Search: overview

Course plan

$\boxed{\text{Search problems}}$

Markov decision processes

Adversarial games

Constraint satisfaction problems

Markov networks

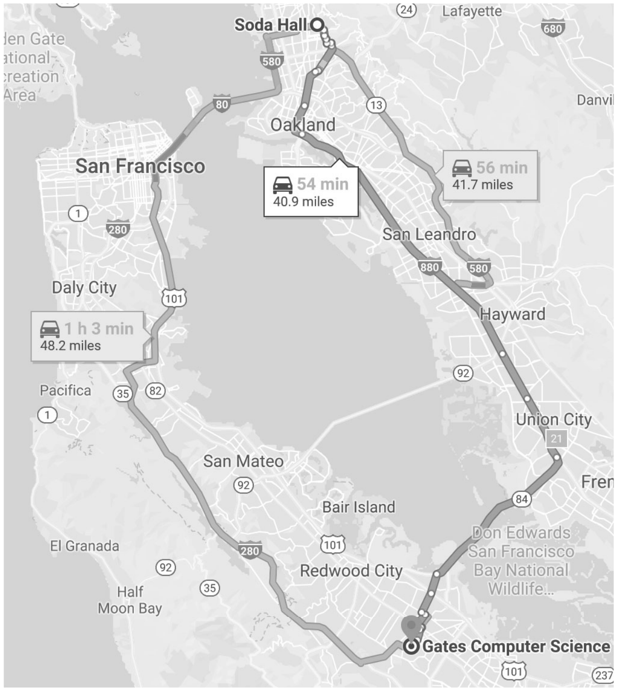
Bayesian networks

Reflex **States** **Variables** **Logic**

Low-level High-level

Machine learning

Application: route finding



Objective: shortest? fastest? most scenic?

Actions: go straight, turn left, turn right

Application: robot motion planning

Objective: fastest path

Actions: acceleration and throttle

Application: robot motion planning

Objective: fastest? most energy efficient? safest? most expressive?

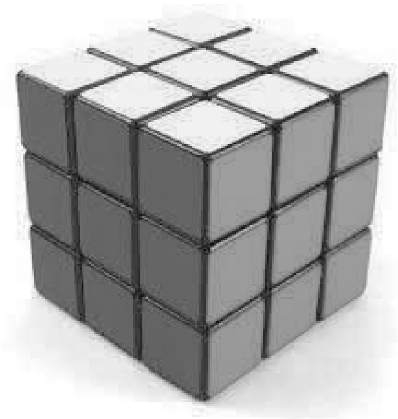
Actions: translate and rotate joints

Application: multi-robot systems

Objective: fastest? most energy efficient?

Actions: acceleration and steering of all robots

Application: solving puzzles



Objective: reach a certain configuration

Actions: move pieces (e.g., `Move12Down`)

Application: machine translation

la maison bleue

$\downarrow$

the blue house

Objective: use fluent English and preserve meaning

Actions: append single words (e.g., `the`)

Beyond reflex

Classifier (reflex-based models):

$x \longrightarrow \boxed{f} \longrightarrow$ single action $y \in \{-1, +1\}$

Search problem (state-based models):

$x \longrightarrow \boxed{f} \longrightarrow$ **action sequence** $(a_1, a_2, a_3, a_4, \dots)$

Key: need to consider future consequences of an action!

Paradigm

Modeling

Inference

Learning

Roadmap

Modeling

Modeling Search Problems

Algorithms

Tree Search

Dynamic Programming

Uniform Cost Search

Programming and Correctness of UCS

A*

A* Relaxations

Learning

Structured Perceptron

Search: modeling

Question

A **farmer** wants to get his **cabbage**, **goat**, and **wolf** across a river. He has a boat that only holds two. He cannot leave the cabbage and goat alone or the goat and wolf alone. How many river crossings does he need?

