

4. Search 1 - Dynamic Programming, Uniform Cost Search

State-based models and Search

Intro



cs221.stanford.edu/q

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

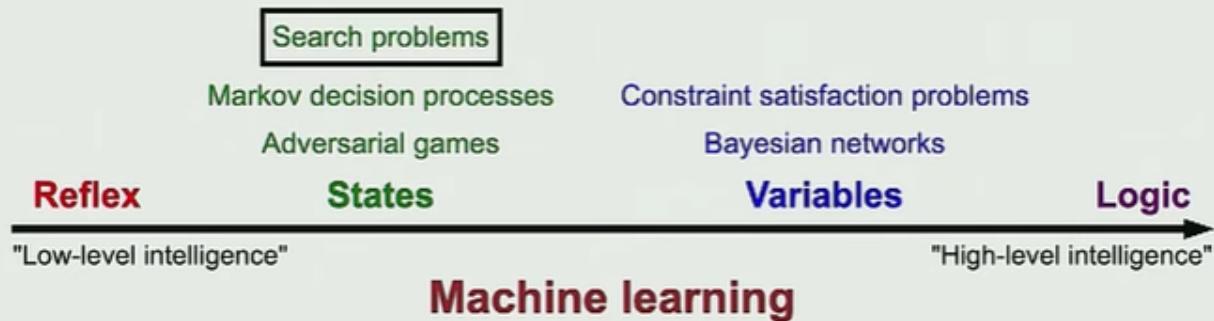
6

7

no solution

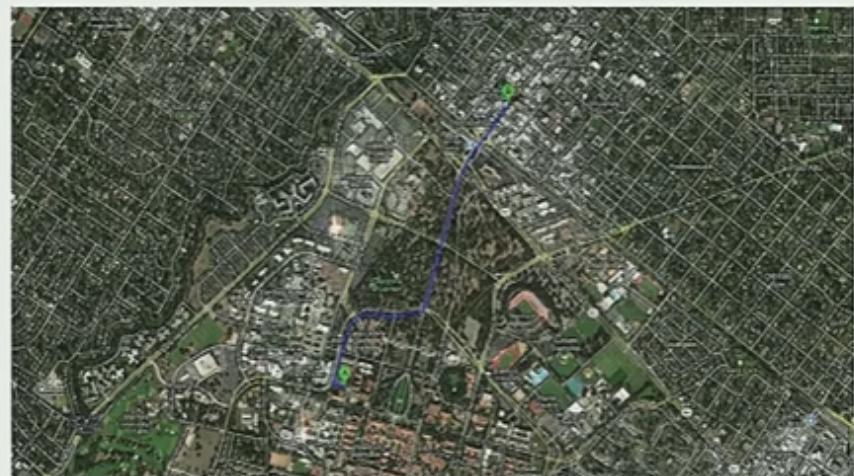
Open question Dynamic Programming Try all the possibilities

Course plan



What are search problems?

Application: route finding



Objective: shortest? fastest? most scenic?

Actions: go straight, turn left, turn right

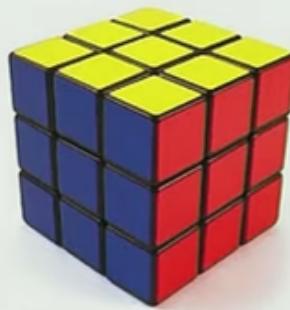
Application: robot motion planning



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

Actions: translate and rotate joints

Application: solving puzzles



Objective: reach a certain configuration

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

Application: machine translation

la maison bleue



the blue house

Objective: fluent English and preserves meaning

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

What's different from reflex-based models?

Beyond reflex

Classifier (reflex-based models):

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

Search problem (state-based models):

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

Key: need to consider future consequences of an action!

