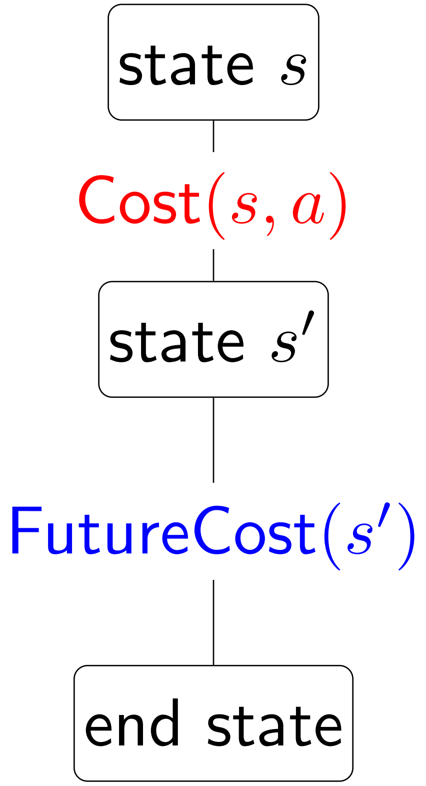
Search: dynamic programming

Dynamic programming



Minimum cost path from state s to a end state:

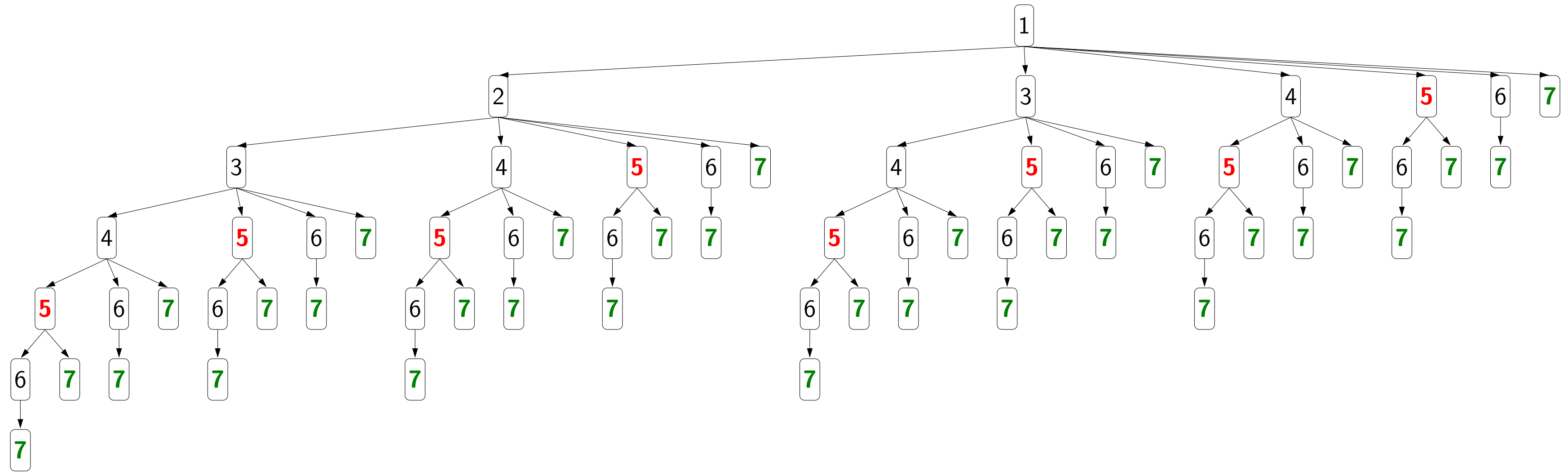
$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

- Now let's see if we can avoid the exponential running time of tree search. Our first algorithm will be dynamic programming. We have already seen dynamic programming in specific contexts. Now we will use the search problem abstraction to define a single dynamic program for all search problems.
- First, let us try to think about the minimum cost path in the search tree recursively. Define $\text{FutureCost}(s)$ as the cost of the minimum cost path from s to some end state. The minimum cost path starting with a state s to an end state must take a first action a, which results in another state s', from which we better take a minimum cost path to the end state.
- Written in symbols, we have a nice recurrence. Throughout this course, we will see many recurrences of this form. The basic form is a base case (when s is a end state) and an inductive case, which consists of taking the minimum over all possible actions a from s, taking an initial step resulting in an **immediate** action cost $\text{Cost}(s, a)$ and a **future** cost.