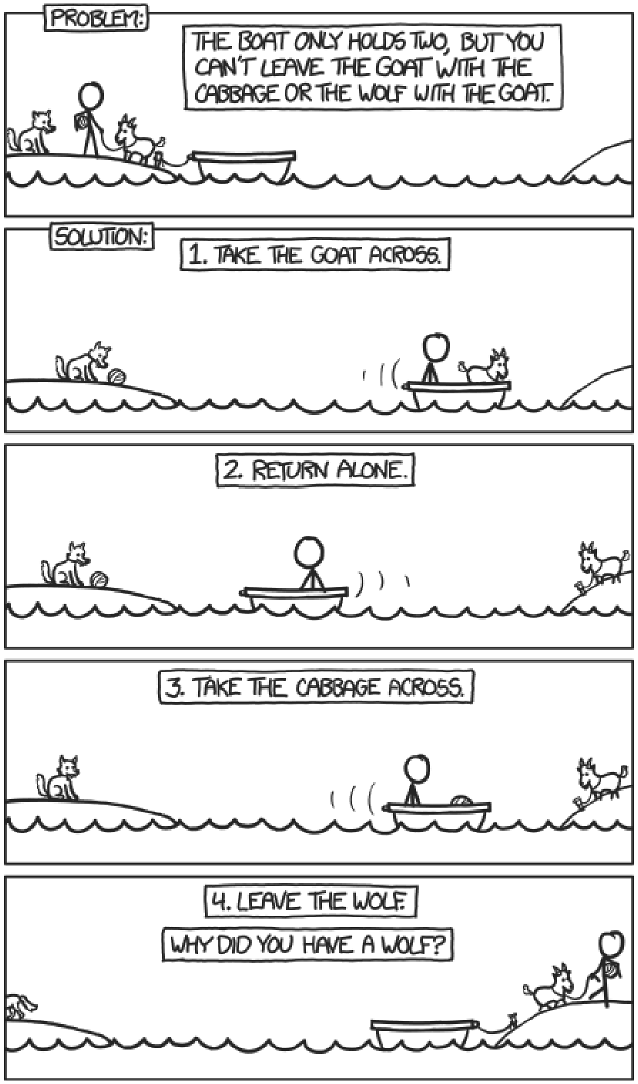
- 4

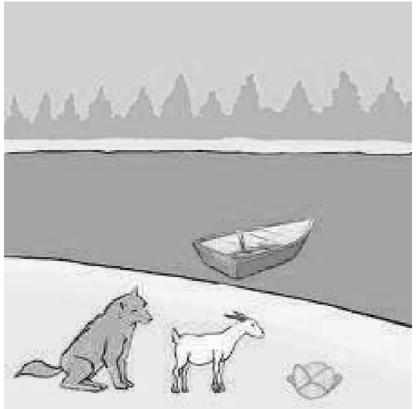
- 5

- 6

- 7

- no solution


PROBLEM:
THE BOAT ONLY HOLDS TWO, BUT YOU CAN'T LEAVE THE GOAT WITH THE CABBAGE OR THE WOLF WITH THE GOAT.
SOLUTION:
1. TAKE THE GOAT ACROSS.
2. RETURN ALONE.
3. TAKE THE CABBAGE ACROSS.
4. LEAVE THE WOLF.
WHY DID YOU HAVE A WOLF?

