Search I:
Tree, Graph search

Hwanjo Yu

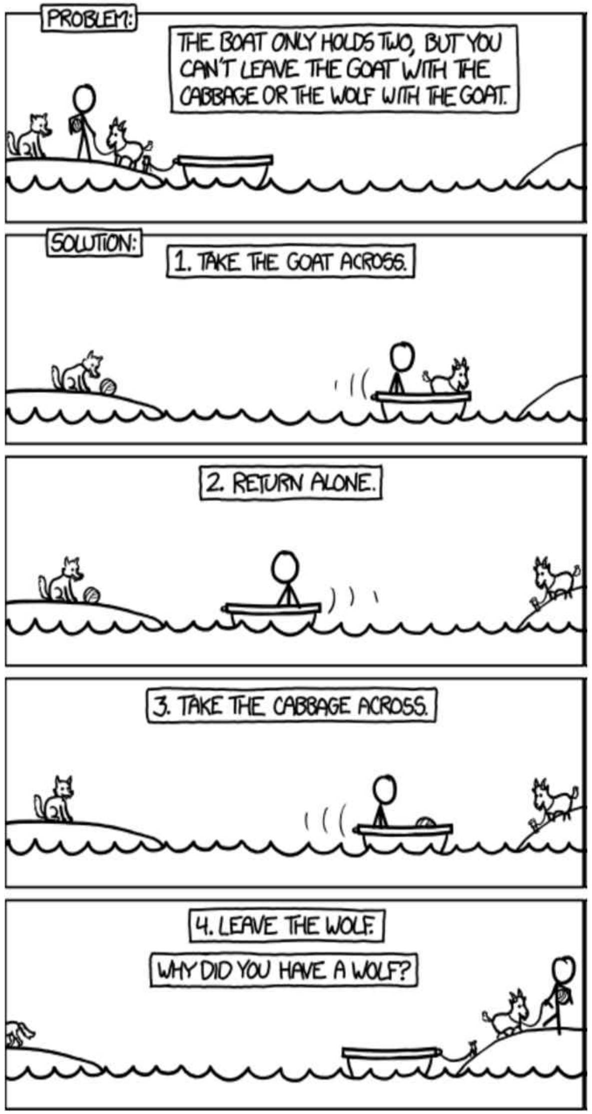
POSTECH

http://di.postech.ac.kr/hwanjoyu

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?

a. 4
b. 5
c. 6
d. 7
e. 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?

Search problem

Markov decision processes

Reinforcement learning & games

Constraint satisfaction problems

Markov networks

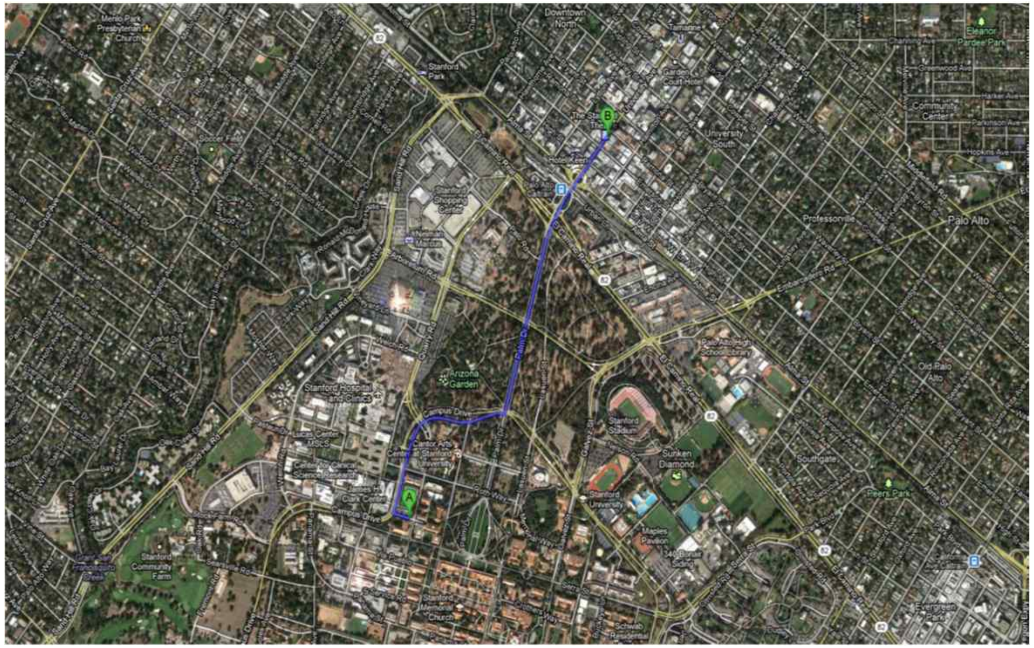
Bayesian networks

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

"Low-level intelligence" "High-level intelligence"