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.

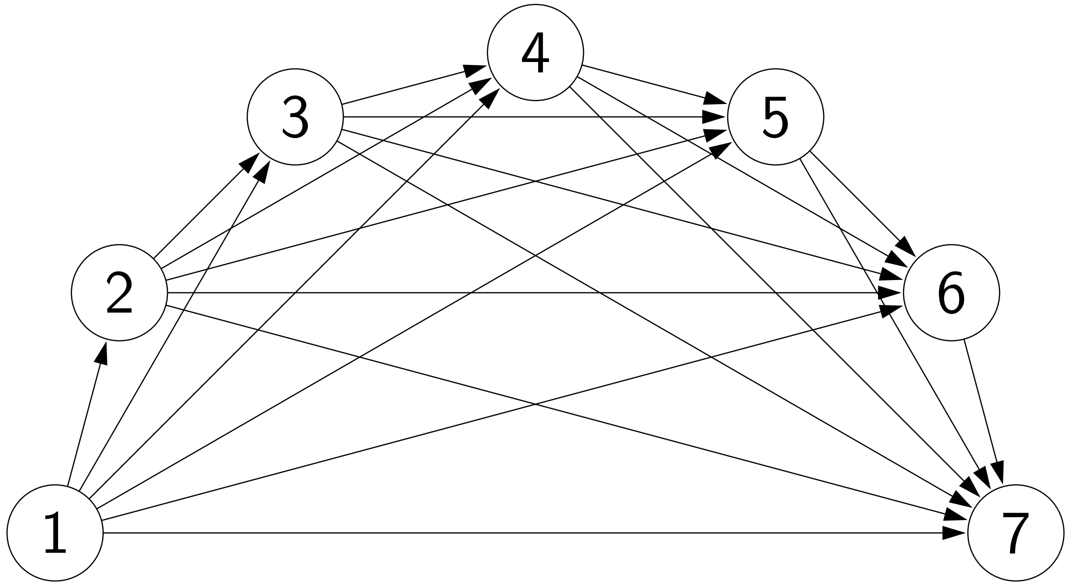


Observation: future costs only depend on current city

- Now let us see if we can avoid the exponential time. If we consider the simple route finding problem of traveling from city 1 to city n, the search tree grows exponentially with n.
- However, upon closer inspection, we note that this search tree has a lot of repeated structures. Moreover (and this is important), the future costs (the minimum cost of reaching a end state) of a state only depends on the current city! So therefore, all the subtrees rooted at city 5, for example, have the same minimum cost!
- If we can just do that computation once, then we will have saved big time. This is the central idea of **dynamic programming**.
- We've already reviewed dynamic programming in the first lecture. The purpose here is to construct one generic dynamic programming solution that will work on any search problem. Again, this highlights the useful division between modeling (defining the search problem) and algorithms (performing the actual search).

Dynamic programming

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



Exponential saving in time and space!

- Let us collapse all the nodes that have the same city into one. We no longer have a tree, but a directed acyclic graph with only n nodes rather than exponential in n nodes.
- Note that dynamic programming is only useful if we can define a search problem where the number of states is small enough to fit in memory.