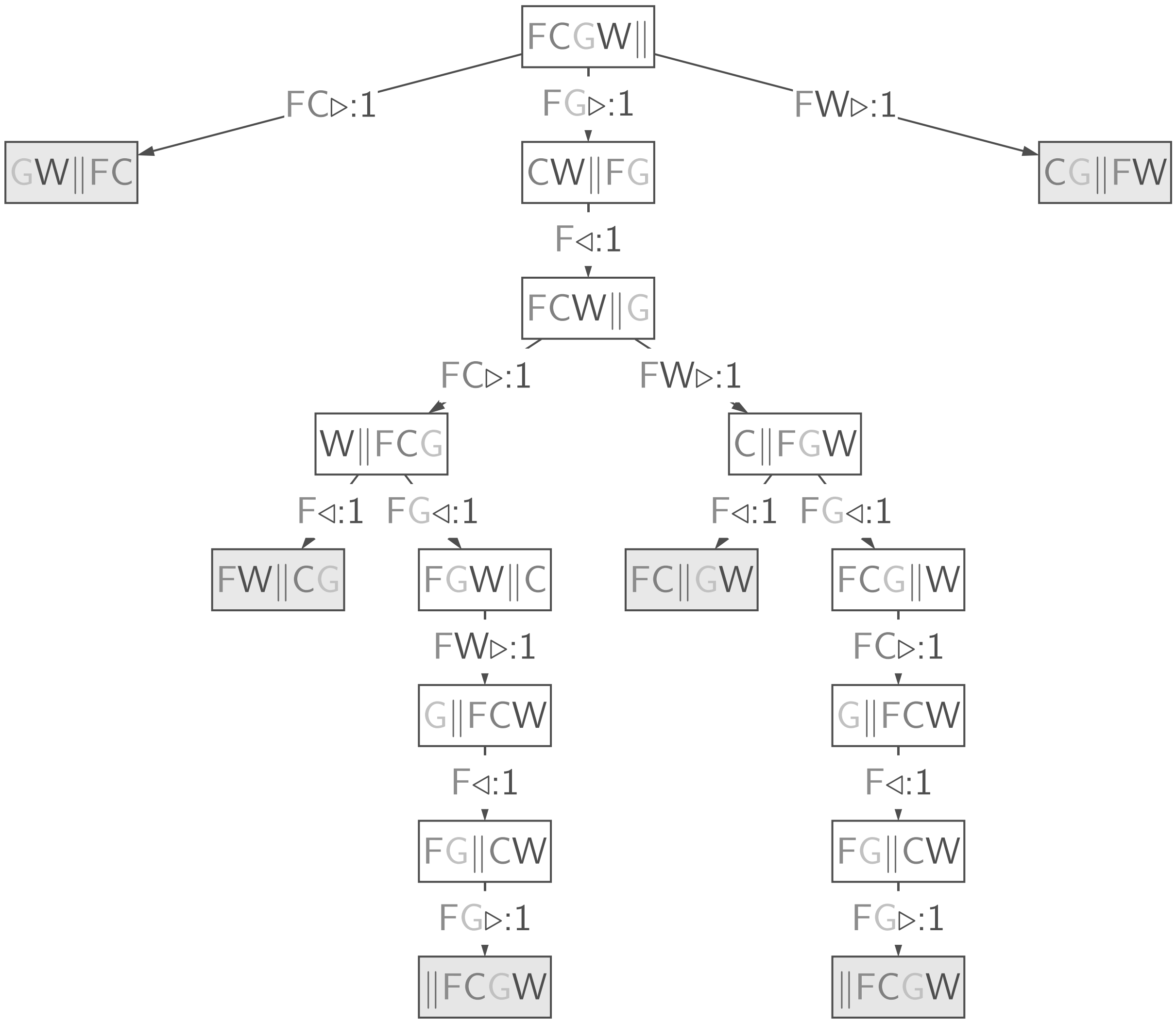


Farmer Cabbage Goat Wolf

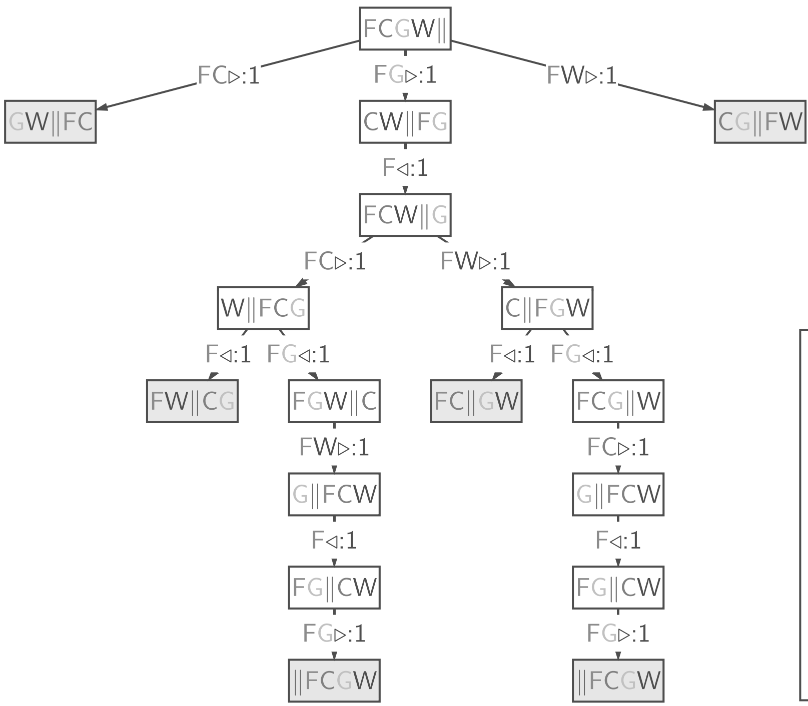
Actions:

F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

Approach: build a **search tree** ("what if?")