Machine learning

Application: route finding

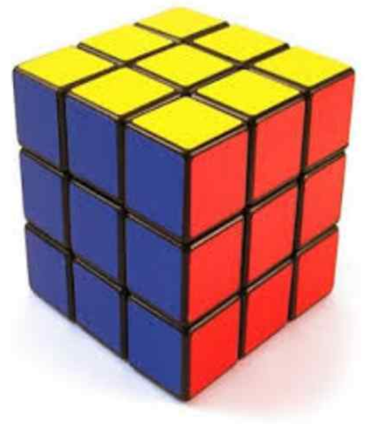


- **Objective**: shortest? fastest? most scenic?
- **Action**: go straight, turn left, turn right

Application: robot motion planning

- **Objective**: fastest? most energy efficient? safest?
- **Action**: translate and rotate joints

Application: solving puzzles



- **Objective**: reach a certain configuration
- **Action**: move pieces (e.g., Move12Down)

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 $(a_1, a_2, a_3, a_4, \ldots)$

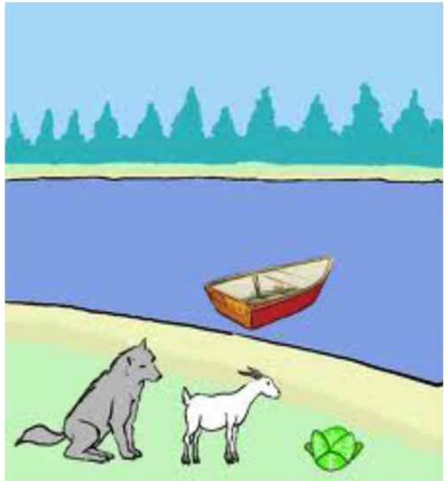
Key: models future consequences of an action!

