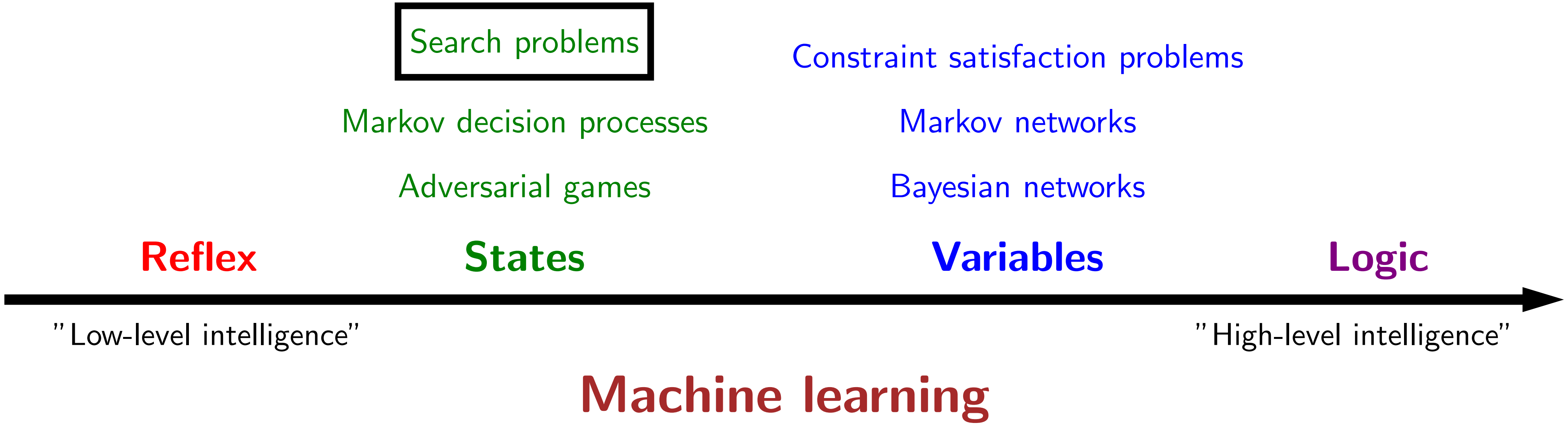
CS221 Section 3: Search

*DP, UCS, A**

Course plan



Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

Markov networks

Bayesian networks

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

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

Machine learning

What are the “ingredients” for a well-defined 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
- $\text{Is}\,\mathbf{End}(s)$: found solution?

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

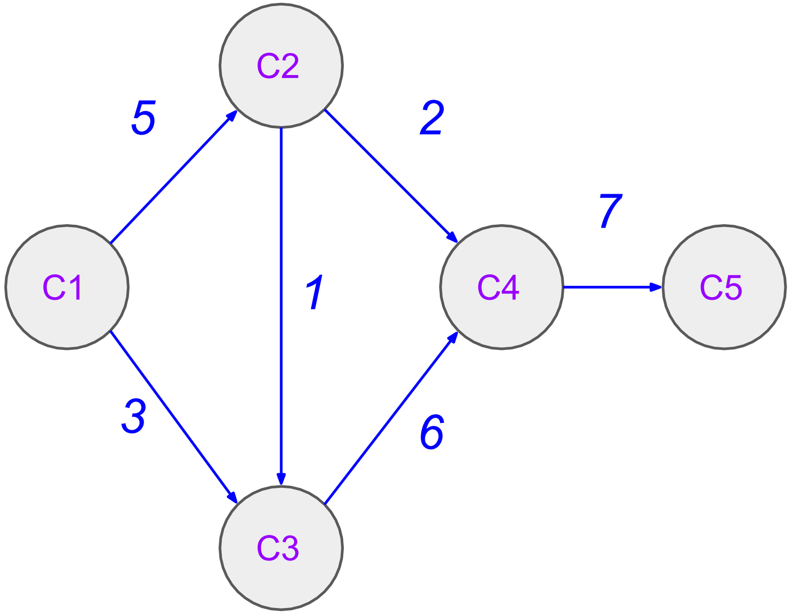
Section Problem

There exists ***N* cities**, labeled from *1* to *N*.

There are one-way roads connecting some pairs of cities. The road connecting city *i* and city *j* takes ***c(i,j)*** time to traverse. However, one can **only travel from a city with smaller label to a city with larger label** (each road is one-directional).

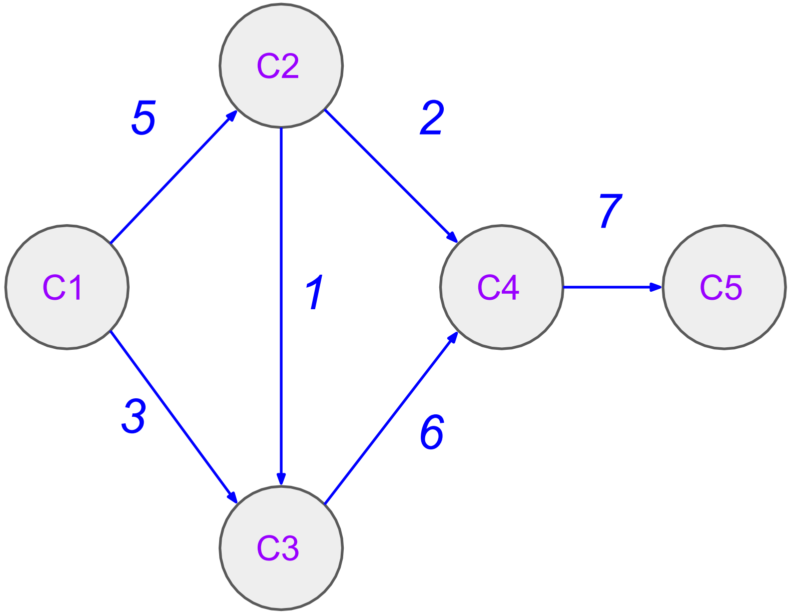
From city *1*, we want to travel to city *N*. What is the **shortest time** required to make this trip, given the **constraint** that we should visit **more odd-labeled cities than even labeled cities**?

Example



1. What is the **shortest path** (without constraint)?
2. What is the **shortest path under the given constraint** (visit more odd than even cities)?

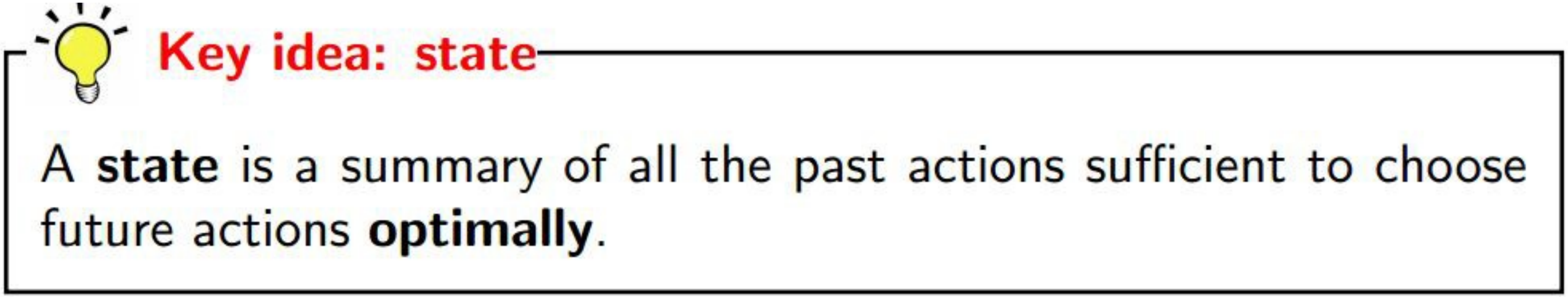
Example



[C1, C2, C4, C5] has cost 14 but visits equal number of odd and even cities.

Best path is [C1, C3, C4, C5] with cost 16.

State Representation



Key idea: state

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

How would you represent a state for this problem?