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($s$): ...
```

[semi-live solution: `Dynamic Programming`]

Assumption: acyclicity

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

- The dynamic programming algorithm is exactly backtracking search with one twist. At the beginning of the function, we check to see if we've already computed the future cost for s. If we have, then we simply return it (which takes constant time if we use a hash map). Otherwise, we compute it and save it in the cache so we don't have to recompute it again. In this way, for every state, we are only computing its value once.
- For this particular example, the running time is $O(n^2)$, the number of edges.
- One important point is that the graph must be acyclic for dynamic programming to work. If there are cycles, the computation of a future cost for s might depend on s' which might depend on s. We will infinite loop in this case. To deal with cycles, we need uniform cost search, which we will describe later.

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

- So far, we have only considered the example where the cost only depends on the current city. But let's try to capture exactly what's going on more generally.
- This is perhaps the most important idea of this lecture: **state**. A state is a summary of all the past actions sufficient to choose future actions optimally.
- What state is really about is forgetting the past. We can't forget everything because the action costs in the future might depend on what we did on the past. The more we forget, the fewer states we have, and the more efficient our algorithm. So the name of the game is to find the minimal set of states that suffice. It's a fun game.

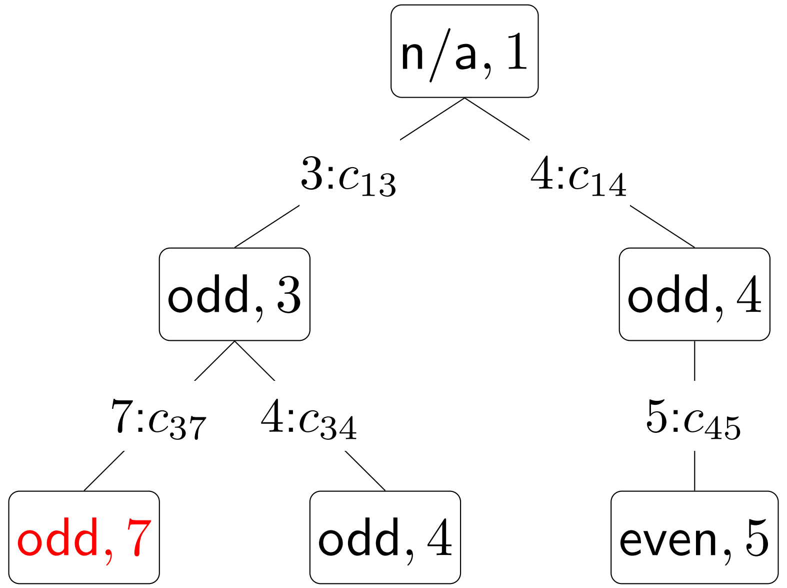
Handling additional constraints

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)



- Let's add a constraint that says we can't visit three odd cities in a row. If we only keep track of the current city, and we try to move to a next city, we cannot enforce this constraint because we don't know what the previous city was. So let's add the previous city into the state.