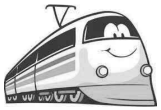

FCGW||
FC▷:1
FG▷:1
FW▷:1
GW||FC
CW||FG
CG||FW
F◁:1
FCW||G
FC▷:1
FW▷:1
W||FCG
C||FGW
F◁:1
FG◁:1
F◁:1
FG◁:1
FW||CG
FGW||C
FC||GW
FCG||W
FW▷:1
FC▷:1
G||FCW
G||FCW
F◁:1
F◁:1
FG||CW
FG||CW
FG▷:1
FG▷:1
||FCGW
||FCGW

Search problem



Definition: search problem

- s_{start}: starting state
- Actions(s): possible actions
- Cost(s, a): action cost
- Succ(s, a): successor
- IsEnd(s): reached end state?



Transportation example

Example: transportation

Street with blocks numbered 1 to n.

Walking from s to $s+1$ takes 1 minute.

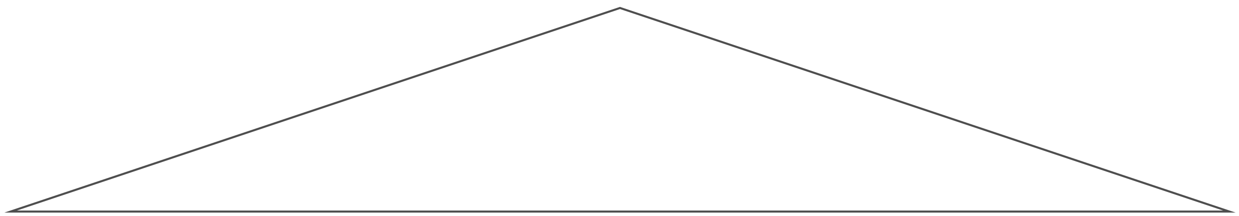
Taking a magic tram from s to $2s$ takes 2 minutes.

How to travel from 1 to n in the least time?

[live solution: `TransportationProblem`]

Search: tree search

Backtracking search



[whiteboard: search tree]

If b actions per state, maximum depth is D actions:

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge) $[2^{50} = 1125899906842624]$

Backtracking search

Algorithm: backtracking search

def backtrackingSearch(s, path):
    If IsEnd(s): update minimum cost path
    For each action $a \in$ Actions(s):
        Extend path with Succ(s, a) and Cost(s, a)
        Call backtrackingSearch(Succ(s, a), path)
    Return minimum cost path

[live solution: `backtrackingSearch`]

Depth-first search

Assumption: zero action costs

Assume action costs $\text{Cost}(s, a) = 0$.

Idea: Backtracking search + stop when find the first end state.

If b actions per state, maximum depth is D actions:

- Space: still $O(D)$
- Time: still $O(b^D)$ worst case, but could be much better if solutions are easy to find

Breadth-first search

Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea: explore all nodes in order of increasing depth.

Legend: b actions per state, solution has d actions

- Space: now $O(b^d)$ (a lot worse!)
- Time: $O(b^d)$ (better, depends on d, not D)

DFS with iterative deepening

Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea:

- Modify DFS to stop at a maximum depth.
- Call DFS for maximum depths $1, 2, \ldots$.

DFS on d asks: is there a solution with d actions?

Legend: b actions per state, solution size d

- Space: $O(d)$ (saved!)
- Time: $O(b^d)$ (same as BFS)