State Representation

We need to know where we are currently at: **current_city**

We need to know how many odd and even cities we have visited thus far: **#odd, #even**

State Representation: (**current_city, #odd, #even**)

Total number of states: **O(N^3)**

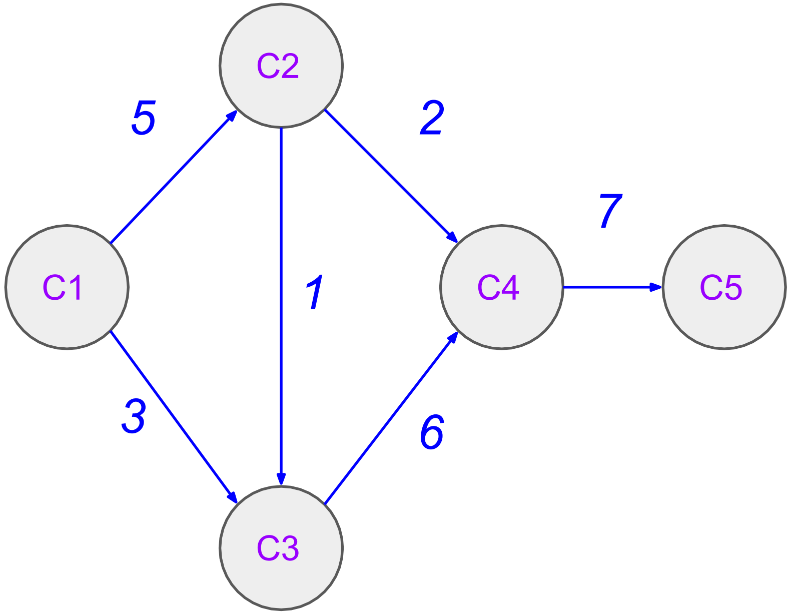
Can We Do Better?

Check if all the information is really required

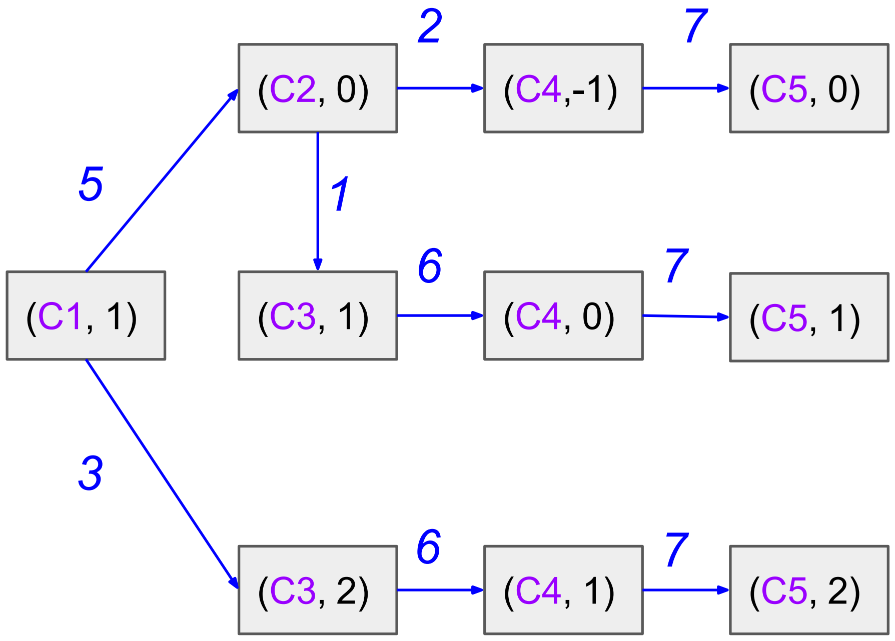
Instead of storing **#odd** and **#even**, we can store **#odd - #even** directly; this still allows us to check whether **#odd** - **#even** > 0 at (N, **#odd**, **#even**)

(**current_city, #odd - #even**) → **O(N^2)** states

Original Graph



State Graph



State $s = (i, d)$ (current city, #odd-#even)

Precise Formulation of Problem

State $s := (i, d)$ (current city, #odd-#even)

$E := \{(i, j) \mid \exists$ road from i to j$\}$

$\text{Actions}(s) := \{move(j) \mid (i, j) \in E\}$

$\text{Cost}(s, move(j)) := c(i, j)$

$$\text{Succ}(s, a) := \begin{cases} (j, d+1) & j \text{ odd} \\ (j, d-1) & j \text{ even} \end{cases}$$

$\text{Start} := (1, 1)$

$\text{isEnd}(s) := i = N$ and $d > 0$

Which algorithms can you use to solve this problem?
Any pros and cons?

Solving the Problem

Since we are computing shortest path, which is some form of optimization, we consider **DP** and **UCS**.

Recall:

- **DP** can handle negative edges but works only on DAGs
- **UCS** works on general graphs, but cannot handle negative edges

➢ *Which one works for our problem?*

Solving the Problem

Since we are computing shortest path, which is some form of optimization, we consider **DP** and **UCS**.

Recall:

- **DP** can handle negative edges but works only on DAGs
- **UCS** works on general graphs, but cannot handle negative edges

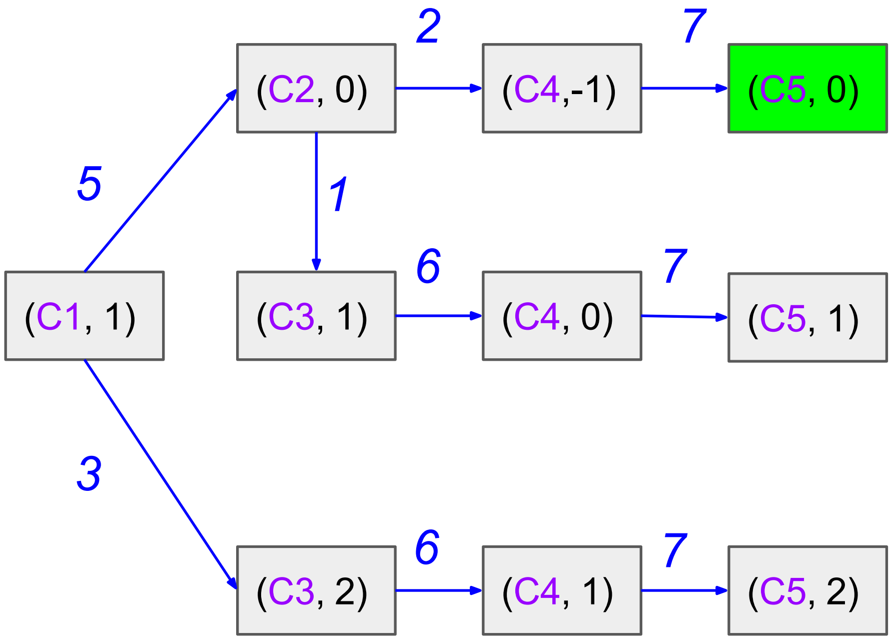
Since we have a **DAG** and all edges are positive, both work!

