Section 2: Search

Richard & Nick

Motivation

A lot of real-word problems can be modeled as search problems where states have actions that lead to successor states.

- Text segmentation and Vowel insertion
  - Where to put whitespace and vowels, which vowels
- Picking flights to from source city to destination
  - Optimize Time, or Cost, or Layovers
- Structured prediction - e.g. Part-of-Speech, Dependency Parsing
  - Globally Normalized Transition-Based Neural Networks (D. Andor et. al, https://www.aclweb.org/anthology/P16-1231)
  - Global Neural CCG Parsing with Optimality Guarantees (K. Lee et. al, https://arxiv.org/abs/1607.01432)

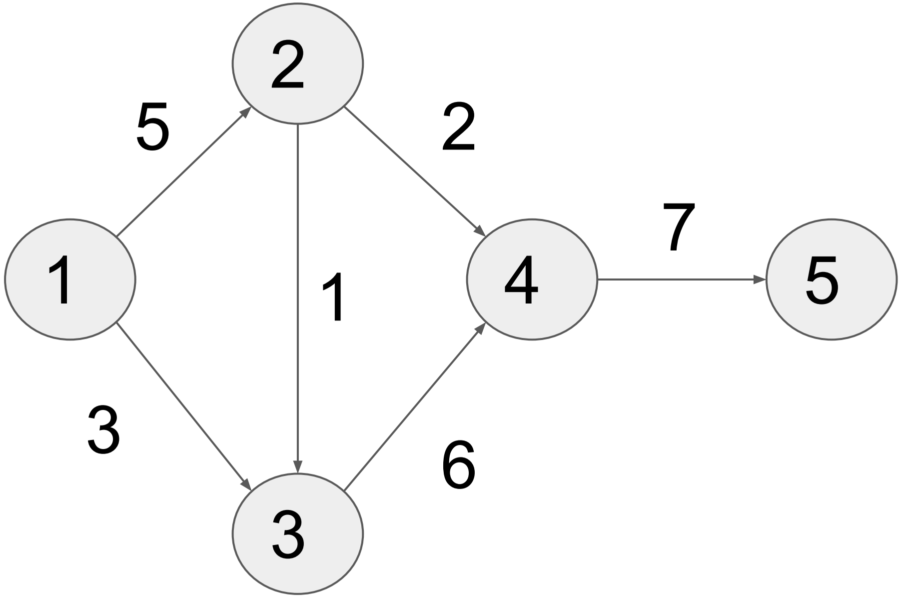
Sample Problem

There exists *N* cities, conveniently labelled from *1* to *N*.

There are 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 (i.e. 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 additional constraint that we should **visit more odd-labeled cities than even labeled cities**?

Example



Best path is [1, 3, 4, 5] with cost 16.

[1, 2, 4, 5] has cost 14 but visits equal number of odd and even cities.

State Representation

Key idea: state

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

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

We store **#odd** and **#even** so that we can check whether **#odd** - **#even** > 0 at (N, **#odd**, **#even**)

Why not store **#odd - #even** directly instead?

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

Precise Formulation of Problem

Let E be the set of one-way roads between cities. $(i, j) \in E$ means there's a road from city i to city j.

State s = (i, d) -- (current_city, #odd - #even)

Start: (1, 1)

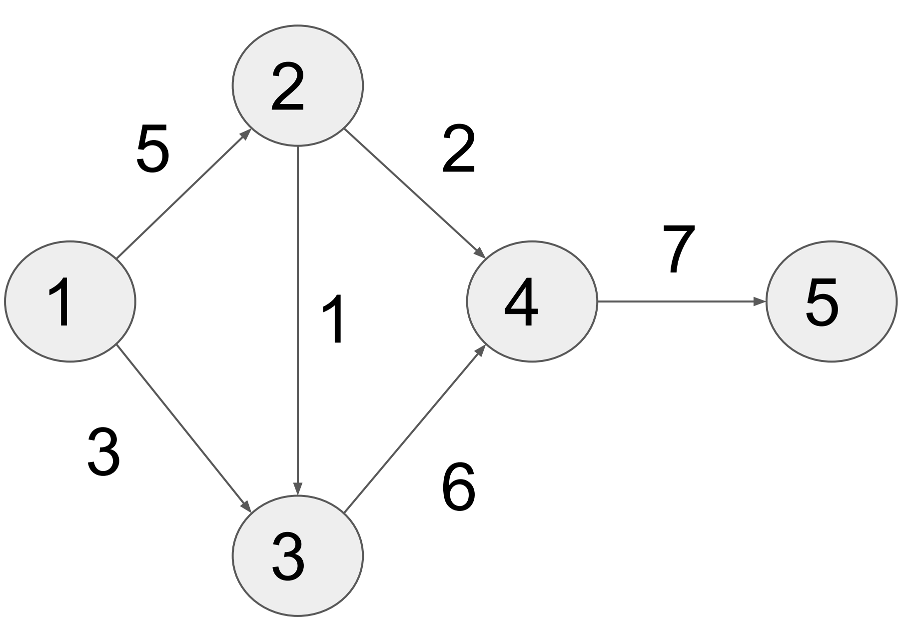
isGoal(s): $i = N$ and $d > 0$

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

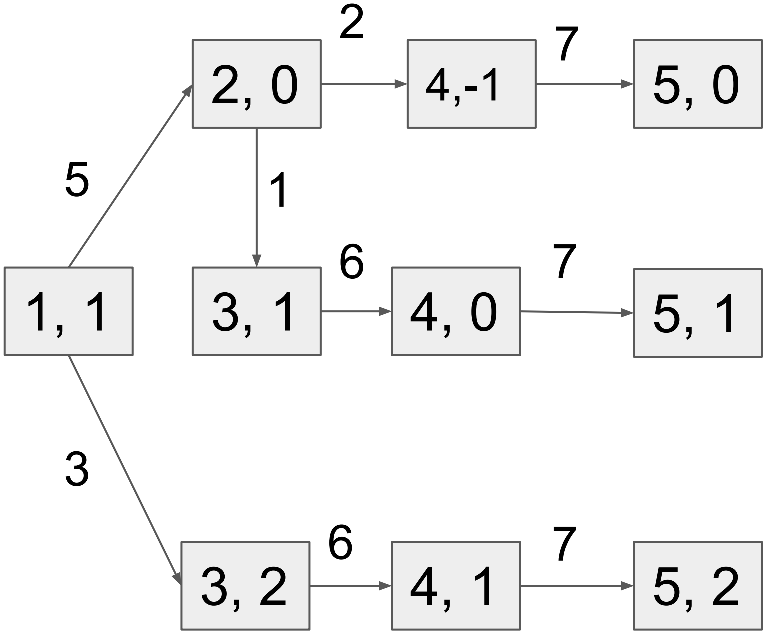
Cost(s, move(j)): c(i,j)