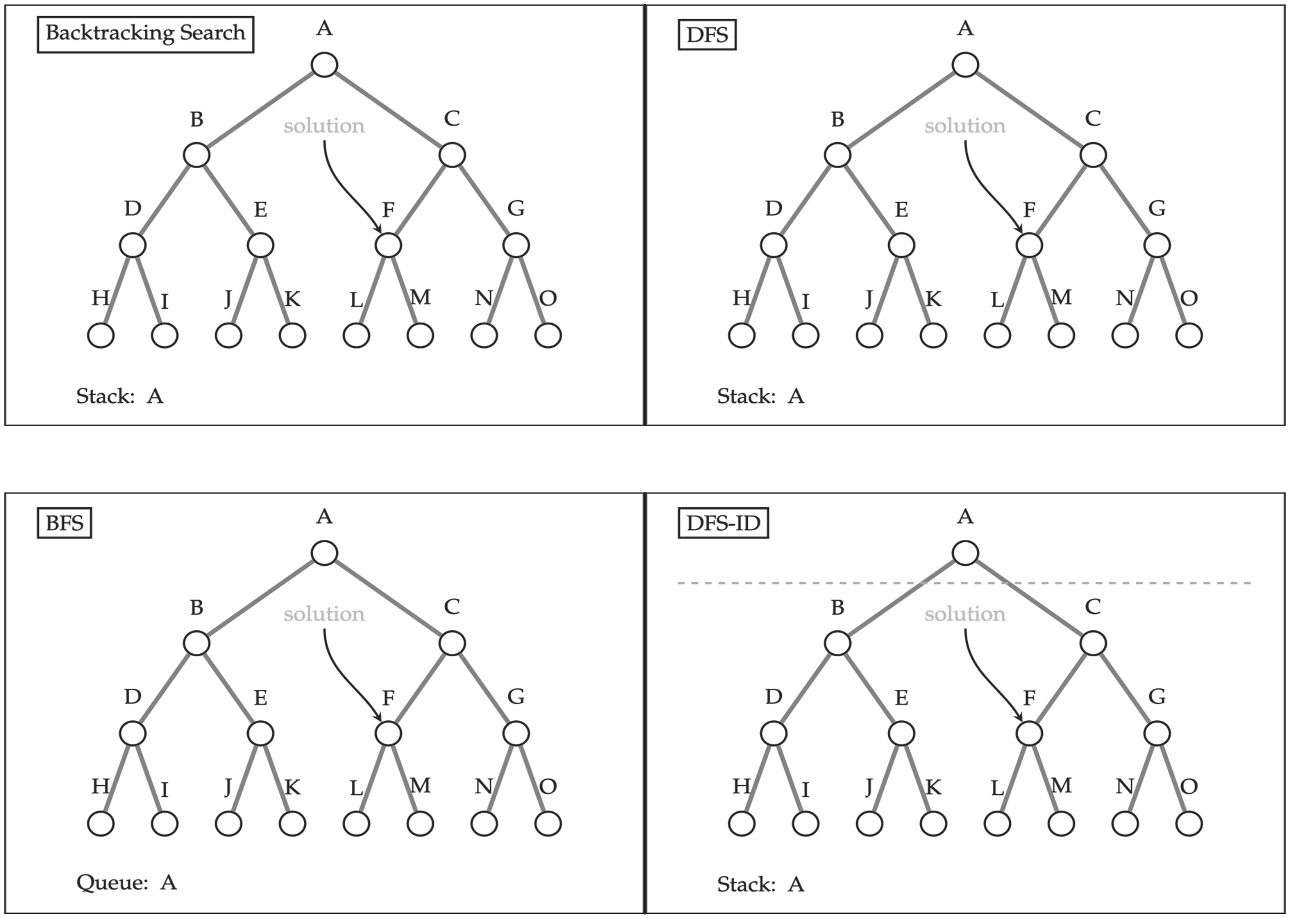
Tree search algorithms

Legend: b actions/state, solution depth d, maximum depth D

Algorithm	**Action costs**	**Space**	**Time**
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

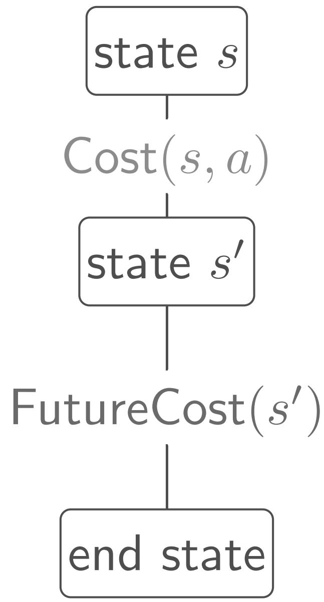
- Always exponential time
- Avoid exponential space with DFS-ID