- This will work, but we can actually make the state smaller. We only need to keep track of whether the previous city was an odd numbered city to enforce this constraint.
- Note that in doing so, we have $2n$ states rather than n^2 states, which is a substantial savings. So the lesson is to pay attention to what information you actually need in the state.

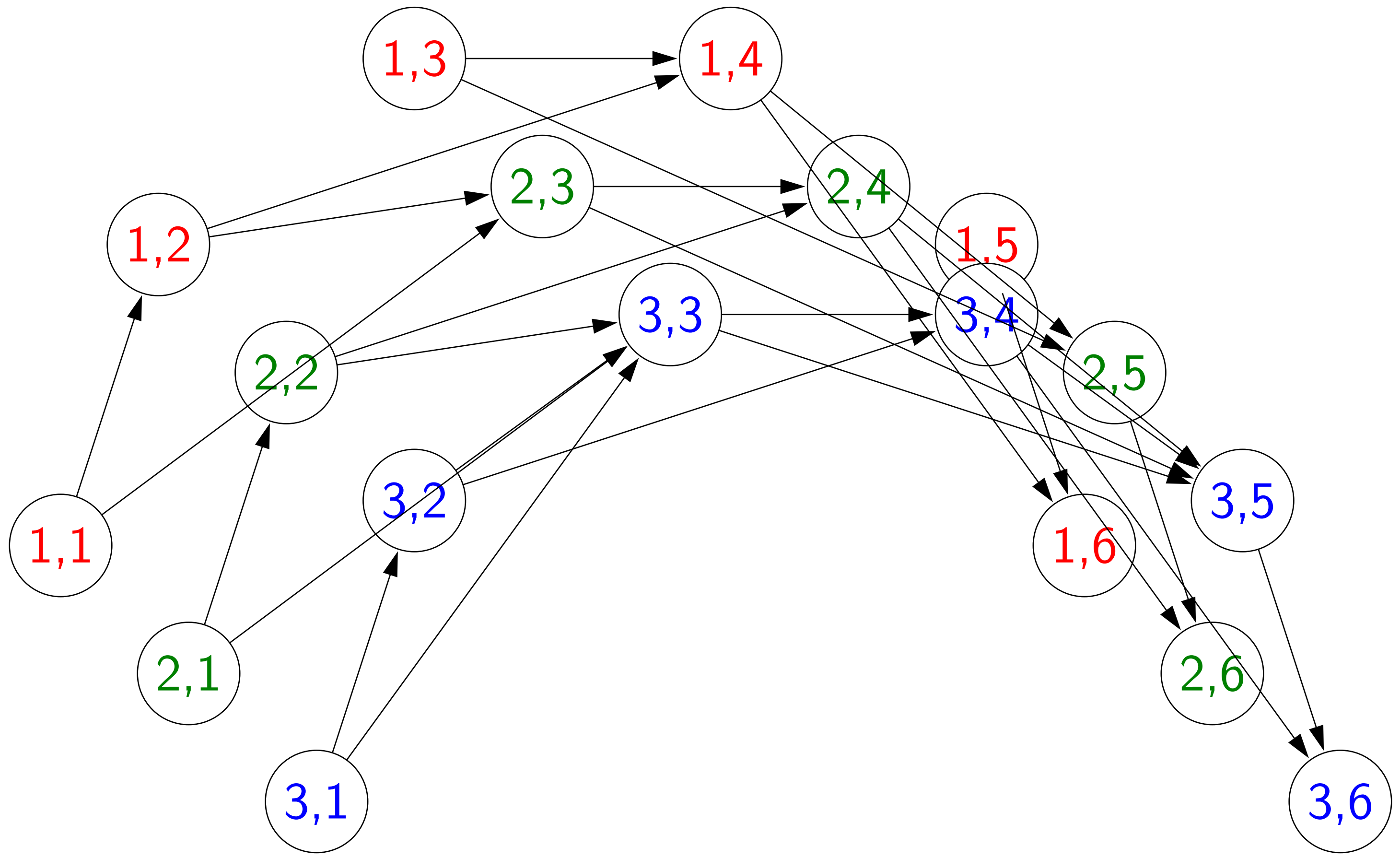
answer in chat

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)



- Our first thought might be to remember how many odd cities we have visited so far (and the current city).
- But if we're more clever, we can notice that once the number of odd cities is 3, we don't need to keep track of whether that number goes up to 4 or 5, etc. So the state we actually need to keep is $(\min(\text{number of odd cities visited}, 3), \text{current city})$. Thus, our state space is $O(n)$ rather than $O(n^2)$.
- We can visualize what augmenting the state does to the state graph. Effectively, we are copying each node 4 times, and the edges are redirected to move between these copies.
- Note that some states such as $(2, 1)$ aren't reachable (if you're in city 1, it's impossible to have visited 2 odd cities already); the algorithm will not touch those states and that's perfectly okay.

answer in chat

Question

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

- An initial guess might be to keep track of the number of even cities and the number of odd cities visited.
- But we can do better. We have to just keep track of the number of odd cities minus the number of even cities and the current city. We can write this more formally as $(n_1 - n_2, \text{current city})$, where n_1 is the number of odd cities visited so far and n_2 is the number of even cities visited so far.

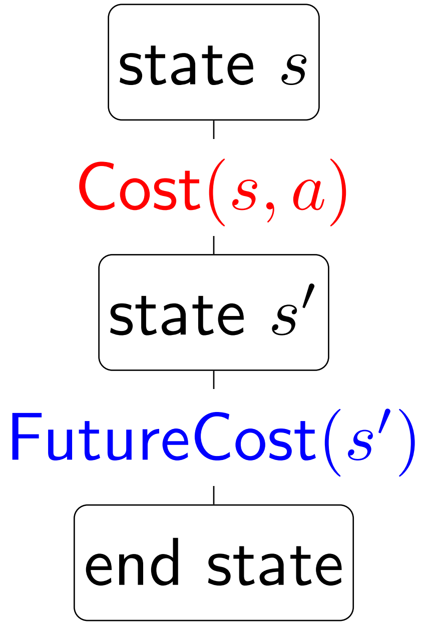
Summary

- 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



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

Key idea: state

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