Solving the Problem: Dynamic Programming

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

If *s* has no successors, we set it as *undefined*

Simulation of DP



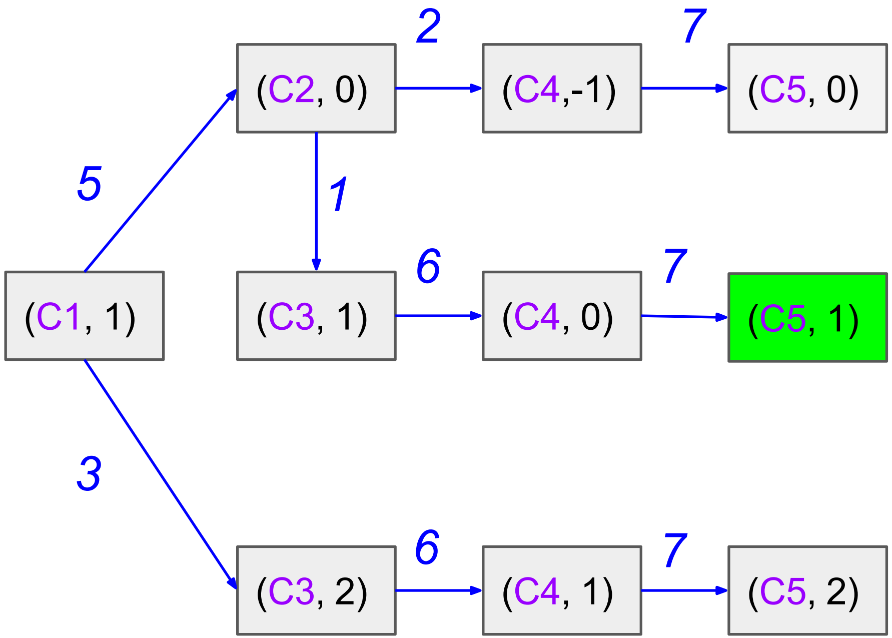
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	-	-	-	-
C3	-	-	-	-	-
C4	-	-	-	-	-
C5	-	?	-	-	-

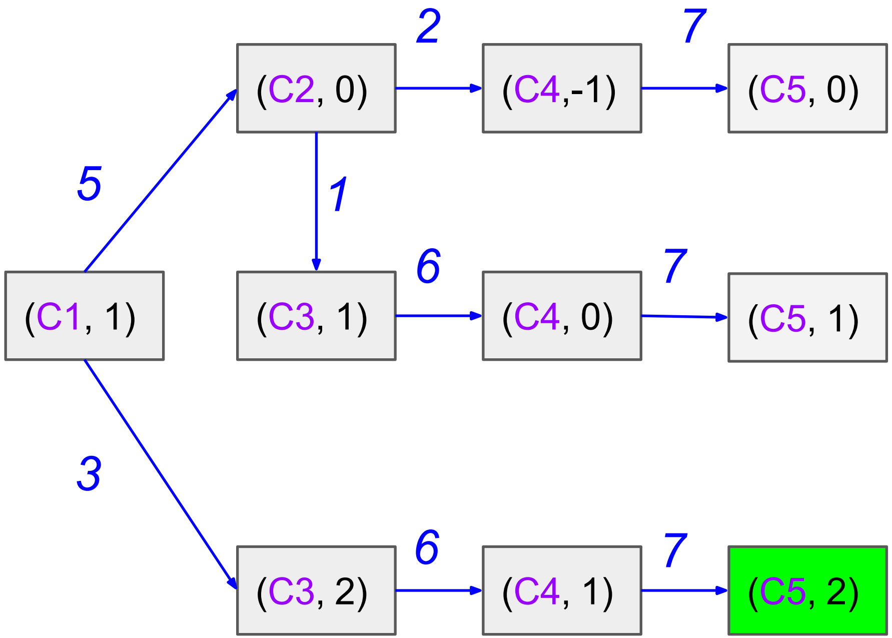
Simulation of DP



State $s = (i, d)$ (current city, #odd-#even)

		#odd - #even				
city		-1	0	1	2	3
	C1	-	-	-	-	-
	C2	-	-	-	-	-
	C3	-	-	-	-	-
	C4	-	-	-	-	-
	C5	-	?	0	-	-

Simulation of DP



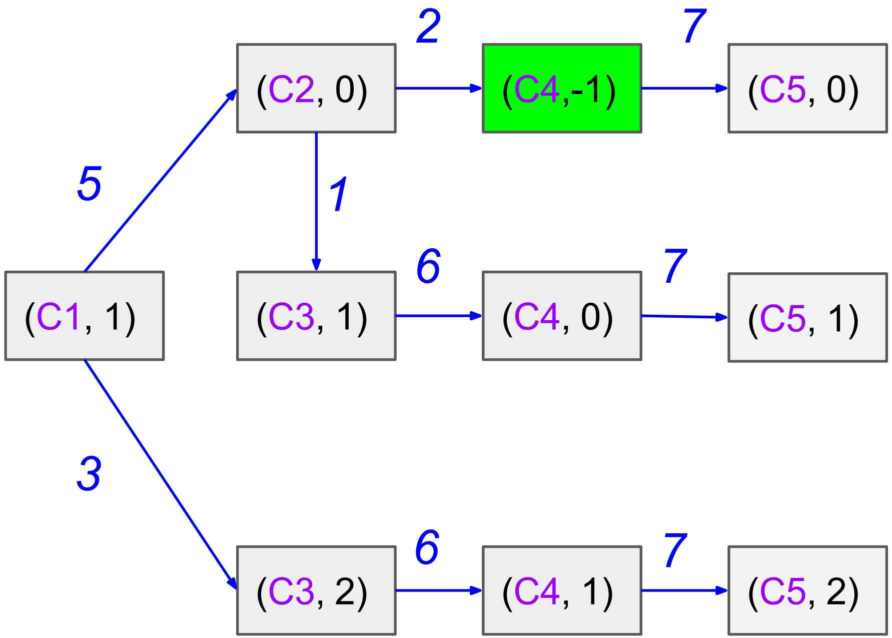
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	-	-	-	-
C3	-	-	-	-	-
C4	-	-	-	-	-
C5	-	?	0	0	-

Simulation of DP



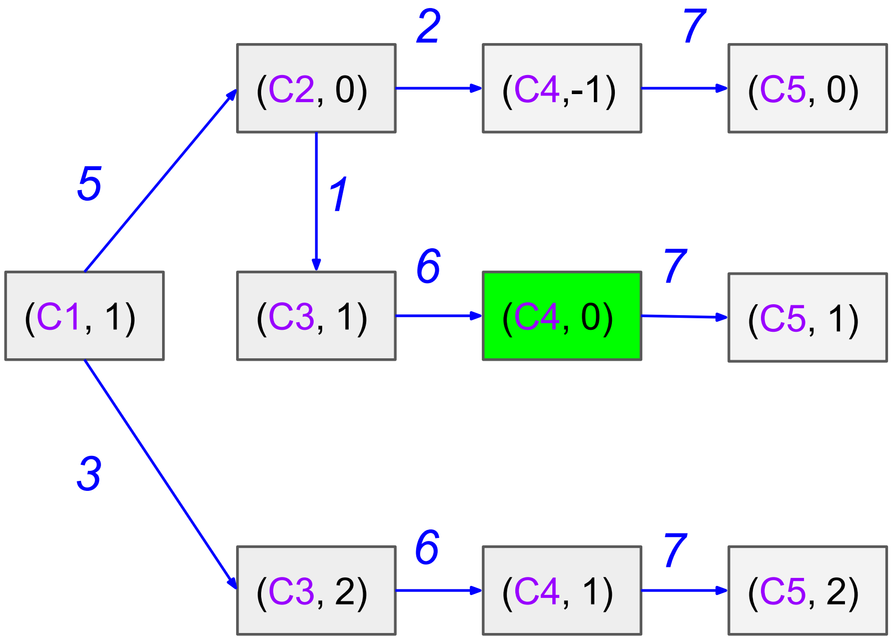
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	-	-	-	-
C3	-	-	-	-	-
C4	?	-	-	-	-
C5	-	?	0	0	-

