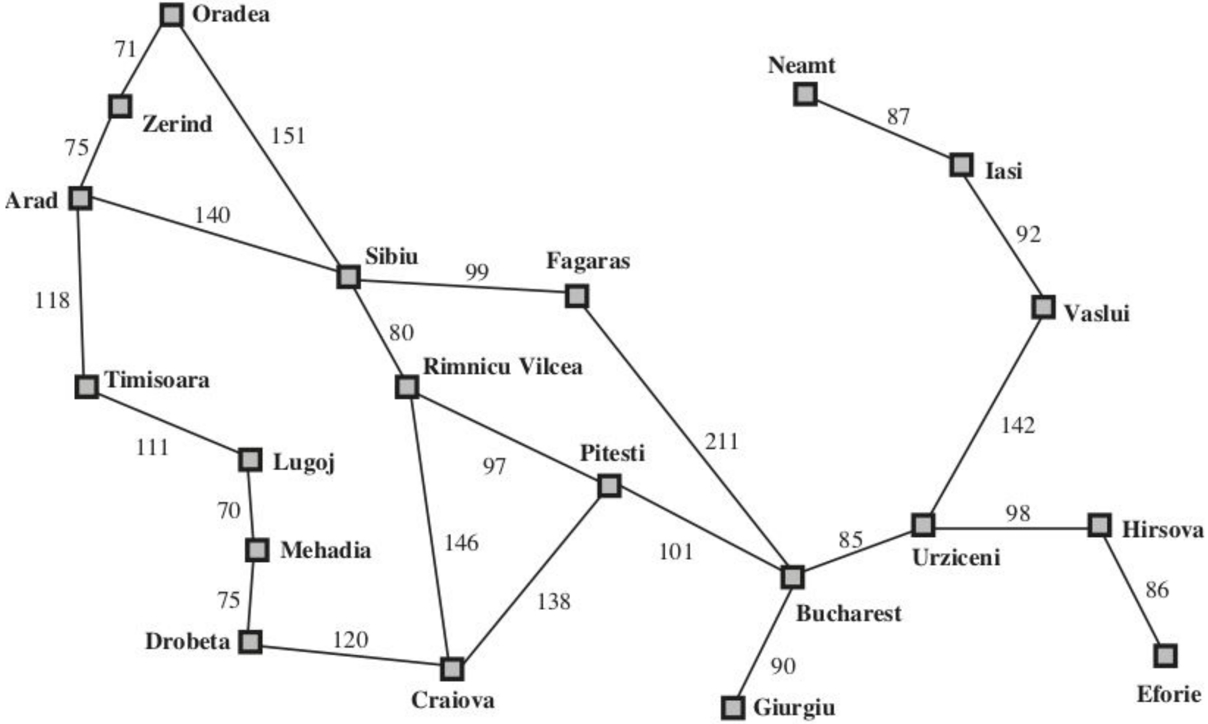
Components of a problem example: sudoku

- **Start state**:
  Partially filled board
- **Successor function**:
  S(state): states generated by filling in one blank space with a non-conflicting number 1-9
- **Goal test function**:
  Goal(state): Is the entire board filled with non-conflicting numbers?
- **Step cost**:
  c(s1, a, s2): 1

					1		5	
7	2		3	5			9	1
			8		7			
		8			5			4
	4	1		3		6	8	
5			4			7		
			6		3			
4	8			7	9		1	6
	9		1					