Succ(s, move(j)): (j, d + 1) if j is odd, (j, d - 1) otherwise

Example



State Graph



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

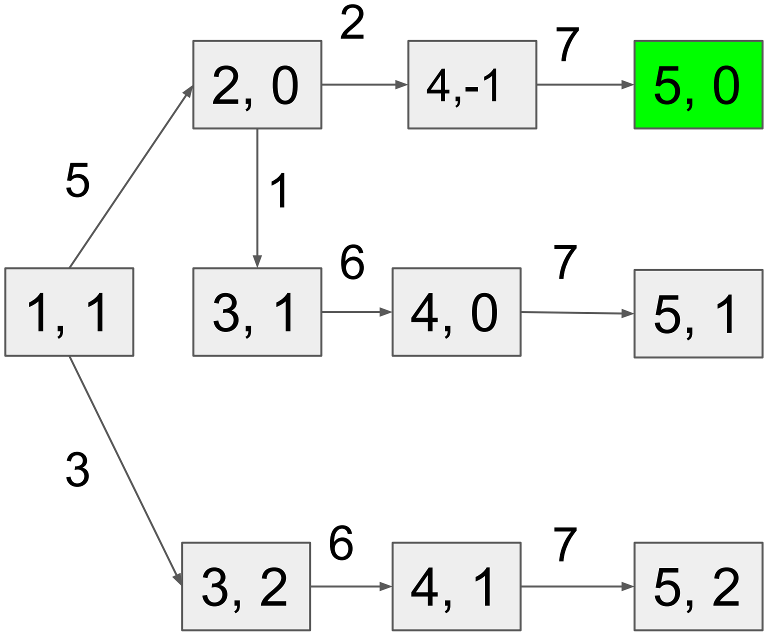
Since we have a DAG and all edges are positive, both work!

Solving the Problem: Dynamic Programming

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

- FutureCost = cost of all future actions
- If state s has no successors, FutureCost(s) is undefined