For state based models, You need to think about future Cuz for each step, it's gonna change ur state

Road map

We are going to talk about three different algo for doing inference, for searching problems

Tree Search

Enumerate all the actions we can take



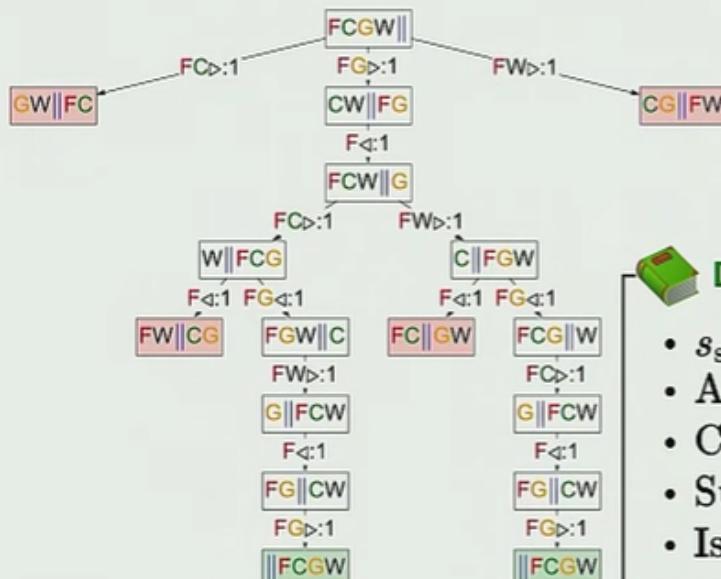
Farmer Cabbage Goat Wolf

Actions:

$F \triangleright$	$F \triangleleft$
$FC \triangleright$	$FC \triangleleft$
$FG \triangleright$	$FG \triangleleft$
$FW \triangleright$	$FW \triangleleft$

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

Search problem



Definition: search problem

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

Walk or tram



Transportation example



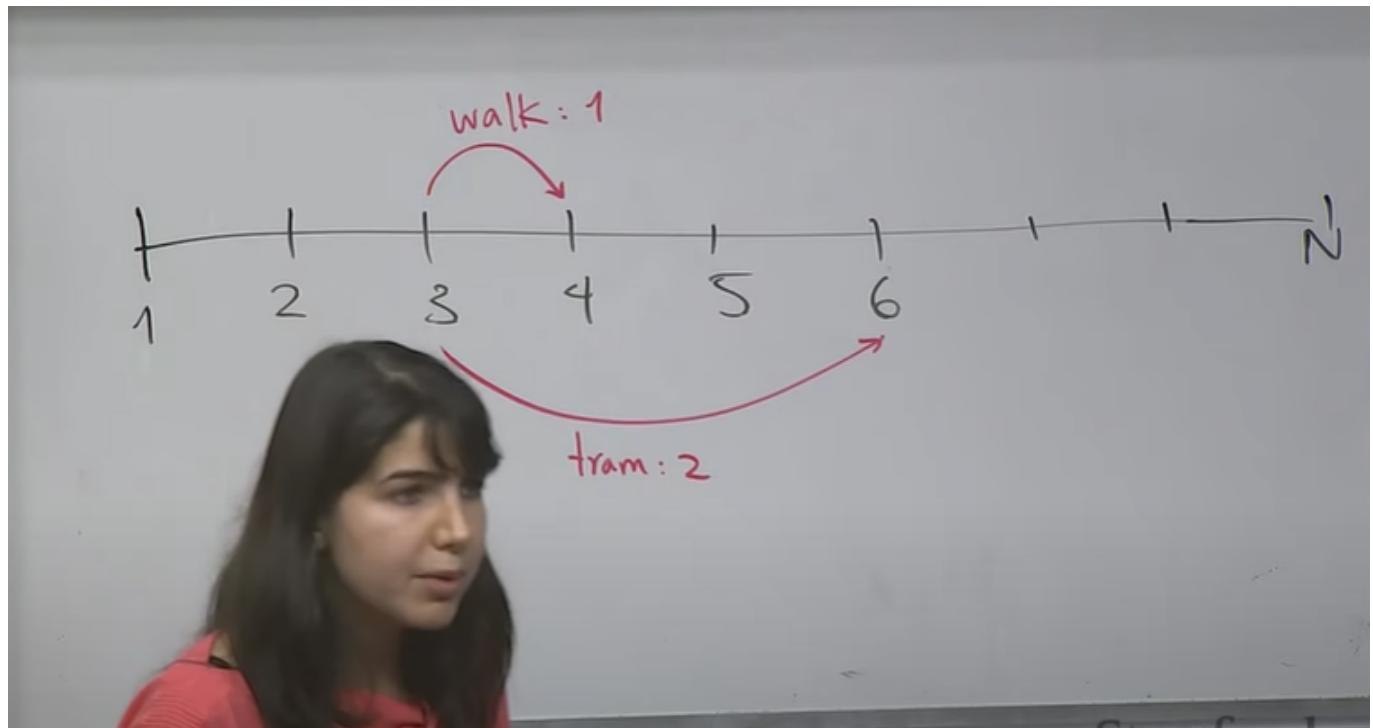
Example: transportation

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?