Components of a problem example: Romania

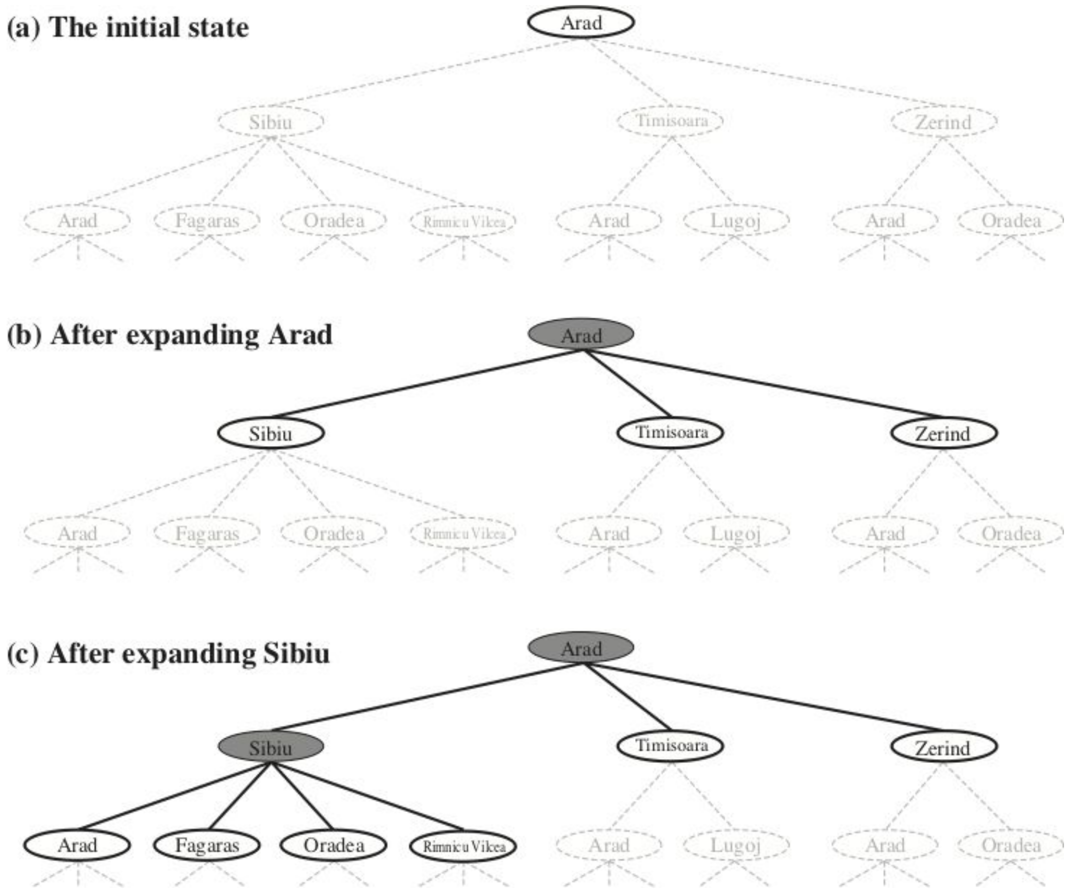
- **Start state**:
  'Arad'
- **Successor function**:
  S(state): Neighboring cities of state
- **Goal test function**:
  Goal(state): state=='Bucharest'
- **Step cost**:
  c(s1, a, s2): distance from s1 to s2 via 'a'



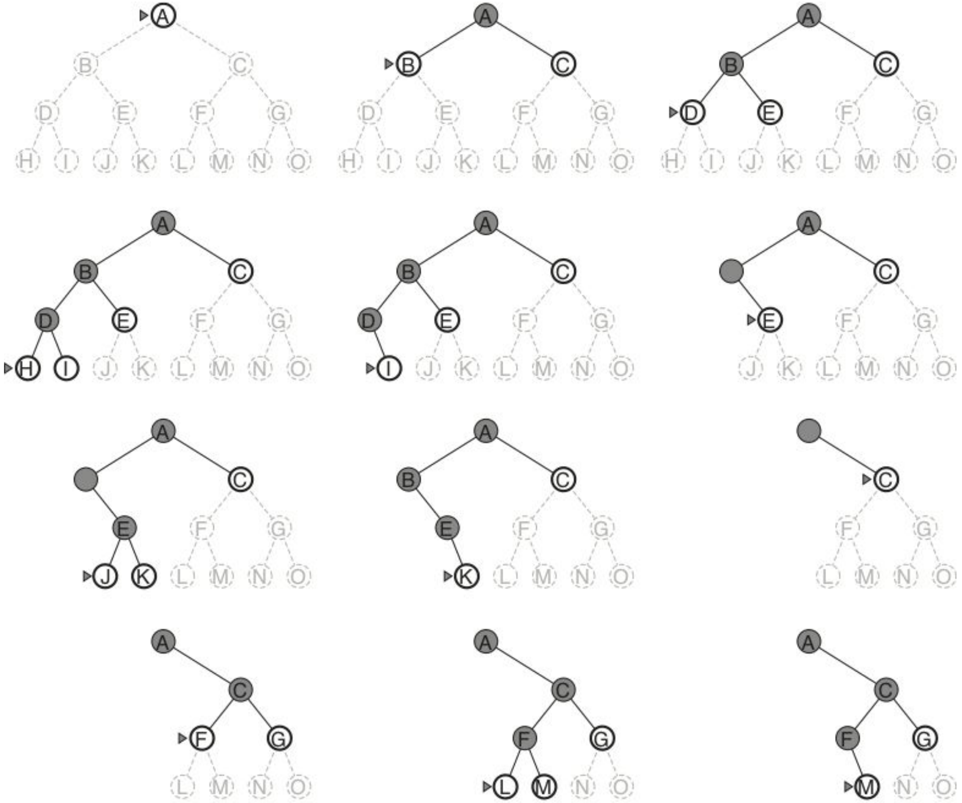
Representing search space - tree version