Simulation of DP

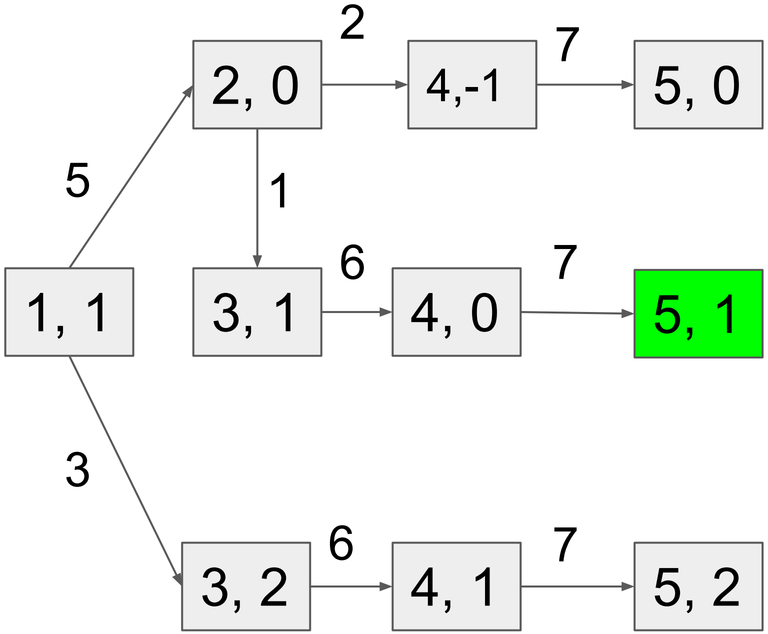


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	-	-
5	-	?	-	-	-

Simulation of DP

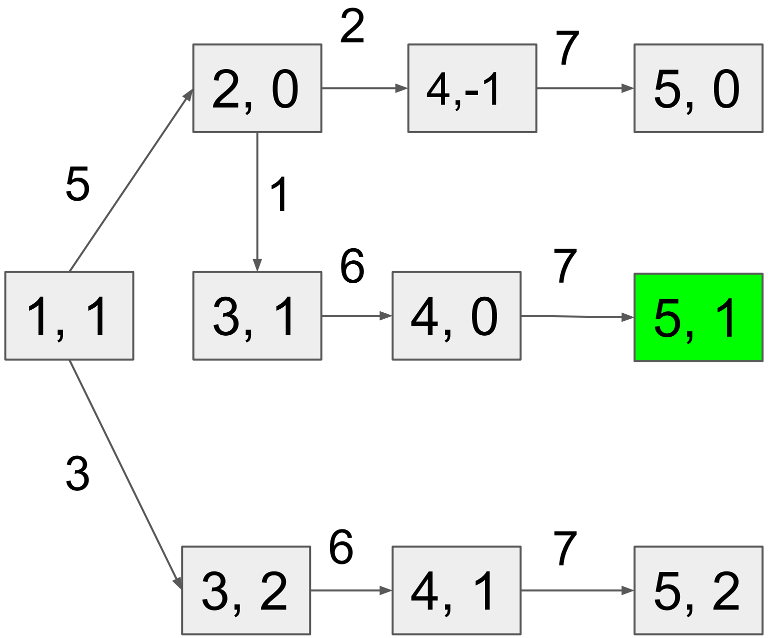


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	-	-
5	-	?		-	-

Simulation of DP

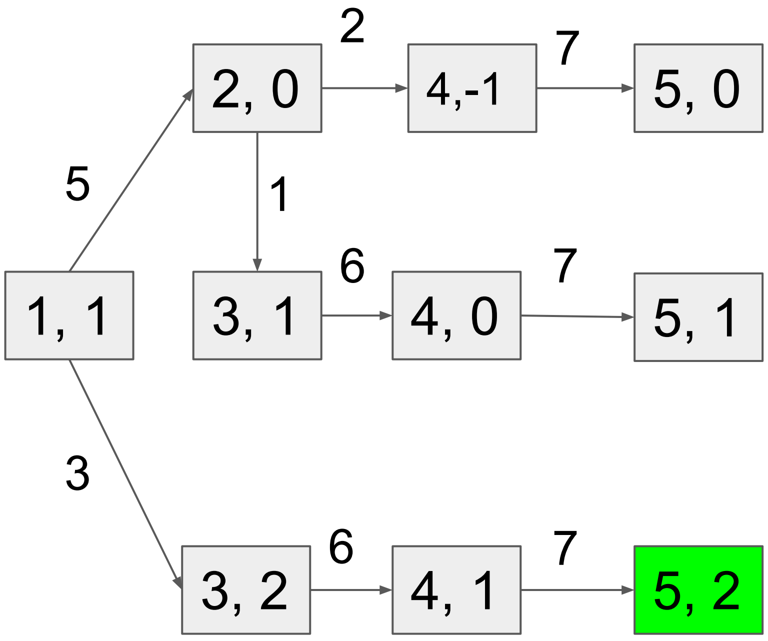


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	-	-
5	-	?	0	-	-

Simulation of DP

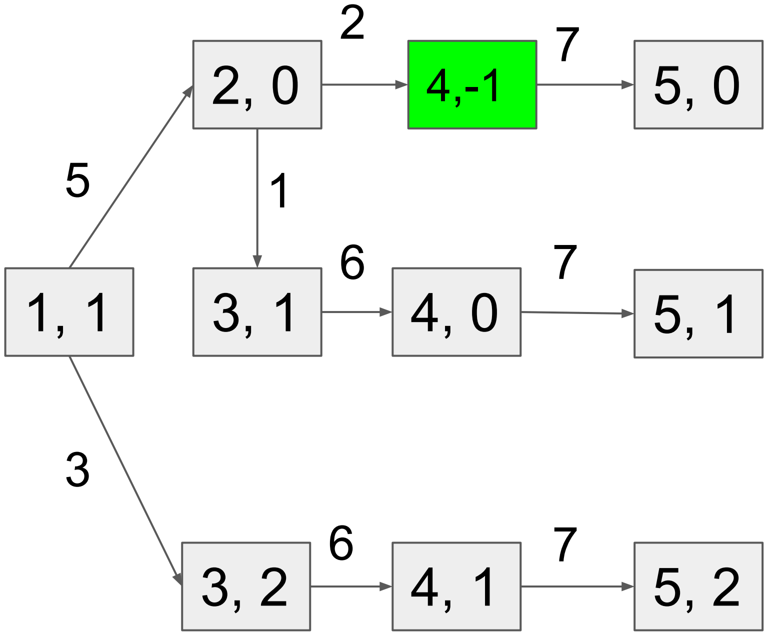


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	-	-
5	-	?	0	0	-

Simulation of DP

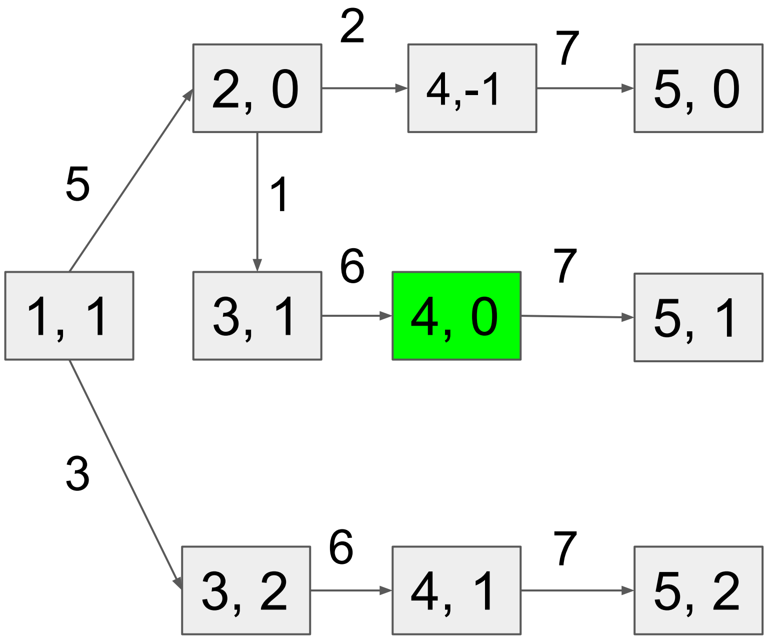


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	?	-	-	-	-
5	-	?	0	0	-

Simulation of DP