Roadmap

Tree search

Dynamic programming

Uniform cost search



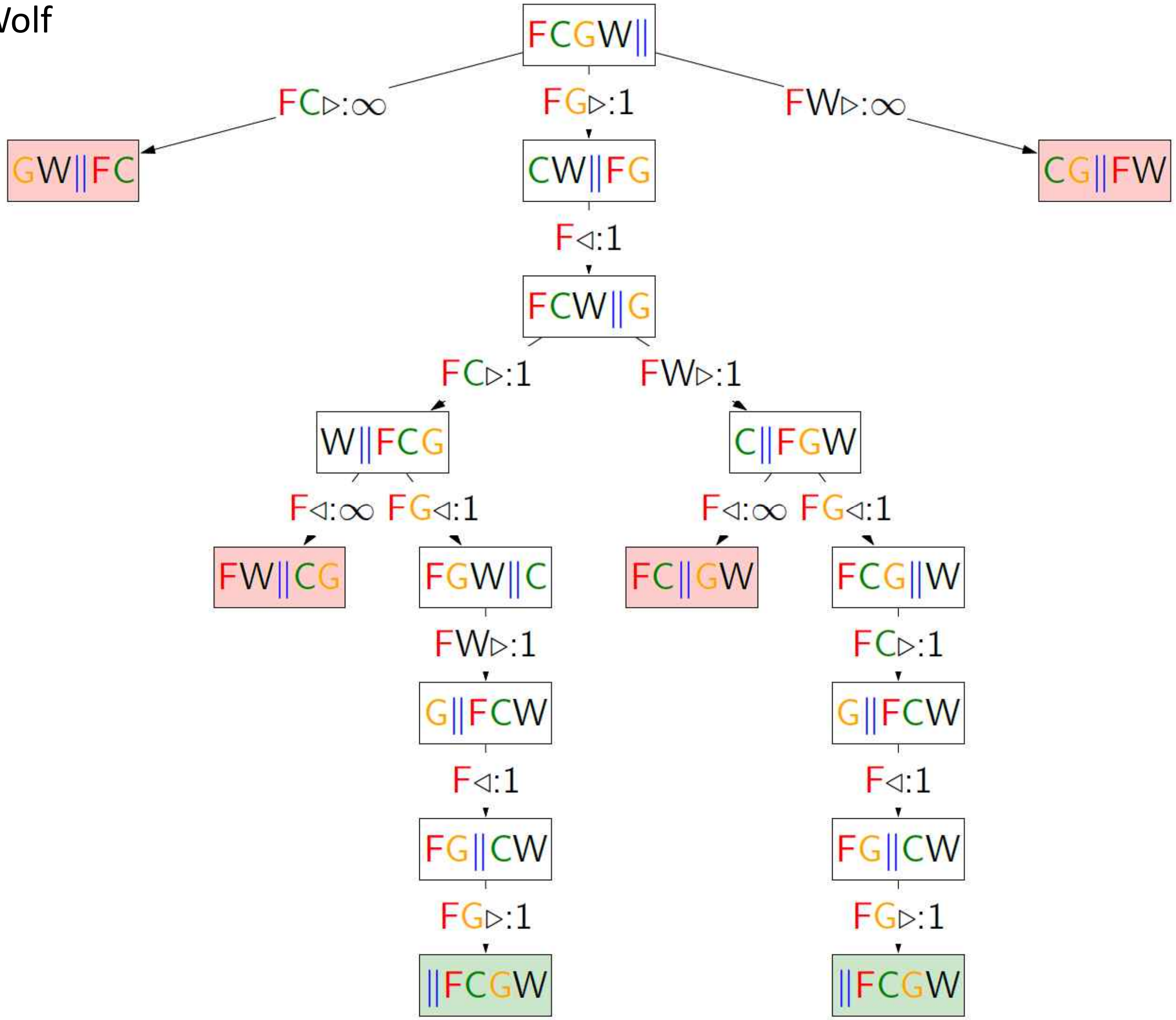
Farmer Cabbage Goat Wolf

Actions:

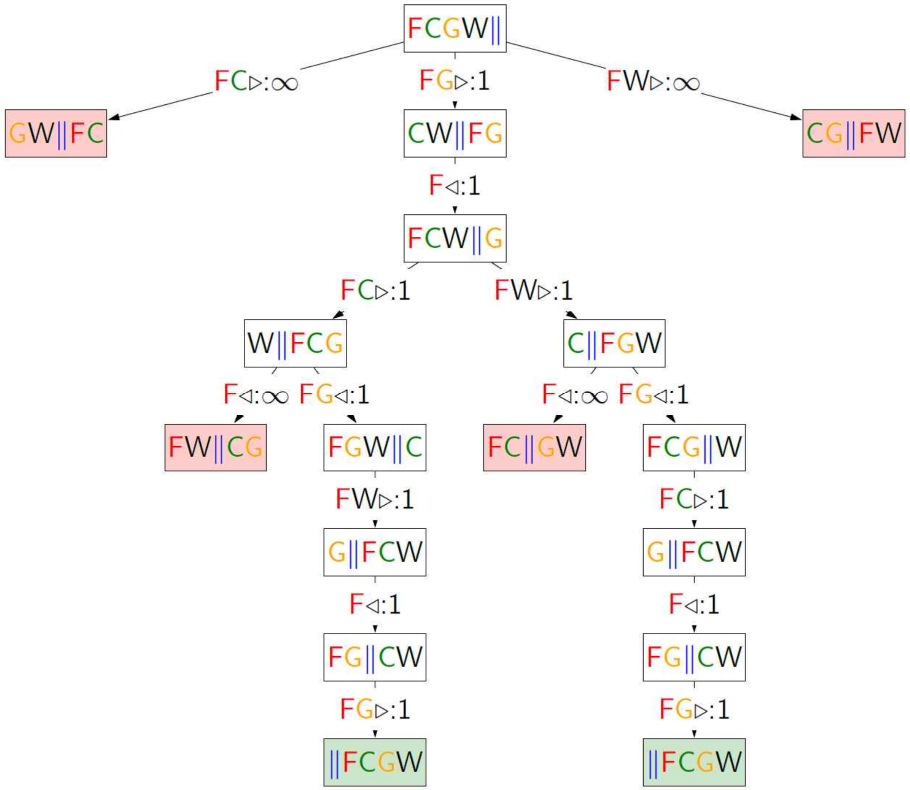
- F⊳ F⊲
- FC⊳ FC⊲
- FG⊳ FG⊲
- FW⊳ FW⊲

Approach: build a search tree (“what if?”)

Farmer Cabbage Goat Wolf



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

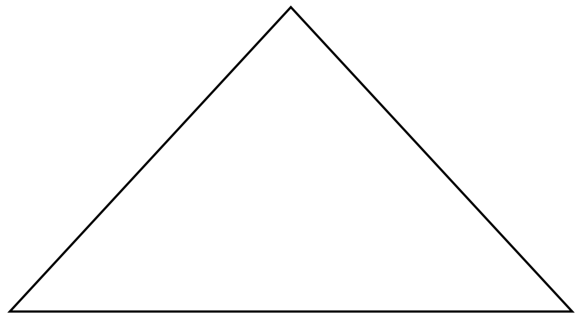
Example: transportation

- 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?