- For some problems, the number of states is too large (infinite?) to construct an explicit graph
- We can build the pieces of the state space we need to search 'as we go'
- The **search tree** is rooted at the initial state, leaves are expanded into their successors, may contain duplicate states (but not **nodes**!)
- Implementation note: children have 'back-pointers' to parents

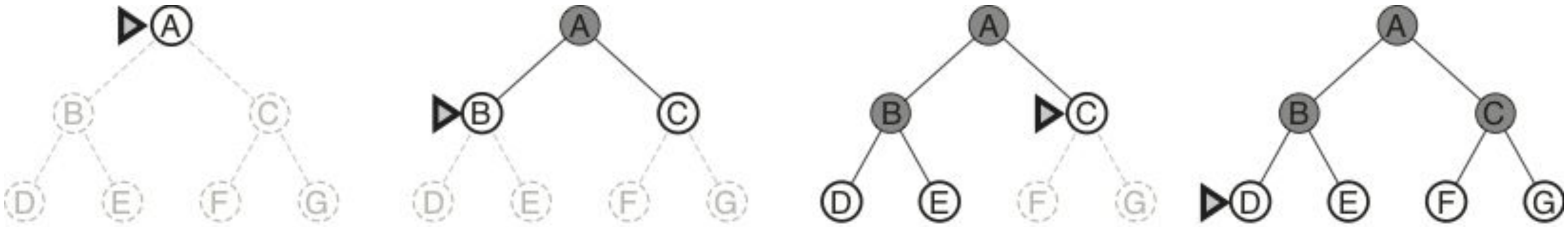
We know how to search trees!