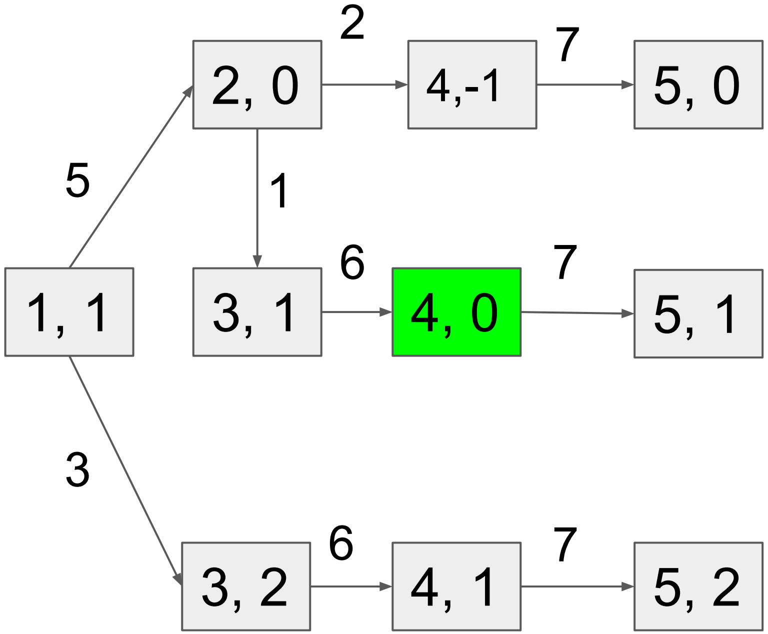


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	?		-	-	-
5	-	?	0	0	-

Simulation of DP

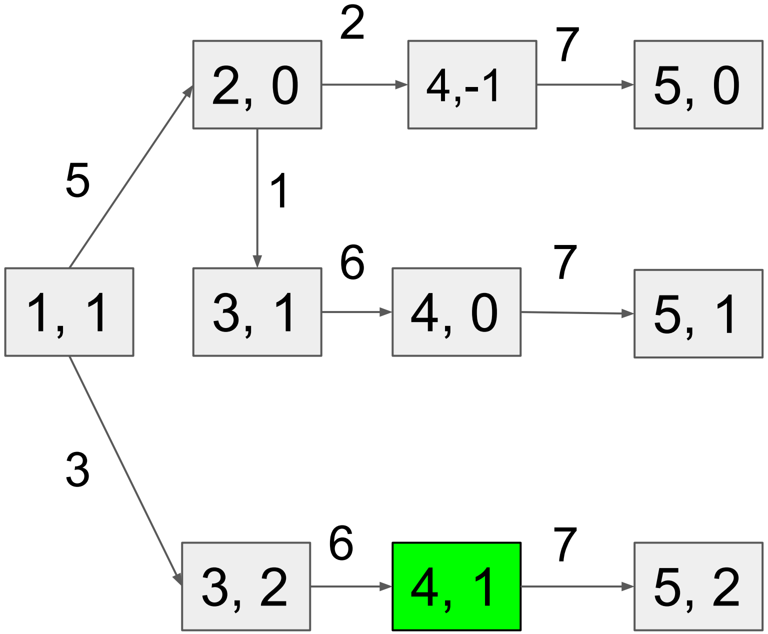


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	?	7	-	-	-
5	-	?	0	0	-

Simulation of DP

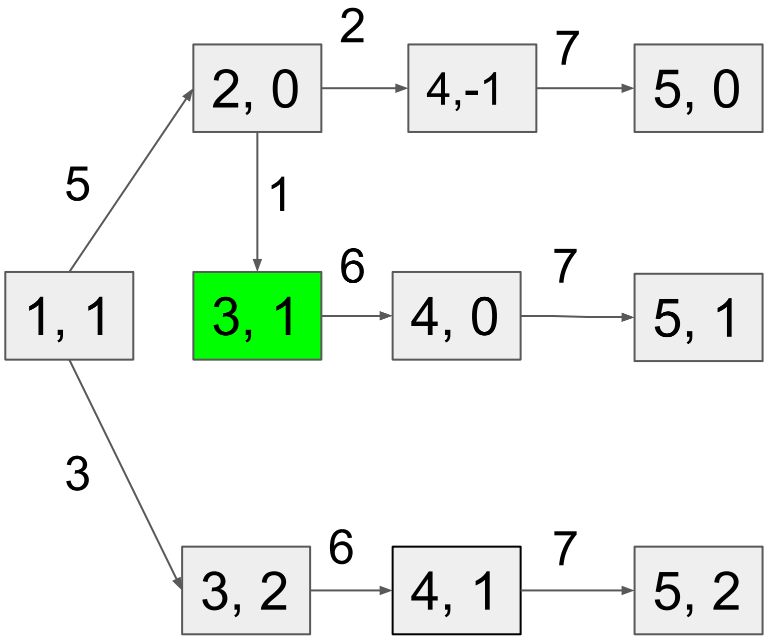


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	?	7	7	-	-
5	-	?	0	0	-

Simulation of DP

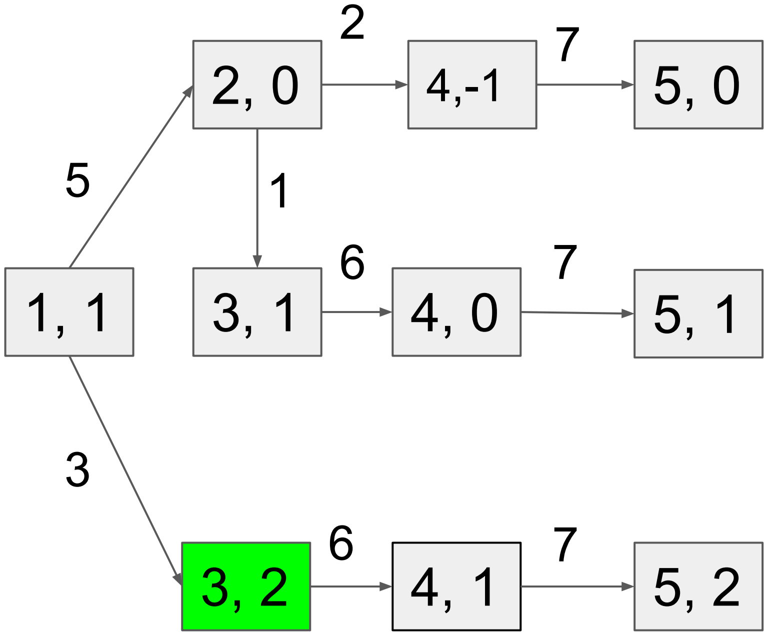


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	13	-	-
4	?	7	7	-	-
5	-	?	0	0	-

Simulation of DP

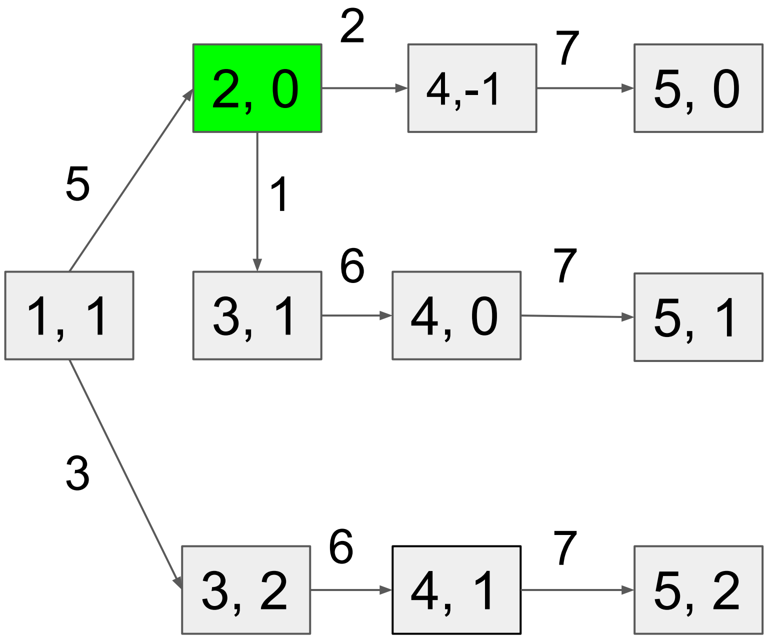


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	13	13	-
4	?	7	7	-	-
5	-	?	0	0	-