Simulation of DP



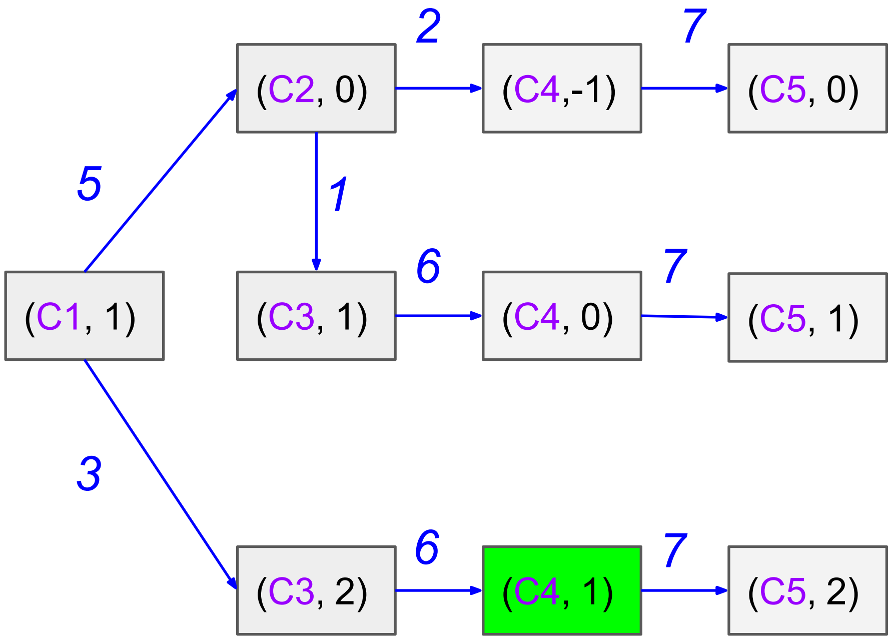
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	-	-	-	-
C3	-	-	-	-	-
C4	?	7	-	-	-
C5	-	?	0	0	-

Simulation of DP



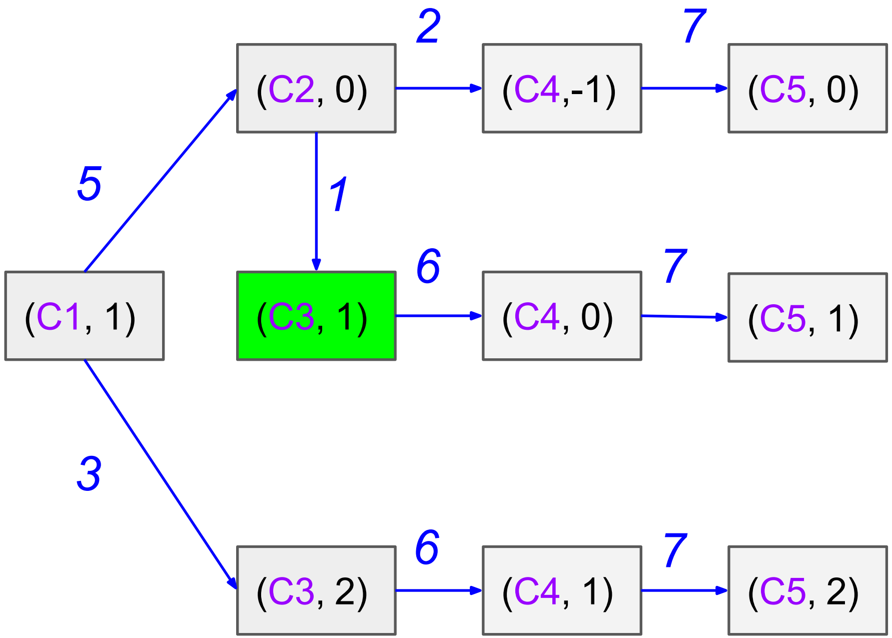
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	-	-	-	-
C3	-	-	-	-	-
C4	?	7	7	-	-
C5	-	?	0	0	-

Simulation of DP



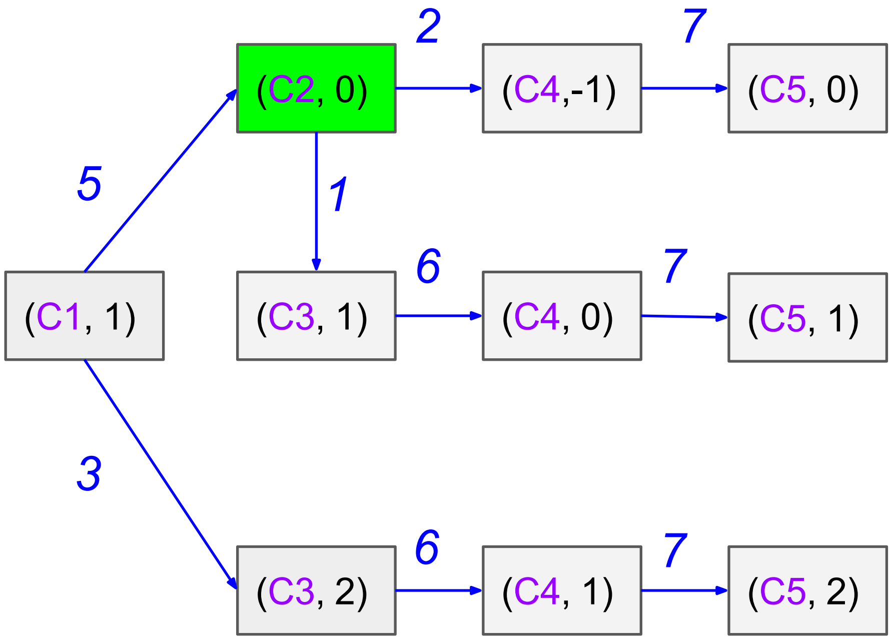
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	-	-	-	-
C3	-	-	13	-	-
C4	?	7	7	-	-
C5	-	?	0	0	-

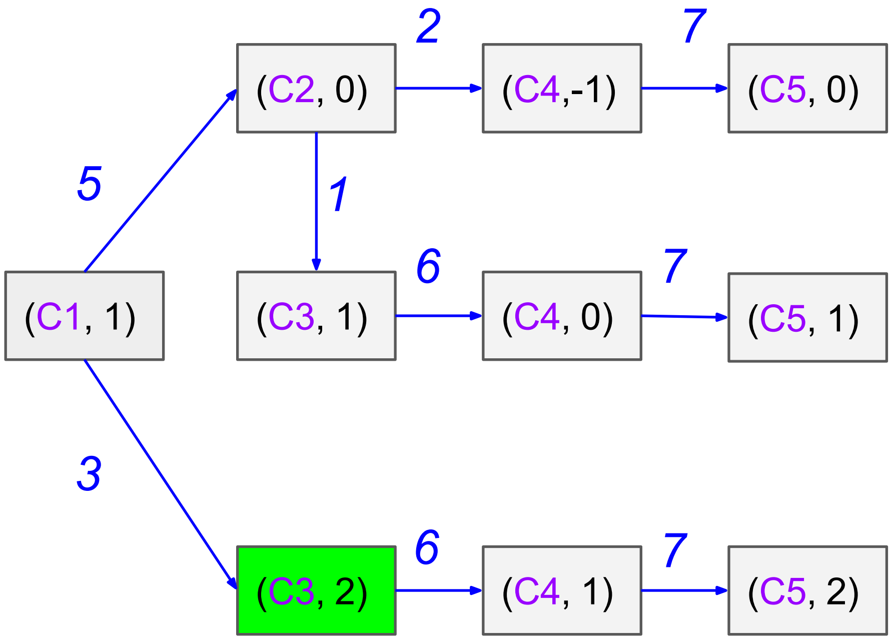
Simulation of DP



State $s = (i, d)$ (current city, #odd-#even)

	#odd - #even				
city	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	14	-	-	-
C3	-	-	13	-	-
C4	?	7	7	-	-
C5	-	?	0	0	-