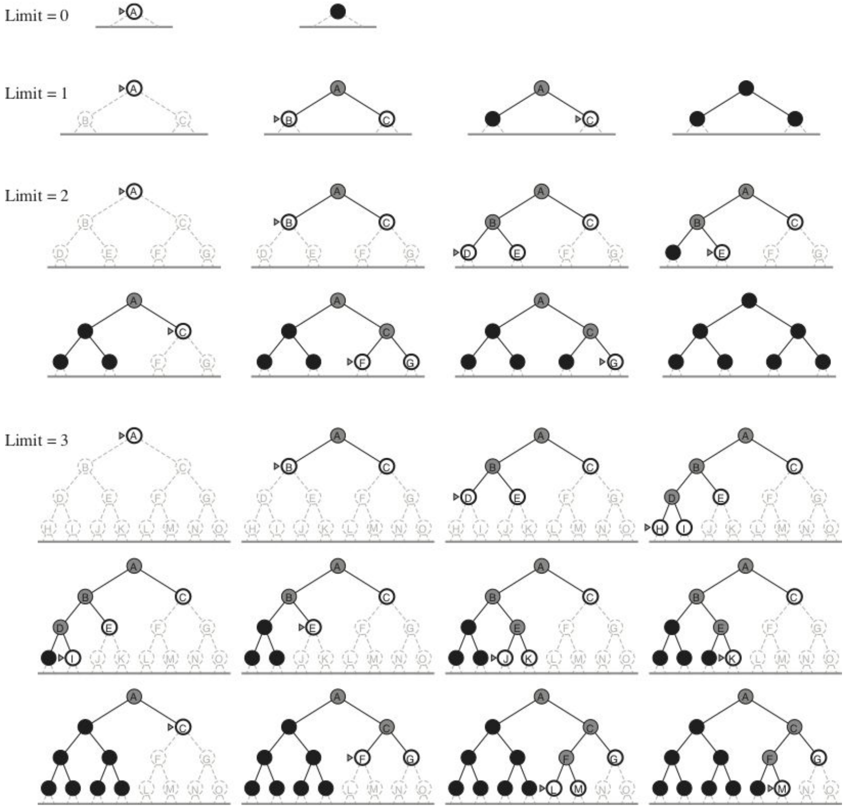
Depth First Search



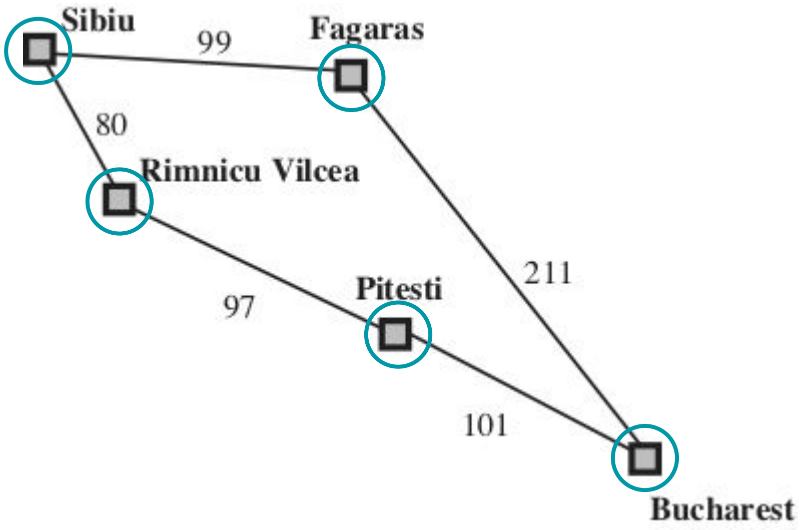
Breadth First Search

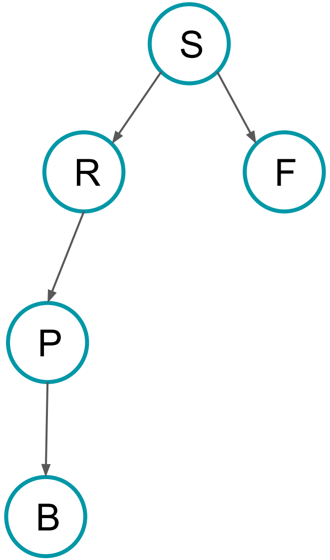


Iterative Deepening Search



Uniform Cost Search (Dijkstra's)





Comparing search algorithms

	BFS	UCS	DFS	IDS
Time	$O(b^d)$	$O(b^{d+1})$, $O(b^{1+C^*/\epsilon})$	$O(b^m)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{d+1})$, $O(b^{1+C^*/\epsilon})$	$O(b*m)$	$O(b*d)$

b: branching factor, **d**: depth of shallowest solution

m: maximum depth of the tree, ϵ: smallest step cost, **C***: cost of optimal solution

Complete: BFS & IDS (if $b<\infty$), DFS(if $m<\infty$), UCS (if $\epsilon>0$, and $b<\infty$)

Optimal: BFS & IDS (if all steps cost ϵ), UCS

Preview: Generic Search Algorithm

DFS, BFS, and UCS can be implemented with **a single algorithm**! Choice of data structure for the “next child to expand” determines which one.

- BFS: queue (children are expanded in the order they are added)
- DFS: stack (children are expanded in last-in-first-out order)
- UCS: priority queue (children are expanded based on cost-from-start)

IDS requires a small tweak: a depth limit parameter

Summary and preview

Wrapping up

- Search based agents work offline to find a sequence of actions that gets them from the initial state to a goal state
- A **search problem** can be represented explicitly as a graph, or implicitly by a start state, a successor function, a goal test function, and a cost function
- With this formulation, we can use any number of well known search algorithms to solve search problems

Preview

- Generic Search Algorithm, Worked Examples