Simulation of DP

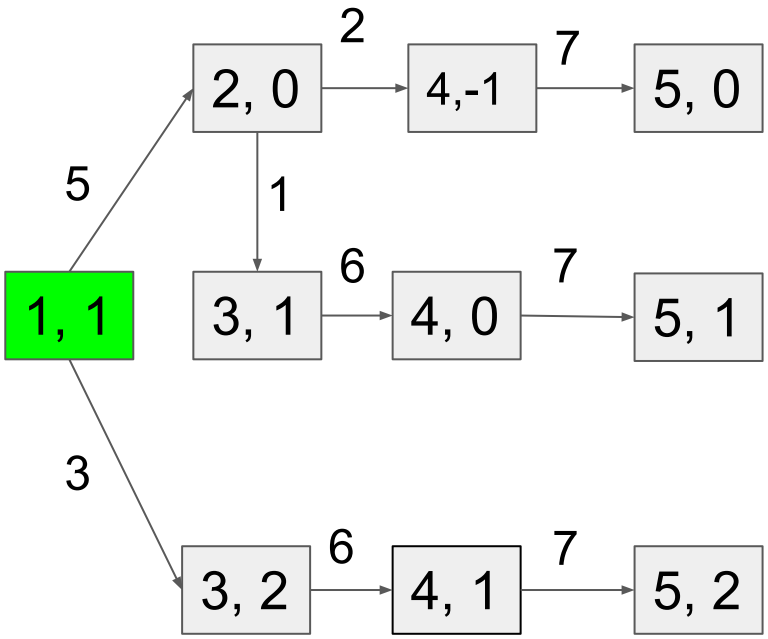


#odd - #even

city

	-1	0	1	2	3
1	-	-	-	-	-
2	-	14	-	-	-
3	-	-	13	13	-
4	?	7	7	-	-
5	-	?	0	0	-

Simulation of DP



#odd - #even

city

	-1	0	1	2	3
1	-	-	16	-	-
2	-	14	-	-	-
3	-	-	13	13	-
4	?	7	7	-	-
5	-	?	0	0	-

Solving the Problem: Uniform Cost Search

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 IsGoal(s): return solution

  Add s to **explored**

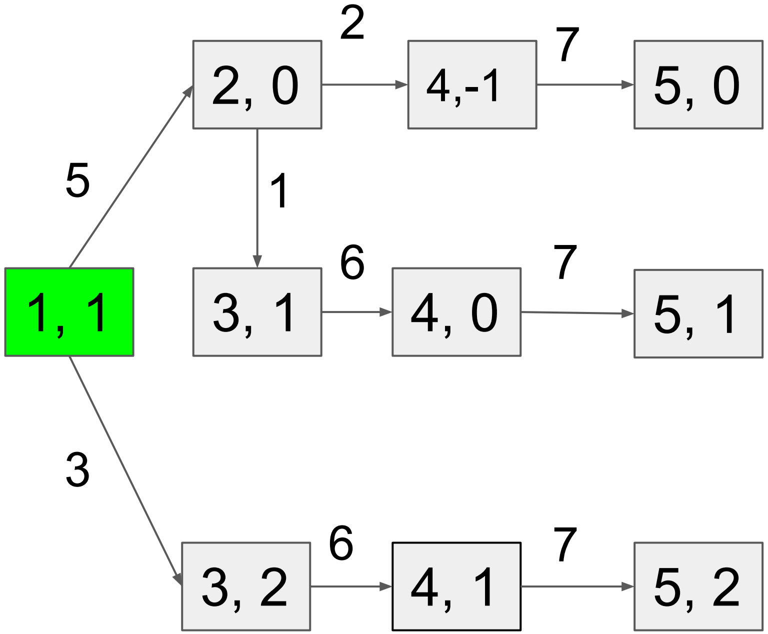
  For each action $a \in$ Actions(s):

    Get successor $s' \leftarrow$ Succ(s, a)

    If s' already in explored: continue

    Update **frontier** with s' and priority $p +$ Cost(s, a)