Tree Search Review



Search: dynamic programming

Dynamic programming



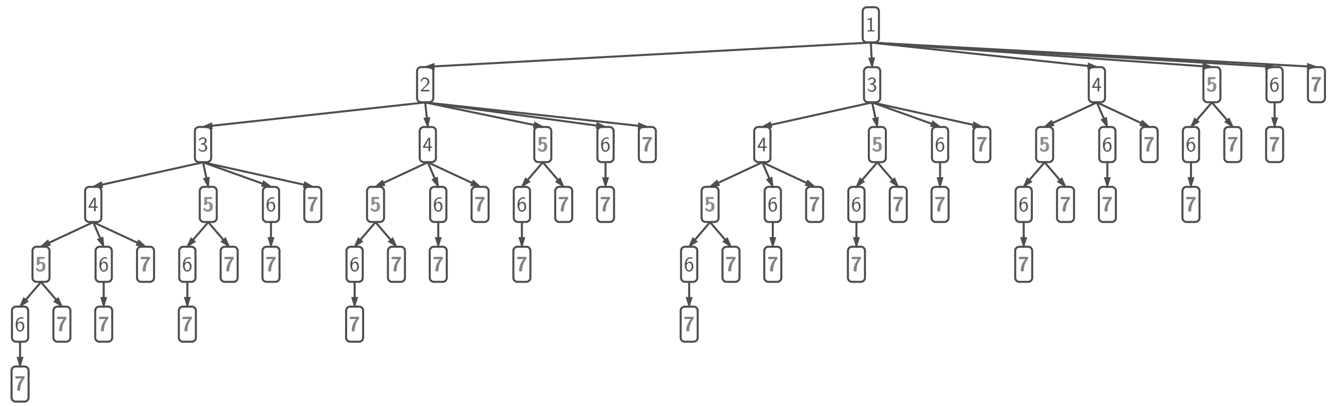
Minimum cost path from state s to a end state:

$$\mathsf{FutureCost}(s) = \begin{cases} 0 & \text{if } \mathsf{IsEnd}(s) \\ \min_{a \in \mathsf{Actions}(s)} [\mathsf{Cost}(s,a) + \mathsf{FutureCost}(\mathsf{Succ}(s,a))] & \text{otherwise} \end{cases}$$

Motivating task

Example: route finding

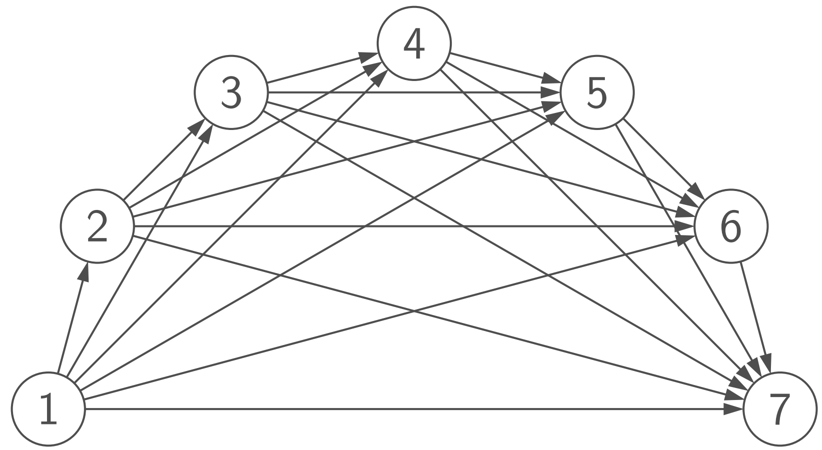
Find the minimum cost path from city 1 to city n, only moving forward. It costs c_{ij} to go from i to j.



Observation: future costs only depend on current city

Dynamic programming

State: ~~past sequence of actions~~ current city



Exponential saving in time and space!

Dynamic programming

Algorithm: dynamic programming

def DynamicProgramming(s):

  If already computed for s, return cached answer.

  If IsEnd(s): return solution

  For each action $a \in$ Actions(s): ...

[live solution: `Dynamic Programming`]

Assumption: acyclicity

The state graph defined by Actions(s) and Succ(s, a) is acyclic.

Dynamic programming

Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities)	1 3 4 6
state (current city)	1 3 4 6

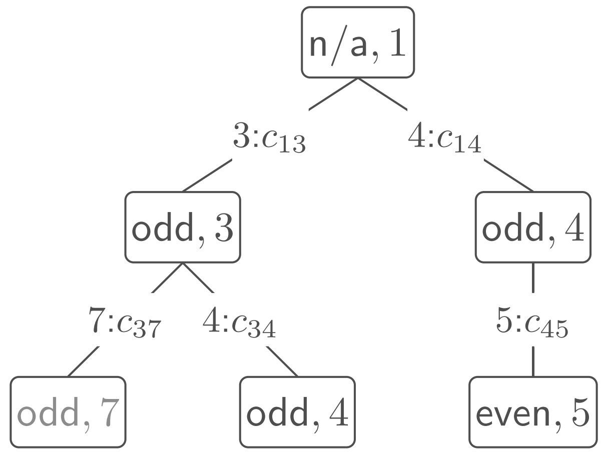
Handling additional constraints

Example: route finding

Find the minimum cost path from city 1 to city n, only moving forward. It costs c_{ij} to go from i to j.