Backtracking search

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?



If $b(=2)$ actions per state, maximum depth is $D(=n)$ actions:

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge) [$2^{50} = 1{,}125{,}899{,}906{,}842{,}624$]

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

- Guarantee to find the minimum path

Depth-first search

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

Assumption: zero action costs (to guarantee to find the minimum path)

- Assume action costs Cost(s, a) = 0

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

Idea: explore all nodes in order of increasing depth.

Assumption: constant (same non-negative) action costs (to guarantee to find the minimum path)

- Assume action costs Cost(s, a) = c for some $c \geq 0$

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

Idea

- Modify DFS to stop at a maximum depth
- Call DFS for maximum depths 1, 2, …
  DFS on d asks: is there a solution with d actions?

Assumption: constant (same non-negative) action costs (to guarantee to find the minimum path)

- Assume action costs Cost(s, a) = c for some $c \geq 0$

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**
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$
Backtracking	any	$O(D)$	$O(b^D)$

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

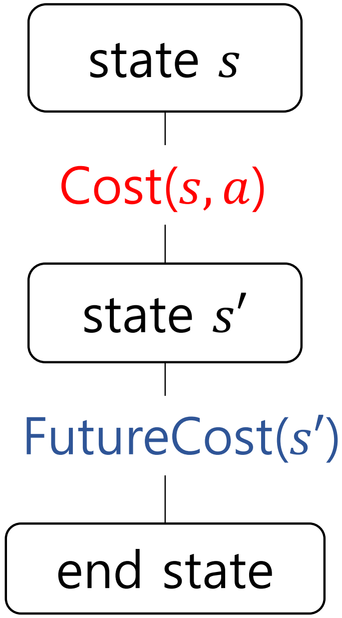
Roadmap

Tree search

Dynamic programming

Uniform cost search

Dynamic programming



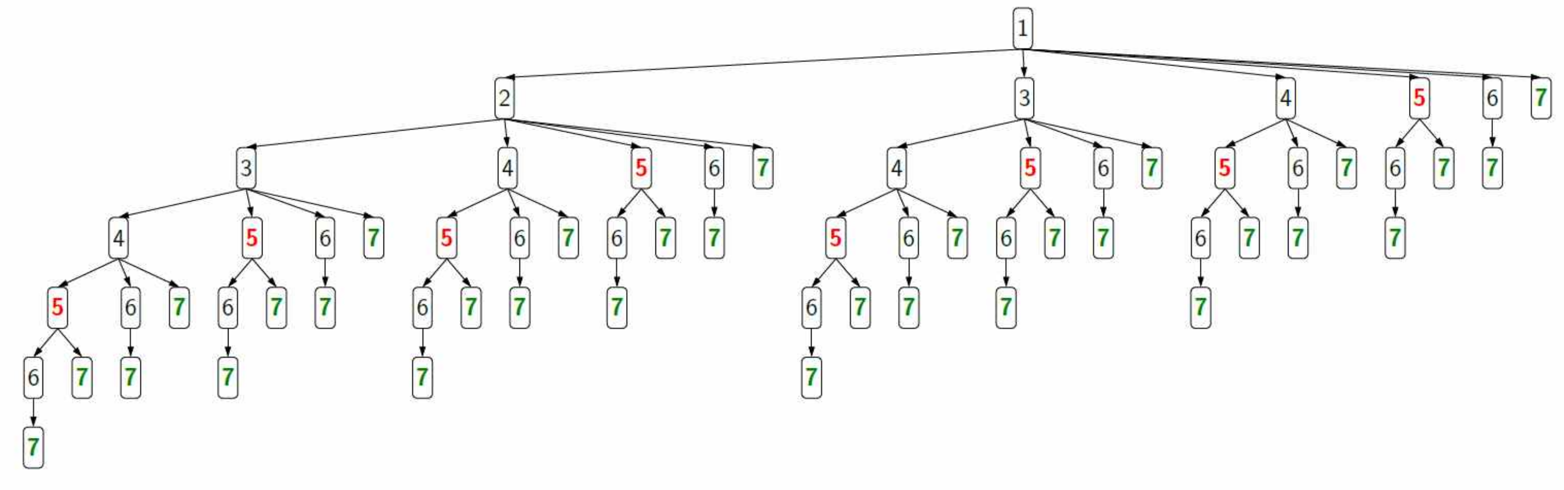
Minimum cost path from state s to an end state:

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

Motivating task

Example: route finding

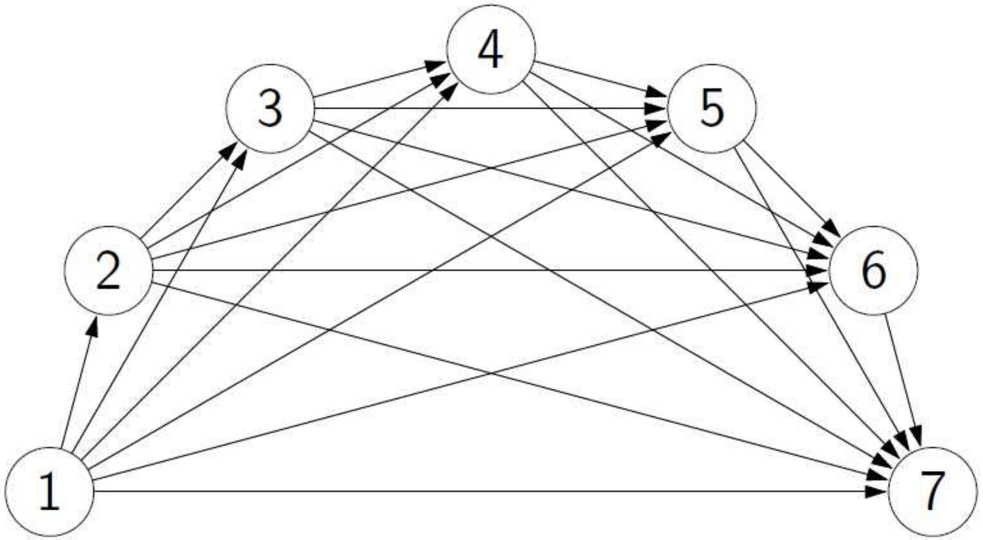
- Find the minimum cost path from city 1 to city n, only moving forward. It costs c_{ij} to go from i to j



Observation: future costs only depend on current city

Dynamic programming

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

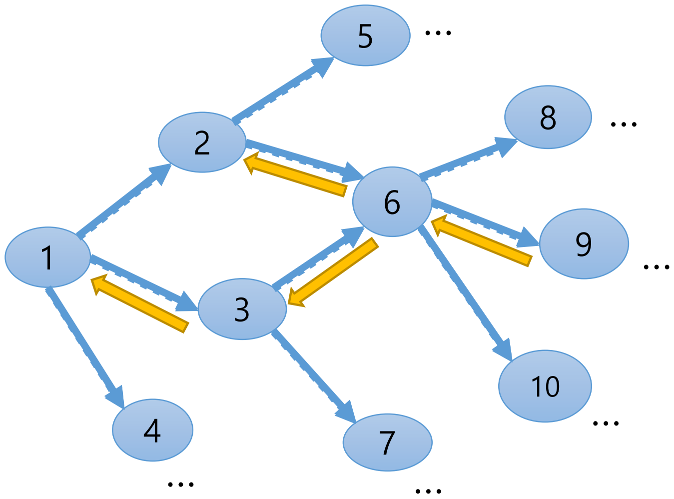


Exponential saving in time and space!

Dynamic programming

Algorithm: dynamic programming

- def DynamicProgramming(s, path):
  - <u>If already computed for s, return cached answer</u>
  - If IsEnd(s): return solution
  - For each action $a \in$ Actions(s):
    - Extend path with Succ(s, a) and Cost(s, a)
    - Call DynamicProgramming(Succ(s, a), path)
  - Return minimum cost path



Assumption: acyclicity

- The state graph defined by Action(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

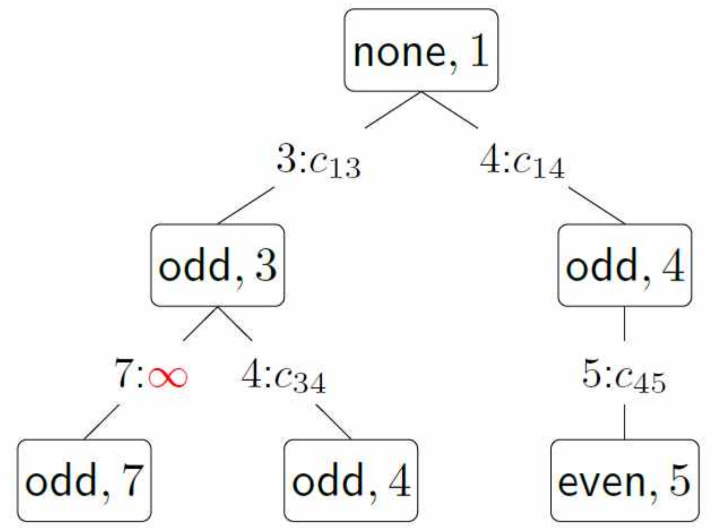
- Find the minimum cost path from city 1 to city n, only moving forward. It costs c_{ij} to go from i to j
- Constraint: Can't visit three odd cities in a row.