Defining the search problem model

The terminal window shows the following sequence of events:

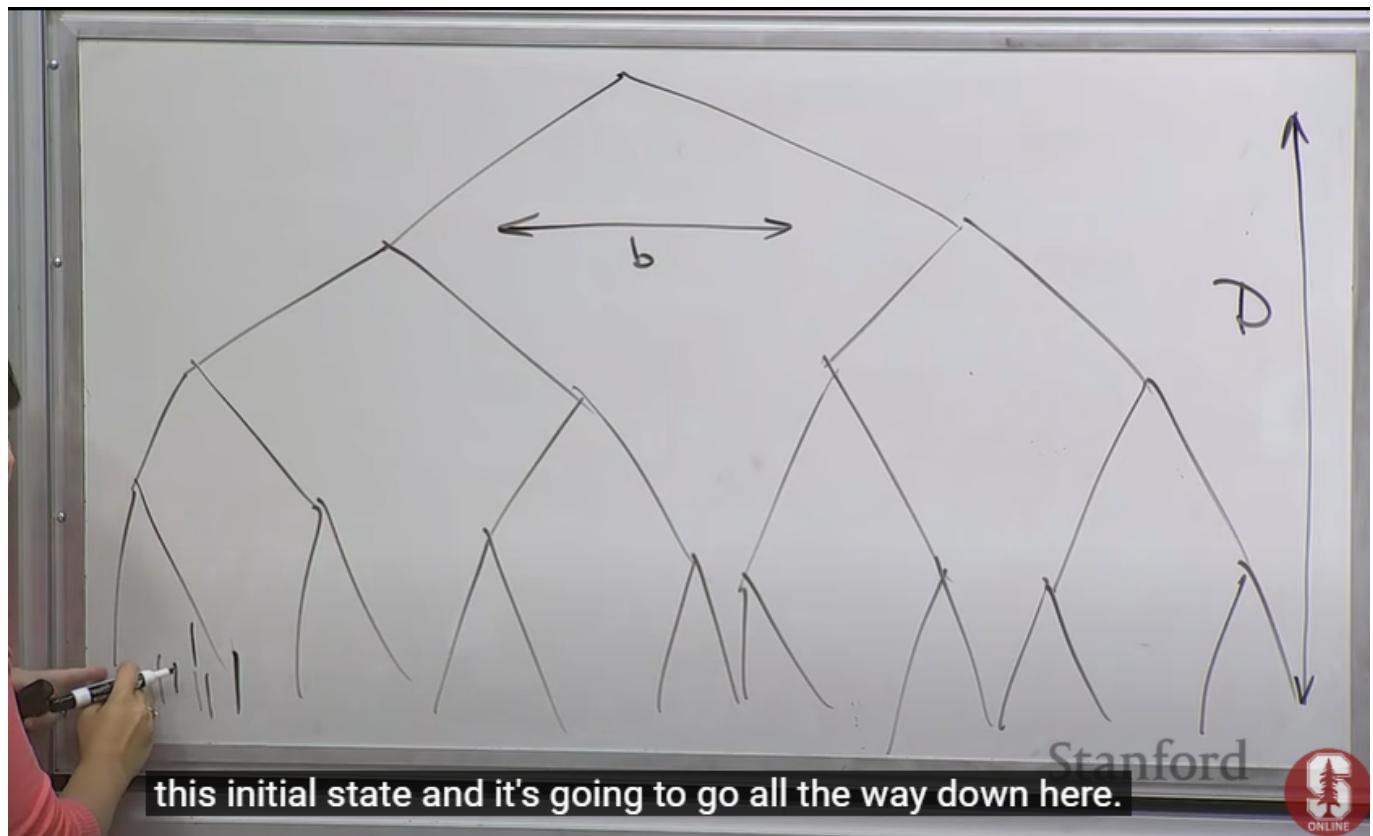
- A Python script named `tram.py` is run from a directory named `search1`.
- The command `git log` is run, showing the commit history for the file.
- The command `python tram.py` is run, which fails with the error message "sh: command not found: python".
- The command `python2 tram.py` is run, which succeeds and prints the output of the script.
- The command `python3 tram.py` is run, which fails with the error message "sh: command not found: python3".
- The command `python3.6 tram.py` is run, which succeeds and prints the output of the script.

```

1 class TransportationProblem(object):
2     def __init__(self, N):
3         # N = number of blocks
4         self.N = N
5     def startState(self):
6         return 1
7     def isEnd(self, state):
8         return state == self.N
9     def succAndCost(self, state):
10        # return list of (action, newState, cost) triples
11        result = []
12        if state+1<=self.N:
13            result.append(('walk', state+1, 1))
14        if state*2<=self.N:
15            result.append(('tram', state*2, 2))
16        return result
17
18 problem = TransportationProblem(N=10)
19 print(problem.succAndCost(3))
20 print(problem.succAndCost(6))
~
```