Constraint: Can't visit three odd cities in a row.

State: (whether previous city was odd, current city)



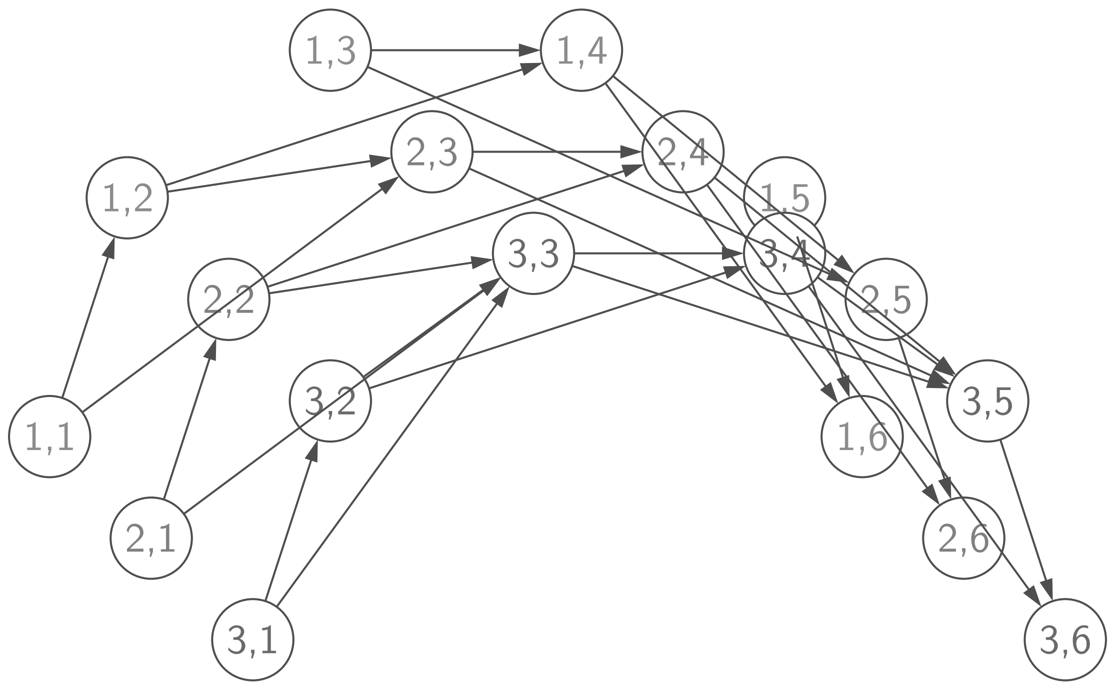
think and share

Question

Objective: travel from city 1 to city n, visiting at least 3 odd cities. What is the minimal state?

State graph

State: (min(number of odd cities visited, 3), current city)



think and share

Question

Objective: travel from city 1 to city n, visiting more odd than even cities. What is the minimal state?

Summary

- State: summary of past actions sufficient to choose future actions optimally

- Dynamic programming: backtracking search with **memoization** — potentially exponential savings

Dynamic programming only works for acyclic graphs...what if there are cycles?

Dynamic Programming Review

state s

$\text{Cost}(s, a)$

state s'

$\text{FutureCost}(s')$

end state

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

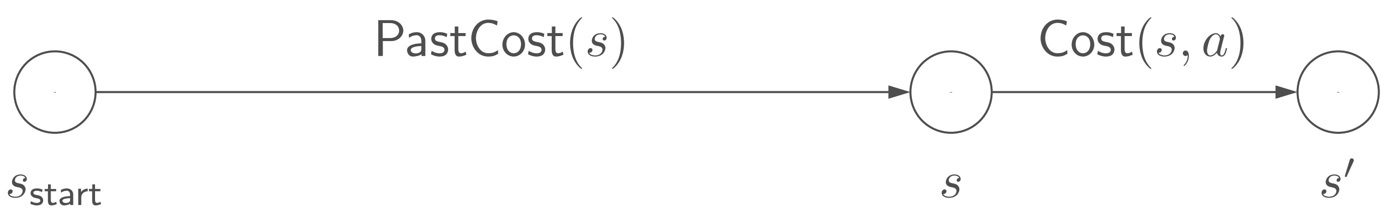
Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Search: uniform cost search

Ordering the states

Observation: prefixes of optimal path are optimal



Key: if graph is acyclic, dynamic programming makes sure we compute PastCost(s) before PastCost(s')

If graph is cyclic, then we need another mechanism to order states...