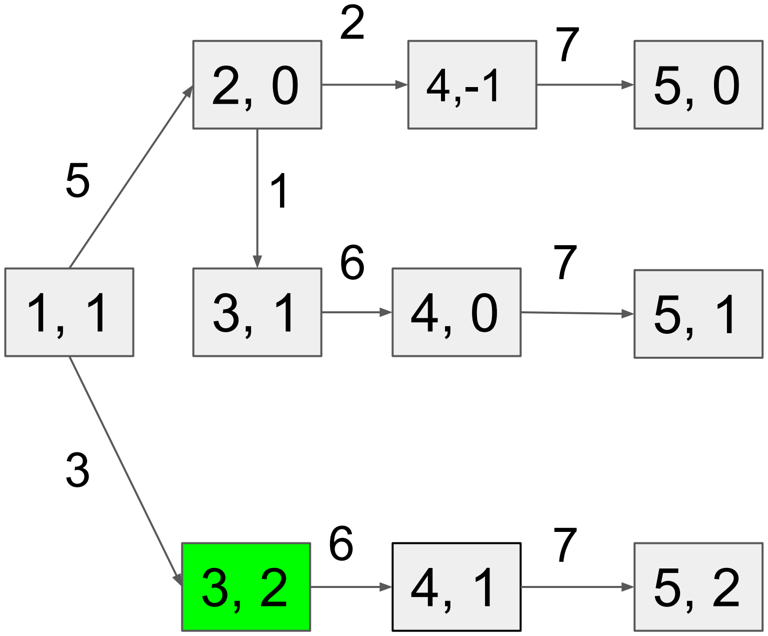
Simulation of UCS



Explored:
(1, 1) : 0

Frontier:
(3, 2) : 3
(2, 0) : 5

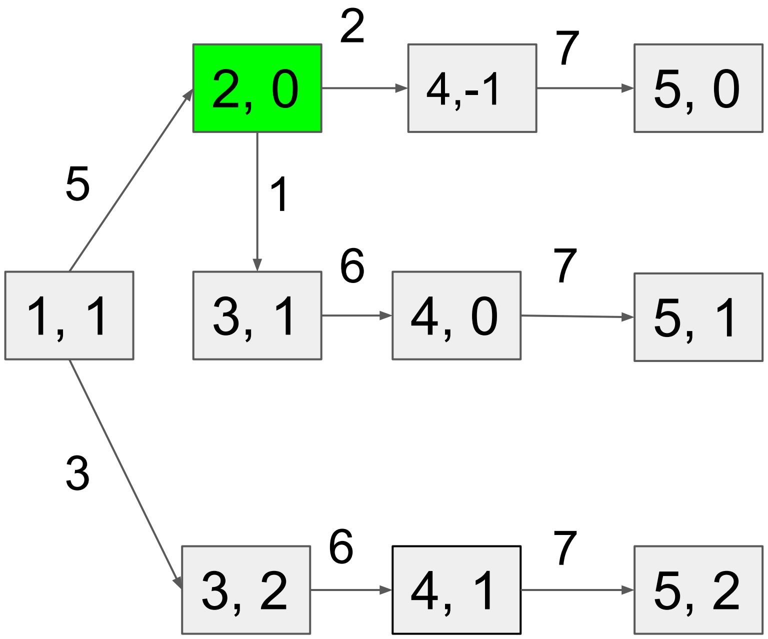
Simulation of UCS



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

Frontier:
(2, 0) : 5
(4, 1) : 9

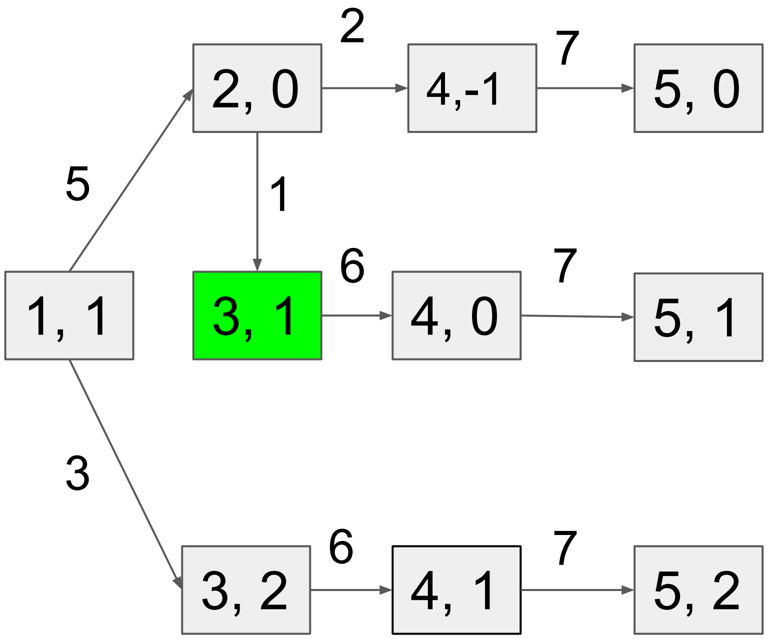
Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5

Frontier:
(3, 1) : 6
(4, -1) : 7
(4, 1) : 9

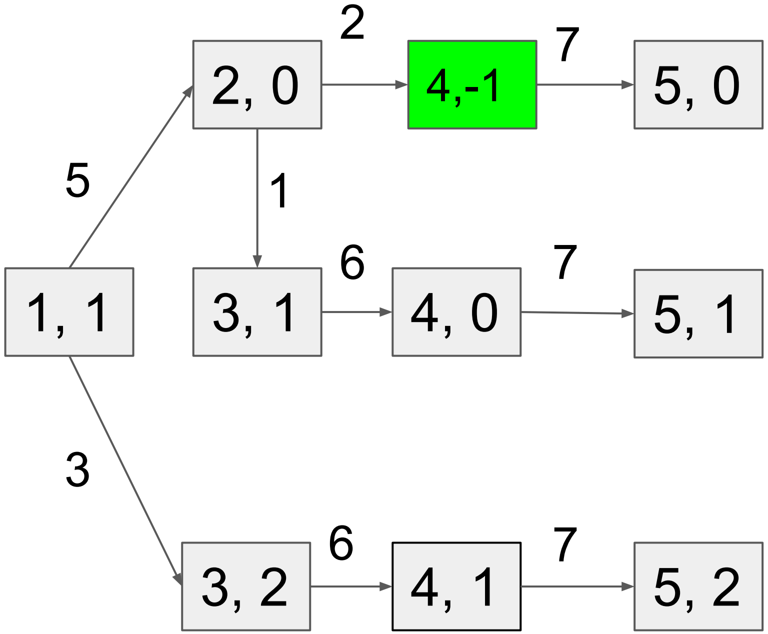
Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5
(3, 1): 6