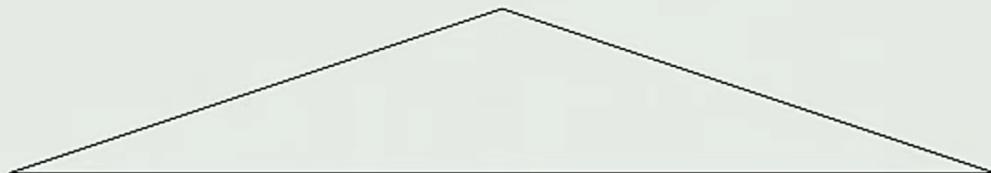
Algo	Cost	Time	Space
Backtracking Search	Any	$O(b^D)$	$O(D)$
DFS	0	Worst case $O(b^D)$	$O(D)$
BFS	cost ≥ 0 (assuming all the cost are the same)	Worst Case $O(b^D)$	Worst case $O(b^D)$???
DFS - ID	cost ≥ 0 (assuming all the cost are the same)	Worst case $O(b^D)$	$O(D)$

Backtracking Search



b : branching factor (How many branches it gonna have from one node) D : Depth of the tree

Backtracking search



[whiteboard: search tree]

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

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

The time complexity is quite bad though lol

DFS

When you don't care the cost of going back and forth

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

BFS

This is useful when the costs are similar

DFS - Iterative deepening

A combination of BFS and DFS

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, ...

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)

Stanford

You can visit the same node multiple times, because you are running DFS each time you proceed

SO in the worst case, when u have to search through the whole tree, the actual time complexity is $O(b^d * b^d) = O((b^d)^2) = O(b^{2d})$. However, in time complexity $O()$, there's no big difference between b^{2d} and b^d , cuz we care about the 数量级 only.



Tree search algorithms

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

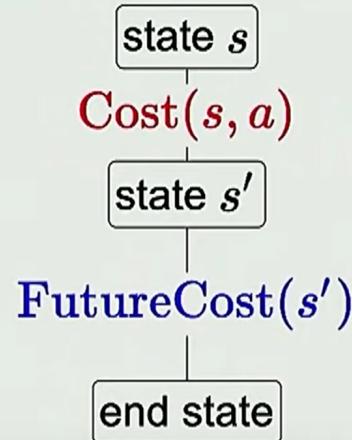
Algorithm	Action costs	Space	Time
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

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

The exponential time is not good, that's when dynamic programming comes to play 😊

Dynamic Programming

Dynamic programming



Minimum cost path from state s to a end state:

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

Reduce the recomputation

State: a summary of all past actions

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

Limitation

It does not run well with graph with cycles

Define the state

What if we have a rule saying you cannot go 3 odd cities in a row? We have to have a context of our past actions

State [prev city, cur city] state space: n^2 The state space is too big, thus the program will be complex =>

State [bool(if prev was odd), cur city] state space: 2^*n

$$S = \left(\# \text{ of odd cities}, \text{current city} \right) \quad |S| = \frac{N^2}{2}$$

$$S = \left(\min(\# \text{ of odd cities}, 3), \text{current city} \right) \quad |S| = 3N$$

For example #means number This can bring the complexity of state space from $n^2 \Rightarrow 3n$, which is linear



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?