State: (previous city, current city)

vs.

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

How many states?



Question

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

State: (# odd cities visited, current city)

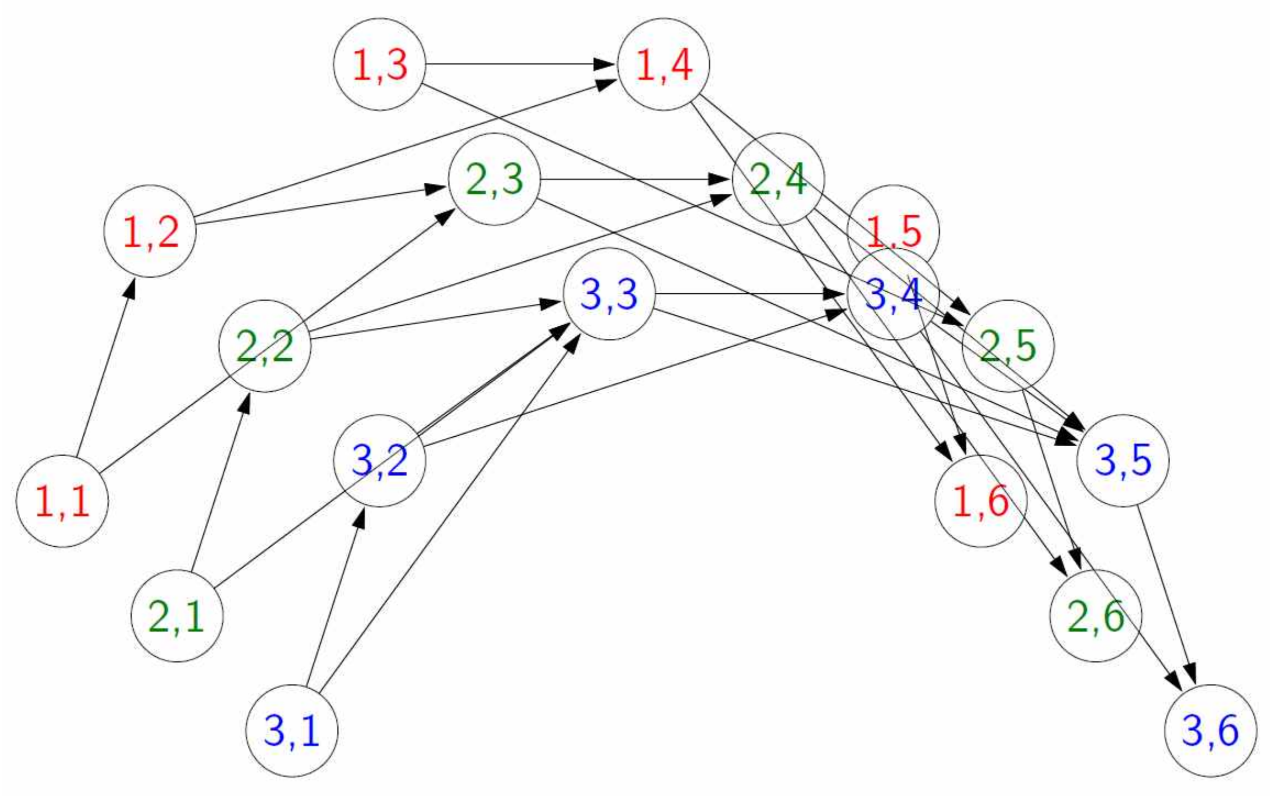
vs.

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

How many states?

State graph

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



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 (and minimal?) 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?

Roadmap

Tree search

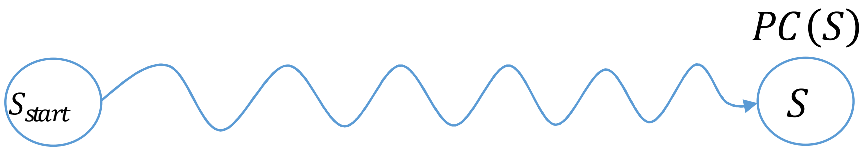
Dynamic programming

Uniform cost search

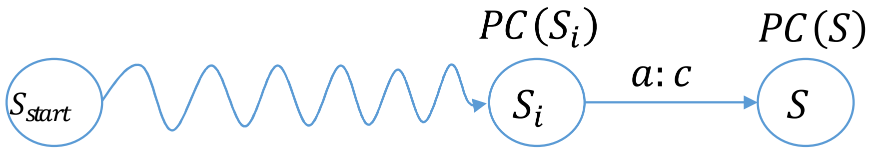
Uniform cost search (UCS)