Frontier:
(4, -1) : 7
(4, 1) : 9
(4, 0): 12

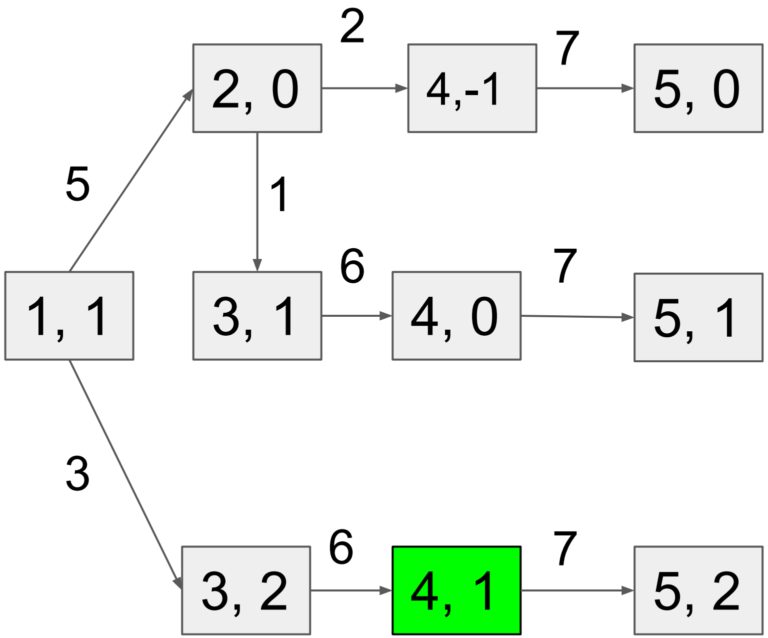
Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5
(3, 1) : 6
(4, -1) : 7

Frontier:
(4, 1) : 9
(4, 0) : 12
(5, 0) : 14

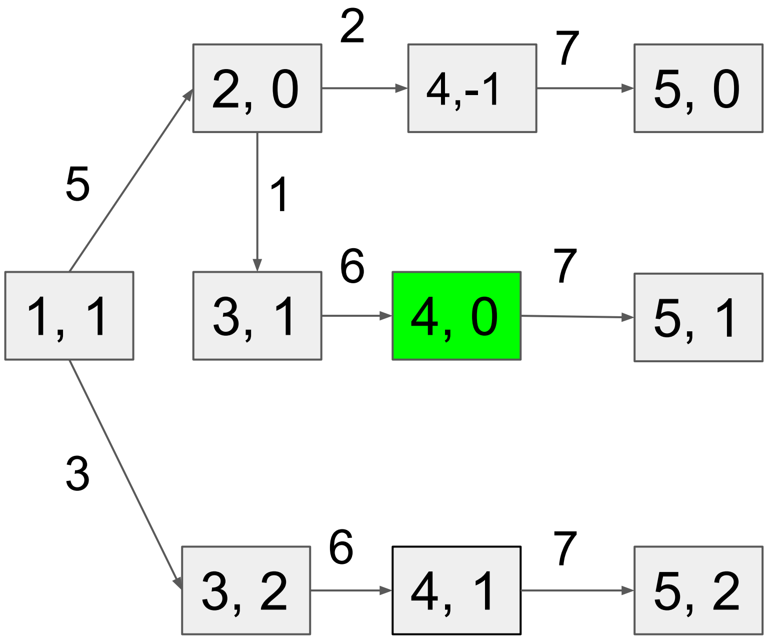
Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5
(3, 1) : 6
(4, -1) : 7
(4, 1) : 9

Frontier:
(4, 0) : 12
(5, 0) : 14
(5, 2) : 16

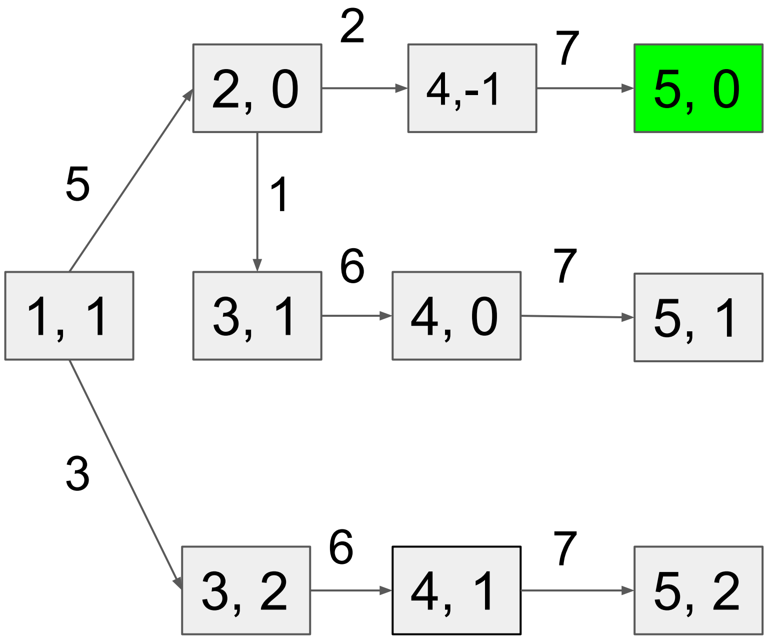
Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5
(3, 1) : 6
(4, -1) : 7
(4, 1) : 9
(4, 0) : 12

Frontier:
(5, 0) : 14
(5, 2) : 16
(5, 1) : 19

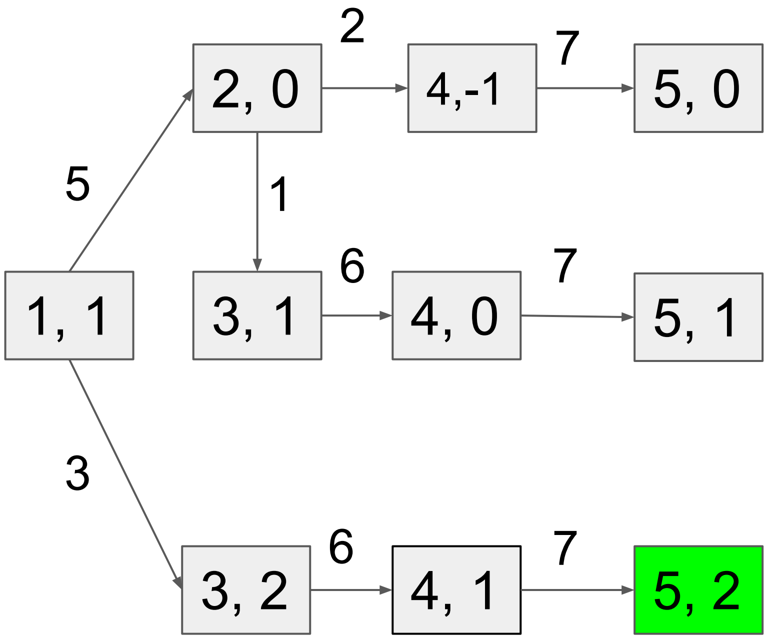
Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5
(3, 1) : 6
(4, -1) : 7
(4, 1) : 9
(4, 0) : 12
(5, 0) : 14

Frontier:
(5, 2) : 16
(5, 1) : 19

Simulation of UCS



Explored:
(1, 1) : 0
(3, 2) : 3
(2, 0) : 5
(3, 1) : 6
(4, -1) : 7
(4, 1) : 9
(4, 0) : 12
(5, 0) : 14
(5, 2) : 16

Frontier:
(5, 1) : 19

STOP!