Uniform cost search (UCS)

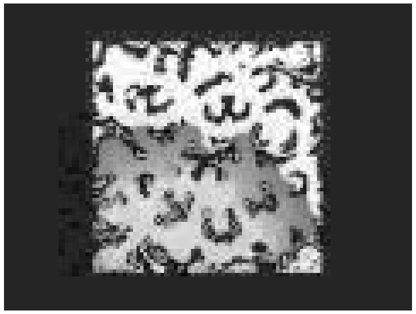
Key idea: state ordering

UCS enumerates states in order of increasing past cost.

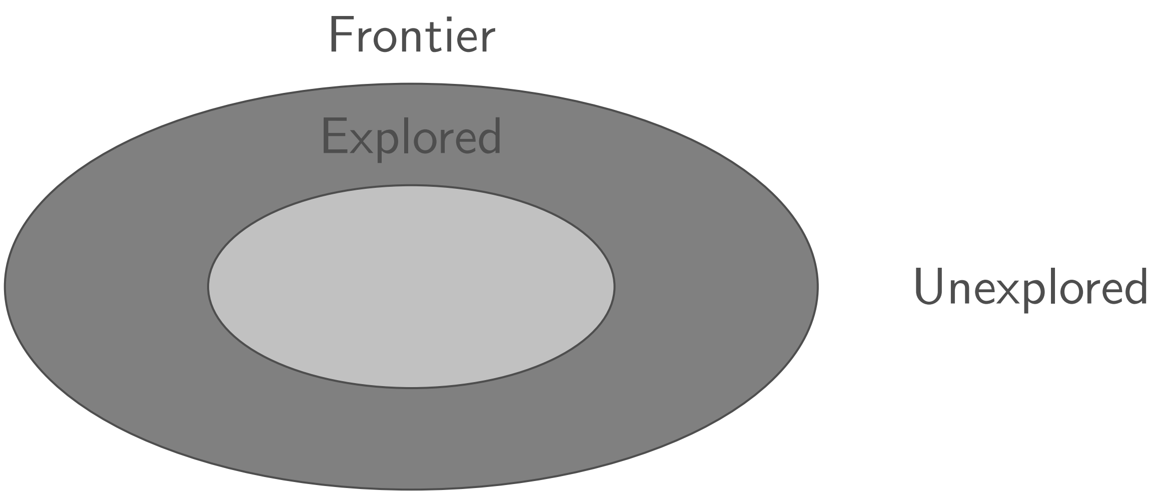
Assumption: non-negativity

All action costs are non-negative: $\text{Cost}(s, a) \geq 0$.

UCS in action:



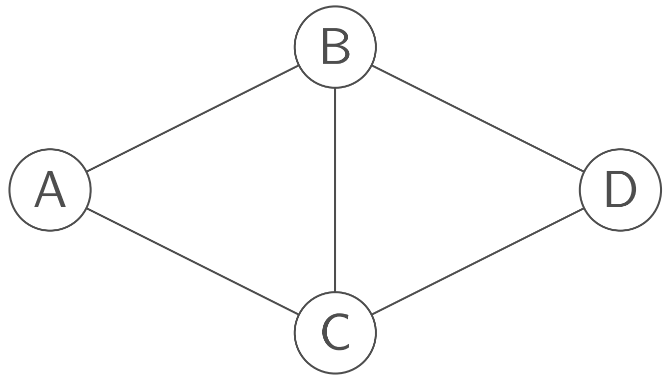
High-level strategy



- Explored: states we've found the optimal path to
- Frontier: states we've seen, still figuring out how to get there cheaply
- Unexplored: states we haven't seen

Uniform cost search example

Example: UCS example



Start state: A, end state: D

[whiteboard]

Minimum cost path:

$$A \rightarrow B \rightarrow C \rightarrow D \text{ with cost } 3$$