Observation: prefixes of optimal path are optimal

- PastCost $PC(S)$: minimal cost from S_{start} to S



- $PC(S) = PC(S_i) + Cost\ (a, c)$



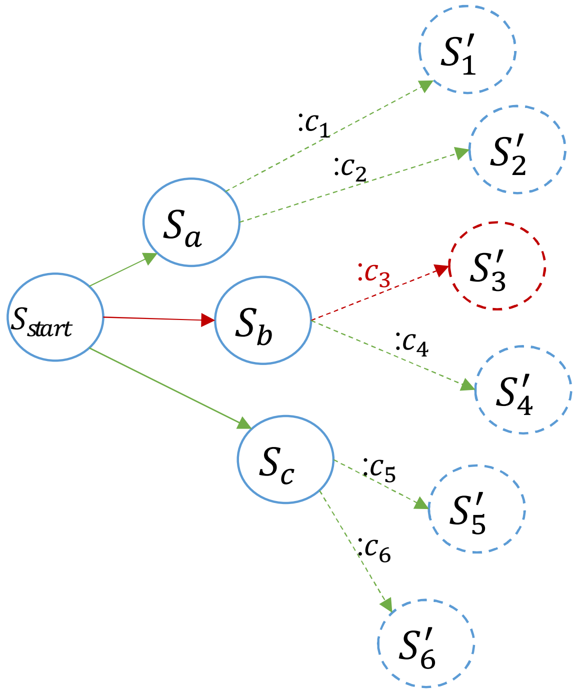
Key idea: state ordering

- <u>UCS enumerates states in order of increasing past cost</u>

Uniform cost search (UCS)

Key idea: state ordering

- <u>UCS enumerates states in order of increasing past cost</u>



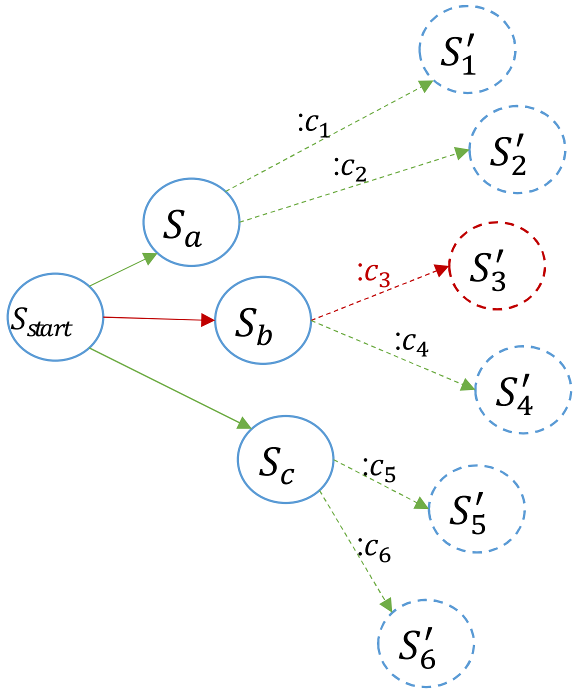
Assume $PC(S_b) + c_3$ *is minimum of* $\begin{cases} PC(S_a) + c_1 \\ PC(S_a) + c_2 \\ PC(S_b) + c_3 \\ PC(S_b) + c_4 \\ PC(S_c) + c_5 \\ PC(S_c) + c_6 \end{cases}.$

Then, $PC(S'_3) = PC(S_b) + c_3$?

Uniform cost search (UCS)

Key idea: state ordering

- <u>UCS enumerates states in order of increasing past cost</u>



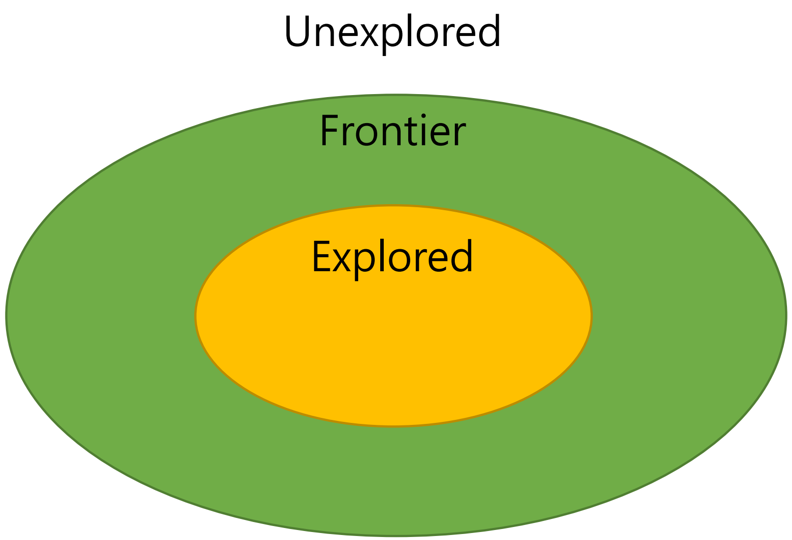
$$\text{Assume } PC(S_b) + c_3 \text{ is minimum of } \begin{cases} PC(S_a) + c_1 \\ PC(S_a) + c_2 \\ PC(S_b) + c_3 \\ PC(S_b) + c_4 \\ PC(S_c) + c_5 \\ PC(S_c) + c_6 \end{cases}.$$