Bring the time O() from exponential \Rightarrow polynomial

Uniform Cost Search

Uniform cost search (UCS)



Key idea: state ordering

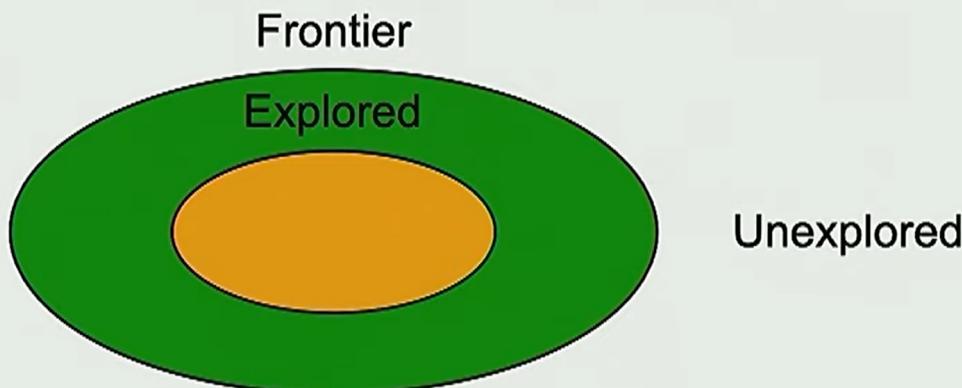
UCS enumerates states in order of increasing past cost.



Assumption: non-negativity

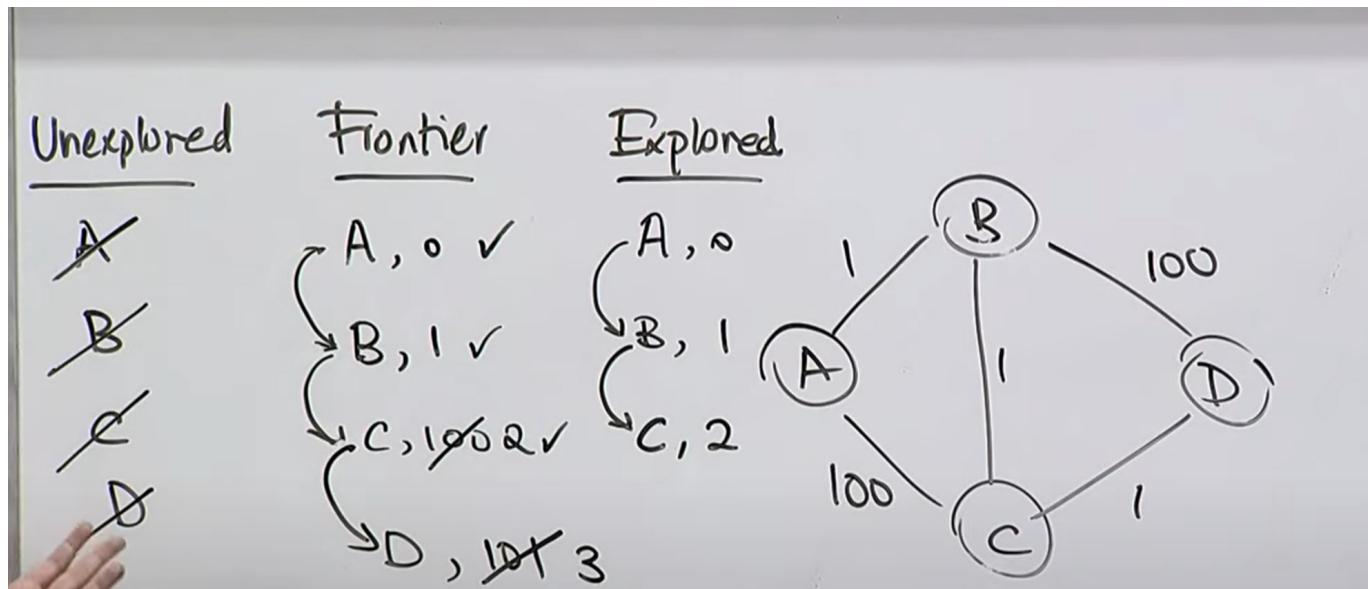
All action costs are non-negative: $\text{Cost}(s, a) \geq 0$.