Simulation of DP



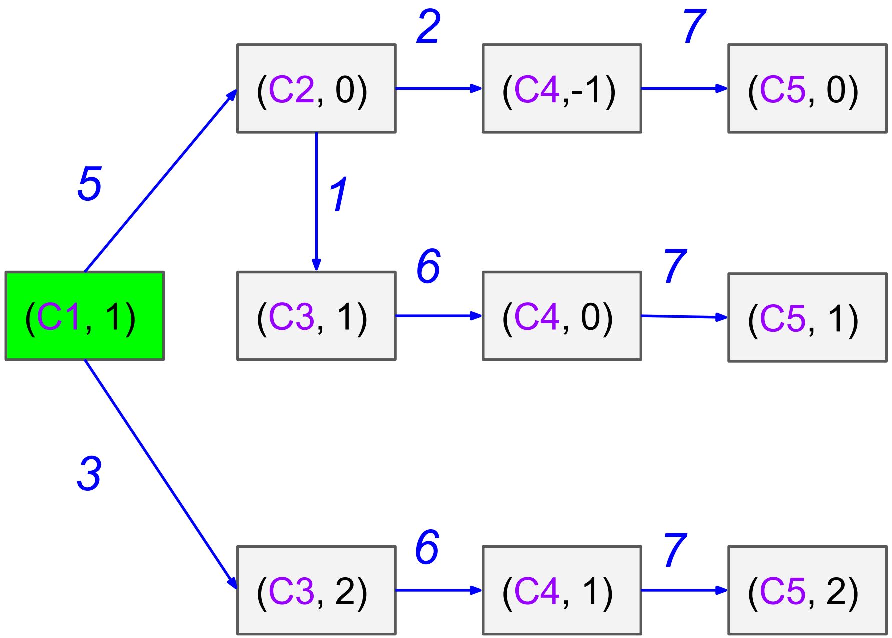
State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	-	-	-
C2	-	14	-	-	-
C3	-	-	13	13	-
C4	?	7	7	-	-
C5	-	?	0	0	-

Simulation of DP



State $s = (i, d)$ (current city, #odd-#even)

#odd - #even

city

	-1	0	1	2	3
C1	-	-	16	-	-
C2	-	14	-	-	-
C3	-	-	13	13	-
C4	?	7	7	-	-
C5	-	?	0	0	-

From depth to breadth first search

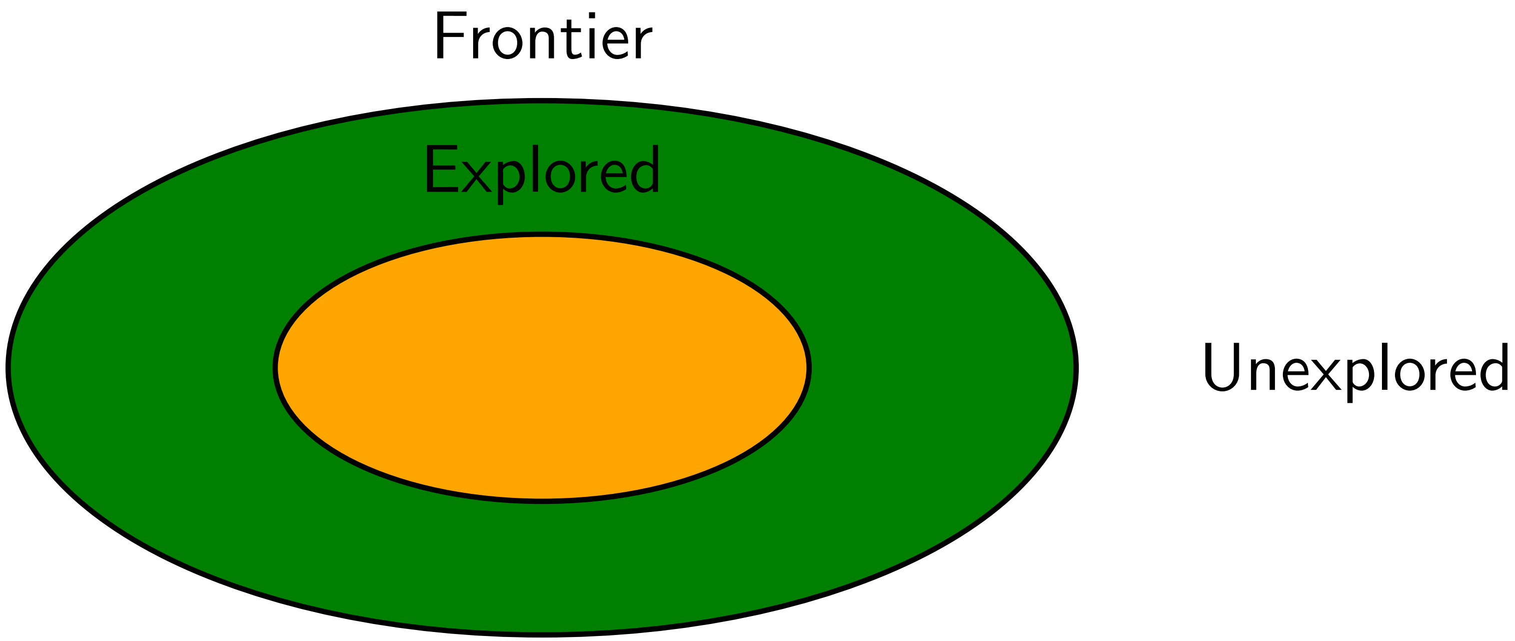
Our earlier DP algorithm: improved exhaustive search

- Go down the tree by taking actions
- Use FutureCost to re-use computation

Up next: improved breadth first search

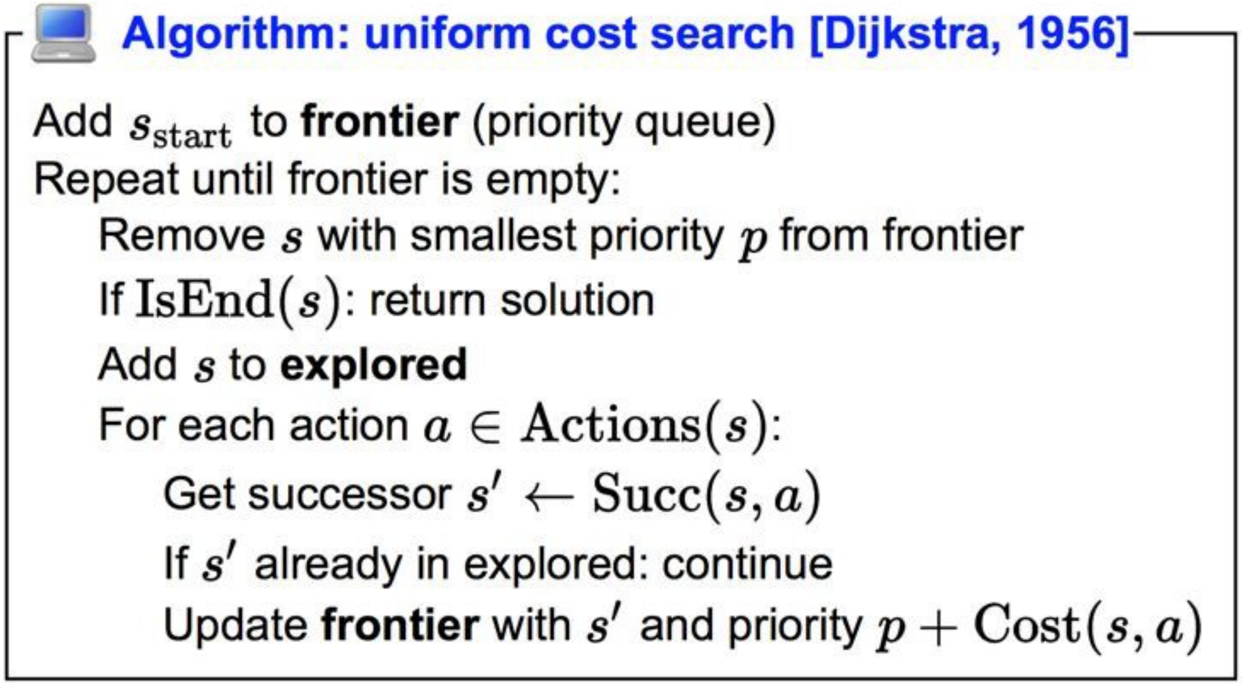
- Expand states close to the start (breadth)
- Use PastCost to re-use computation