Then, $PC(S_3') = PC(S_b) + c_3$?

Assumption: non-negativity

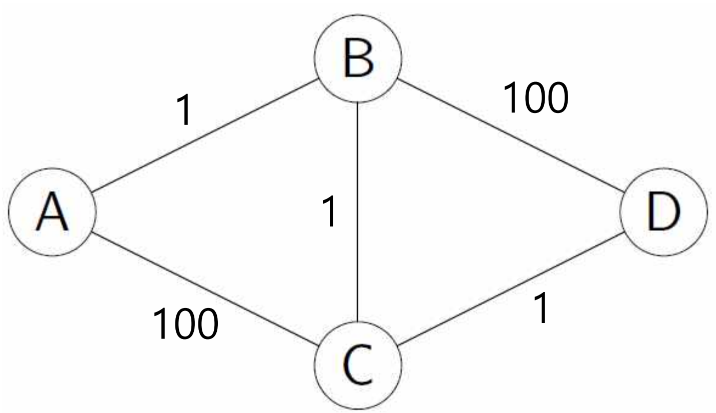
All action costs are non-negative: $\text{Cost}(s, a) \geq 0$

UCS: High-level strategy



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

UCS example



Start state: A, end state: D

Minimum cost path:

A -> B -> C -> D with cost 3

Uniform cost search (UCS)

Algorithm: uniform cost search [Dijkstra, 1956]

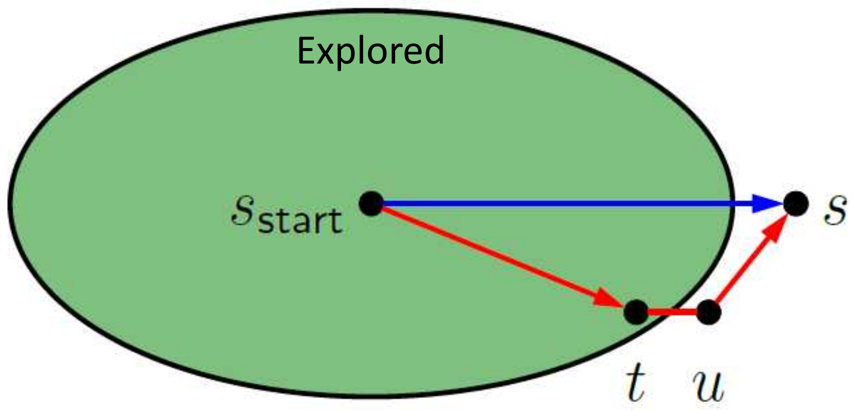
- Add s_{start} to **frontier** (priority queue)
- Repeat until frontier is empty:
  - Remove s with smallest priority p from frontier
  - If IsEnd(s): return solution
  - Add s to **explored**
  - For each action $a \in$ Actions(s):
    - Get successor $s' \leftarrow \text{Succ}(s, a)$
    - If s' already in explored: continue
    - Update **frontier** with s' and priority $p + \text{cost}(s, a)$
    - If updated: $\text{backpointers}[\text{s}'] = (s, a)$

Analysis of uniform cost search

Theorem: correctness

- When a state s is popped from the frontier and moved to explored, its priority is PastCost(s), the minimum cost to s.

Proof:



DP versus UCS

N total states, n of which are closer than end state

Algorithm	**Cycles?**	**Action costs**	**Time/space**
DP	no	any	$O(N)$
UCS	yes	≥ 0	$O(n \log n)$

Note:

- UCS potentially explores fewer states, but requires more overhead to maintain the priority queue
- Assume number of actions per state is constant (independent of n and N)

Summary

- Tree search: memory efficient, suitable for huge state spaces but exponential worst-case running time
- State: summary of past actions sufficient to choose future actions optimally
- Graph search: dynamic programming and uniform cost search construct optimal paths (exponential savings!)
- Next time: learning action costs, searching faster with A*