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

Frontier: Explored by not sure about the optimal path to get there YET



In runtime, pop out the best one from forntier

What is the difference from A*??

Code

```

from enum import Enum
import sys
from queue import PriorityQueue

sys.setrecursionlimit(100000)

Destination = 10

### Model (Search Problem)
class TransportationProblem(object):

    WALK_COST = 1
    TRAM_COST = 2

    WALK = "walk"
    TRAM = "tram"

    def __init__(self, destination):
        # N number of blocks
        self.destination = destination

    def startState(self) -> int:
        return 1

    def isEnd(self, state):
        return state == self.destination

    def succAndCost(self, state : int):
        ...
  
```

```
        Return a list of (action, newState, cost) triples
        Meaning return the a list of: actions we can take, what new state we gonna
endup at, and what the cost gonna be
        ...
        result = []
        if(state + 1 <= self.destination):
            result.append((self.WALK, state + 1, self.WALK_COST))

        if(state * 2 <= self.destination):
            result.append((self.TRAM, state * 2, self.TRAM_COST))

    return result

### Algorithms
def backtrackingSearch(self, problem):

    best = {
        "totalCost" : sys.maxsize,
        "history": None
    }

    memo = {}

    def recurse(currentState, history, totalCost):

        if(currentState in memo):
            for totalCost, history in memo[currentState]:
                best["totalCost"] = totalCost
                best["history"] = history
            return

        if(problem.isEnd(currentState)):
            #Update the best cost if we find a better solution
            if(totalCost < best["totalCost"]):
                best["totalCost"] = totalCost
                best["history"] = history
                memo[currentState] = (totalCost, history)

        for action, newState, cost in problem.succAndCost(currentState):
            recurse(newState, history + [(action, newState, cost)], totalCost +
cost)

    recurse(problem.startState(), history=[], totalCost=0)

    return best

def printSolution(solution):
    totalCost = solution["totalCost"]
    history = solution["history"]
    print("minimum cost is {}".format(totalCost))

    for h in history:
        print(h)
```

```
# You just need to know the current state
def dynamicProgramming(problem):

    memo = {} # state -> futureCost(state) action, newState, cost
    def futureCost(state):

        if problem.isEnd(state):
            return 0

        if state in memo:
            return memo[state][0]

        minFutureCostWithAction = min(
            (curCost + futureCost(newState), action, newState, curCost)
            for action, newState, curCost in problem.succAndCost(state)
        )

        memo[state] = minFutureCostWithAction
        minFutureCost = minFutureCostWithAction[0]
        return minFutureCost

    state = problem.startState()
    minCost = futureCost(state)

    # Recover History
    history = []
    while not problem.isEnd(state):
        _, action, newState, cost= memo[state]
        history.append((action, newState, cost))
        state = newState

    return {
        "totalCost" : minCost,
        "history": history
    }

# searching method in a graph that visits nodes in order of their path cost from
the start node.
# It delves into the graph, visiting the node with the smallest cumulative path
cost first.
def UniformCostSearch(problem):
    frontier = PriorityQueue()
    frontier.put((0, problem.startState()))

    while(True):
        # Move minumun cost from frontier to explored
        pastCost, state = frontier.get()
```

```
# If the end state popped up, meaning we have already have the minimum
cost to the end state
# Everything else left in the queue would have a larger past cost.
# If it's not (The end state may or may not in the queue yet), meaning
there's other possible path
# That may have a lower cost
if(problem.isEnd(state)):
    return {
        "totalCost" : pastCost,
        "history": []
    }

# Add all the successors of current state to frontier
for action, newState, cost in problem.succAndCost(state):
    frontier.put((pastCost + cost, newState))

### Inference
problem = TransportationProblem(destination=Destination)
# solution = problem.backtrackingSearch(problem)
solution = dynamicProgramming(problem)
printSolution(solution)

# solution = UniformCostSearch(problem)
# problem.printSolution(solution)
# print(problem.succAndCost(9))
```