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

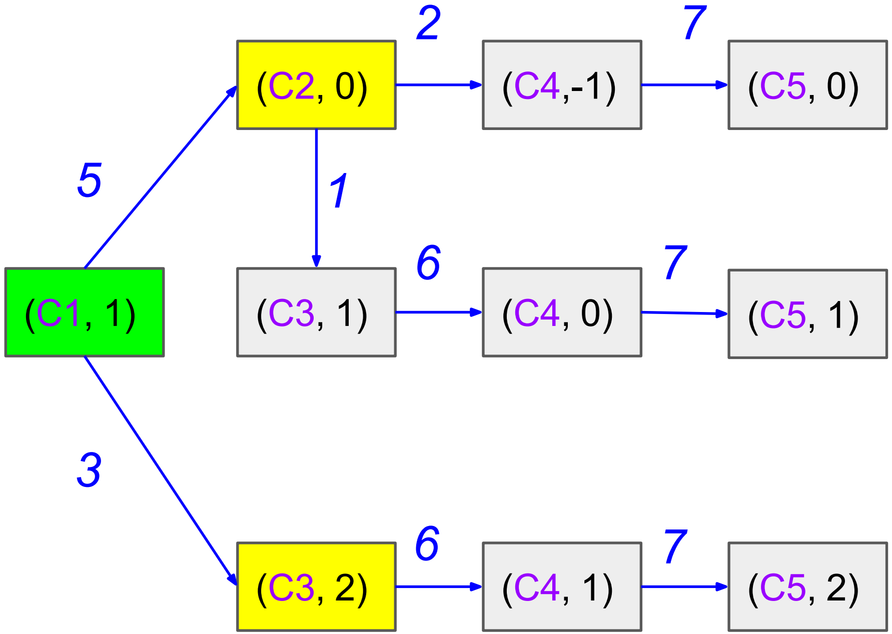
Solving the Problem: Uniform Cost Search



Algorithm: uniform cost search [Dijkstra, 1956]

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

Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

Explored:

(C1, 1) : 0

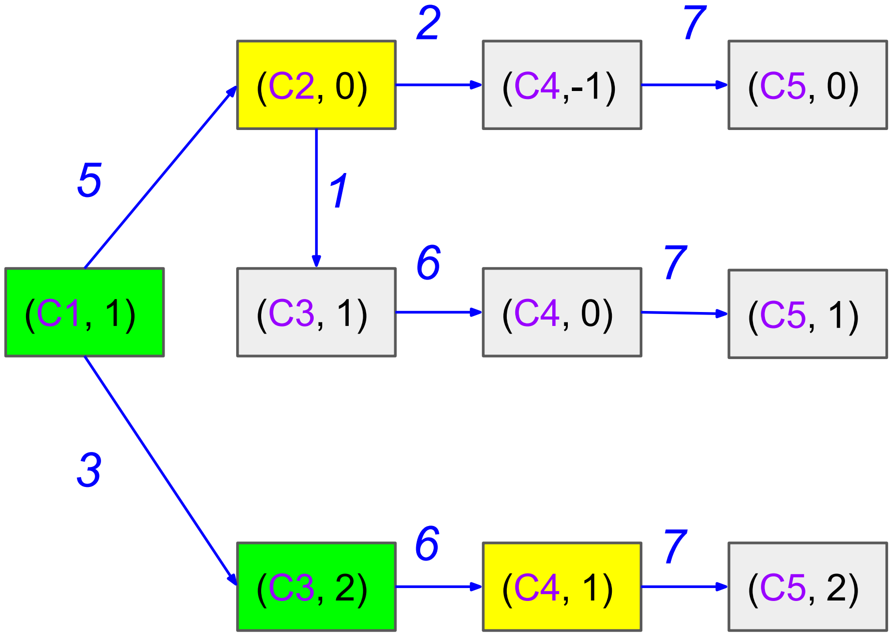
Frontier:

(C3, 2) : 3

(C2, 0) : 5

→ Frontier is a priority queue.

Simulation of UCS

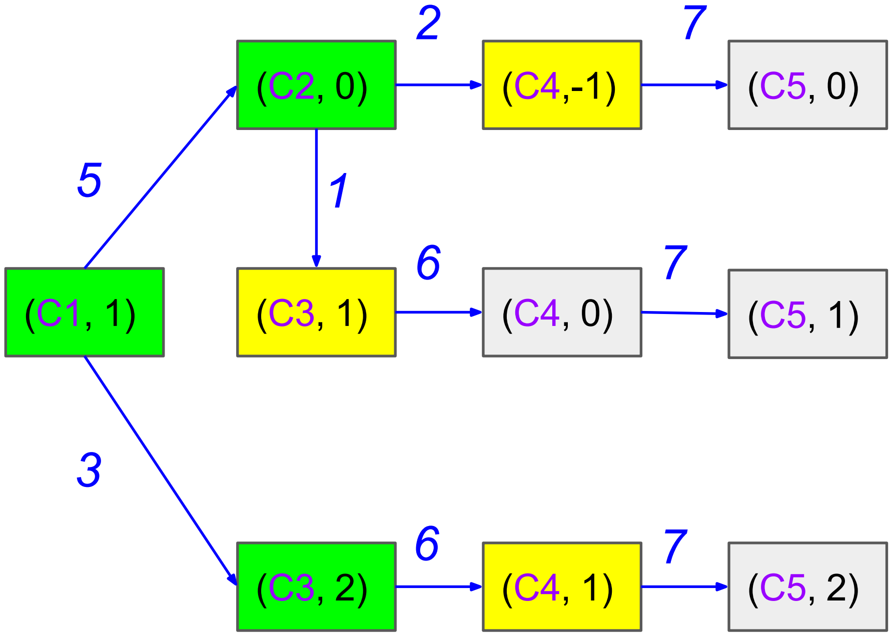


State $s = (i, d)$ (current city, #odd-#even)

Explored:
(C1, 1) : 0
(C3, 2) : 3

Frontier:
(C2, 0) : 5
(C4, 1) : 9

Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

Explored:

(C1, 1) : 0

(C3, 2) : 3

(C2, 0) : 5

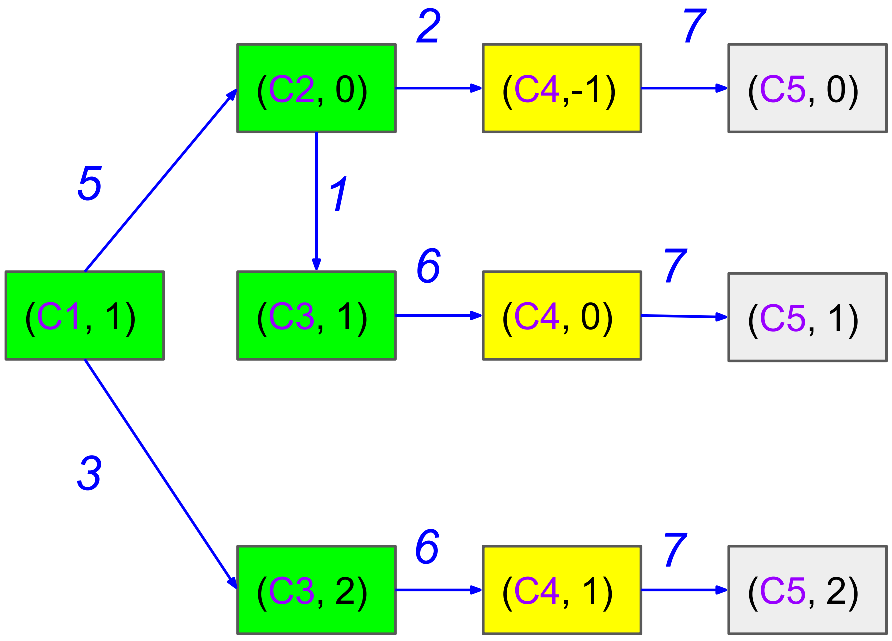
Frontier:

(C3, 1) : 6

(C4, -1) : 7

(C4, 1) : 9

Simulation of UCS

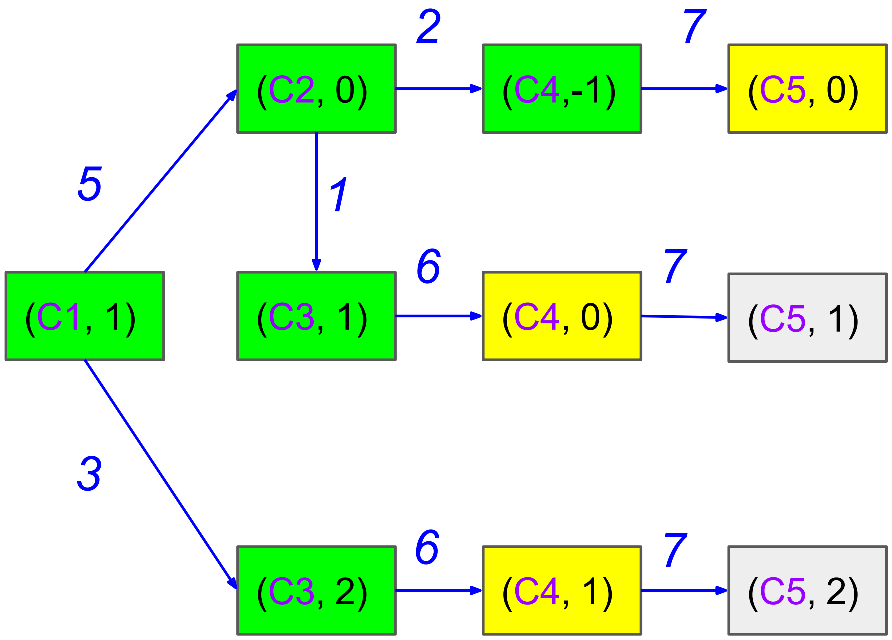


State $s = (i, d)$ (current city, #odd-#even)

Explored:
(C1, 1) : 0
(C3, 2) : 3
(C2, 0) : 5
(C3, 1): 6

Frontier:
(C4, -1) : 7
(C4, 1) : 9
(C4, 0): 12

Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

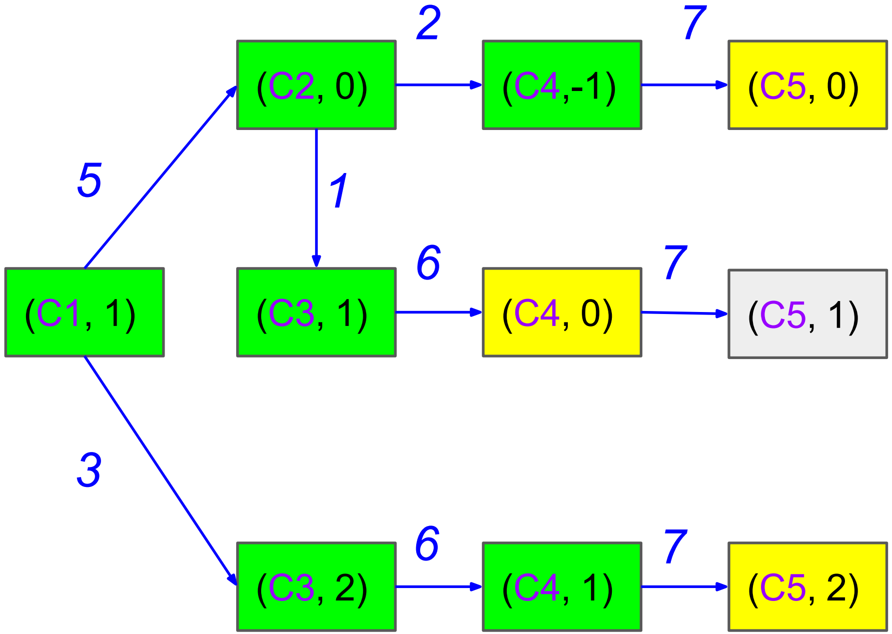
Explored:

(C1, 1) : 0
(C3, 2) : 3
(C2, 0) : 5
(C3, 1) : 6
(C4, -1) : 7

Frontier:

(C4, 1) : 9
(C4, 0) : 12
(C5, 0) : 14

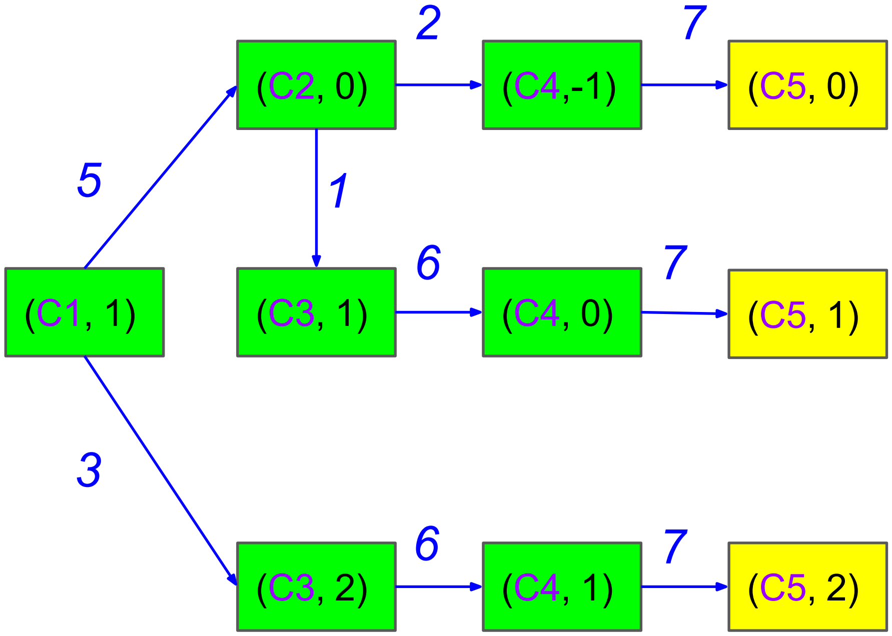
Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

Explored:	Frontier:
(C1, 1) : 0	(C4, 0) : 12
(C3, 2) : 3	(C5, 0) : 14
(C2, 0) : 5	(C5, 2) : 16
(C3, 1) : 6	
(C4, -1) : 7	
(C4, 1) : 9	

Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

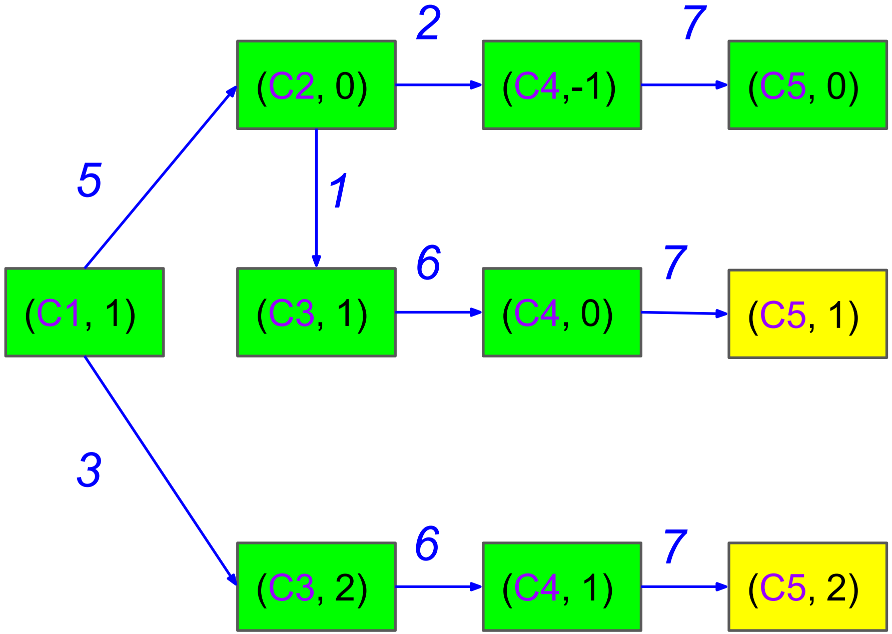
Explored:

(C1, 1) : 0
(C3, 2) : 3
(C2, 0) : 5
(C3, 1) : 6
(C4, -1) : 7
(C4, 1) : 9
(C4, 0) : 12

Frontier:

(C5, 0) : 14
(C5, 2) : 16
(C5, 1) : 19

Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

Explored:

(C1, 1) : 0

(C3, 2) : 3

(C2, 0) : 5

(C3, 1) : 6

(C4, -1) : 7

(C4, 1) : 9

(C4, 0) : 12

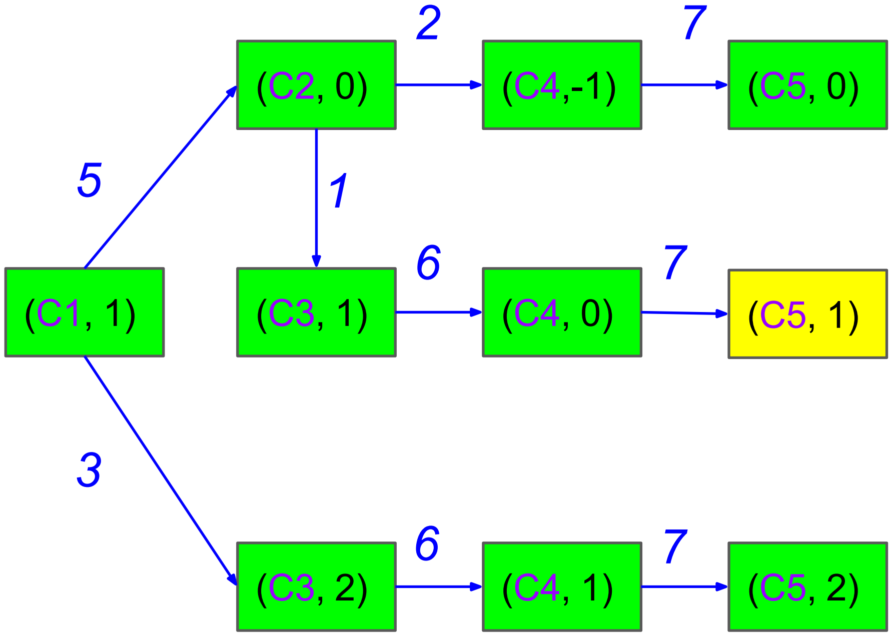
(C5, 0) : 14

Frontier:

(C5, 2) : 16

(C5, 1) : 19

Simulation of UCS



State $s = (i, d)$ (current city, #odd-#even)

Explored:

(C1, 1) : 0
(C3, 2) : 3
(C2, 0) : 5
(C3, 1) : 6
(C4, -1) : 7
(C4, 1) : 9
(C4, 0) : 12
(C5, 0) : 14
(C5, 2) : 16

Frontier:

(C5, 1) : 19

STOP!

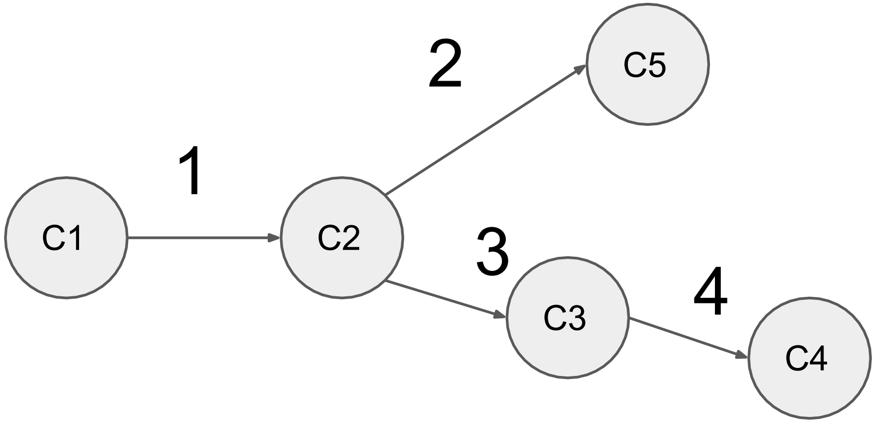
(Since we found C5 with #odd-#even > 0)

Comparison between DP and UCS

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

Runtime of DP is O(N)

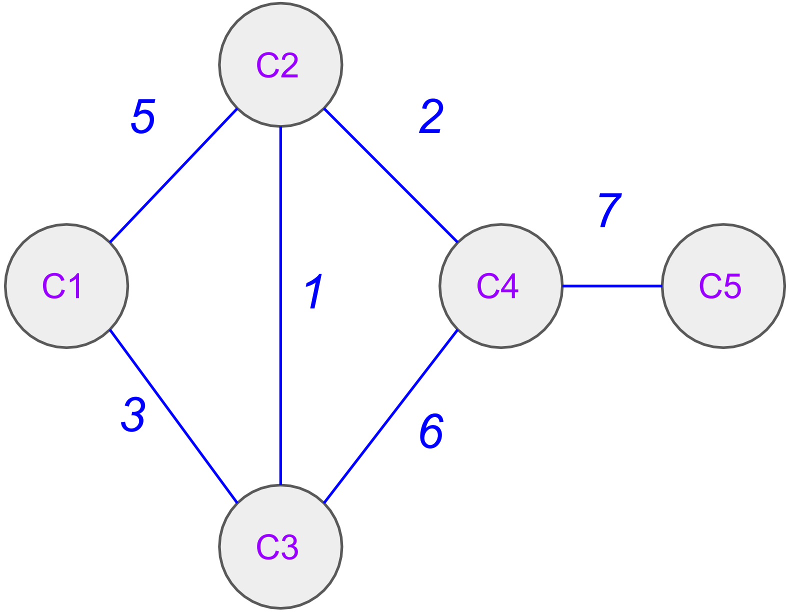
Runtime of UCS is O(n log n)



Example:
Start state C1, end state C5

-DP explores O(N) states.
-UCS will explore {C1, C2, C5} only. C3 will be in the frontier and C4 will be unexplored.

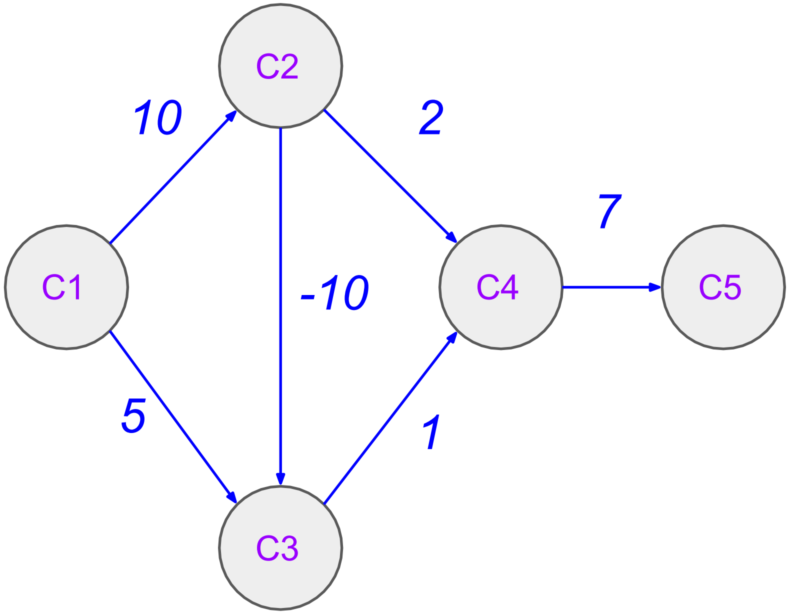
DP cannot handle cycles



Shortest path is [C1, C3, C2, C5] with cost 13.

Hard to define subproblems in undirected or cyclic graphs.

UCS cannot handle negative edge weights



Best path is [C1,C2,C3,C4,C5] with cost of 8, but UCS will output [C1,C3,C4,C5] with cost of 13 because C3 is marked as ‘explored’ before C2.

Back to our section problem,
can we do the search faster than UCS?

Use A!*

https://qiao.github.io/PathFinding.js/visual/