Comparison between DP and UCS

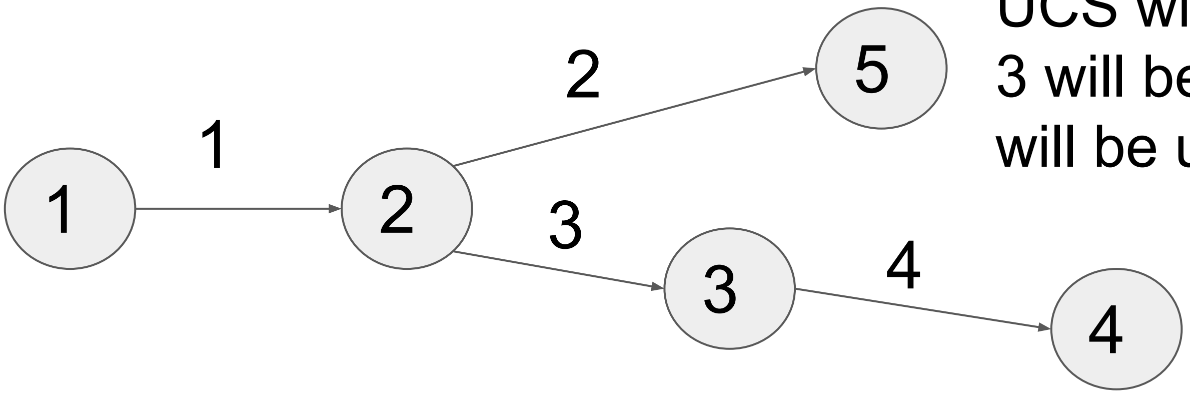
N total states, n of which are closer than goal state

Runtime of DP is O(N)

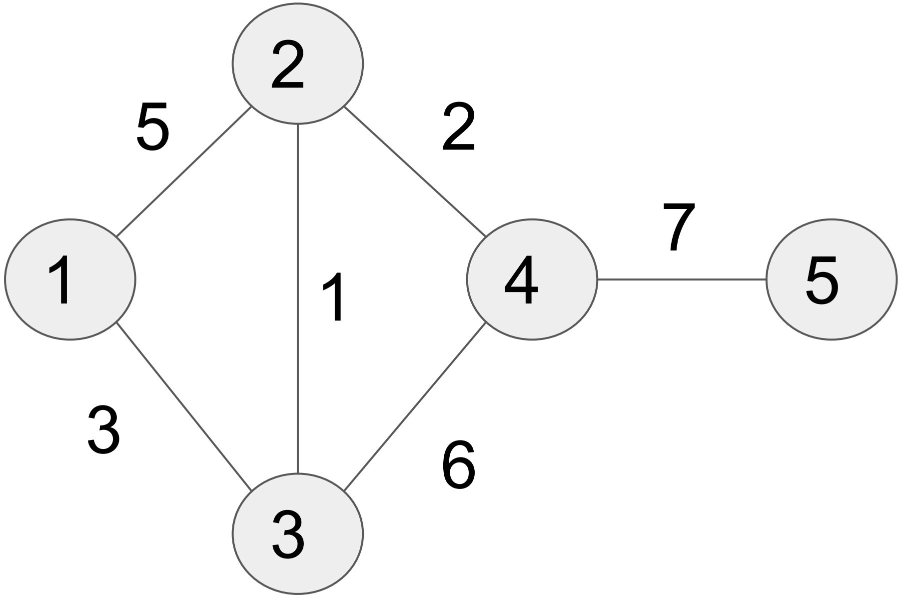
Runtime of UCS is O(n log n)

DP explores O(N) states.

UCS will explore {1, 2, 5} only. 3 will be in the frontier and 4 will be unexplored.



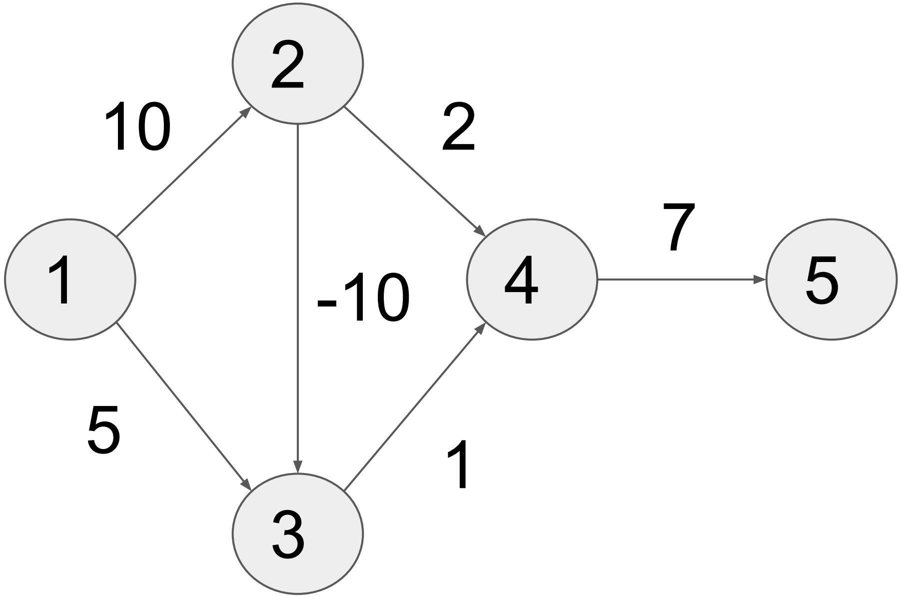
DP cannot handle cycles



Shortest path is [1, 3, 2, 4, 5] with cost 13.

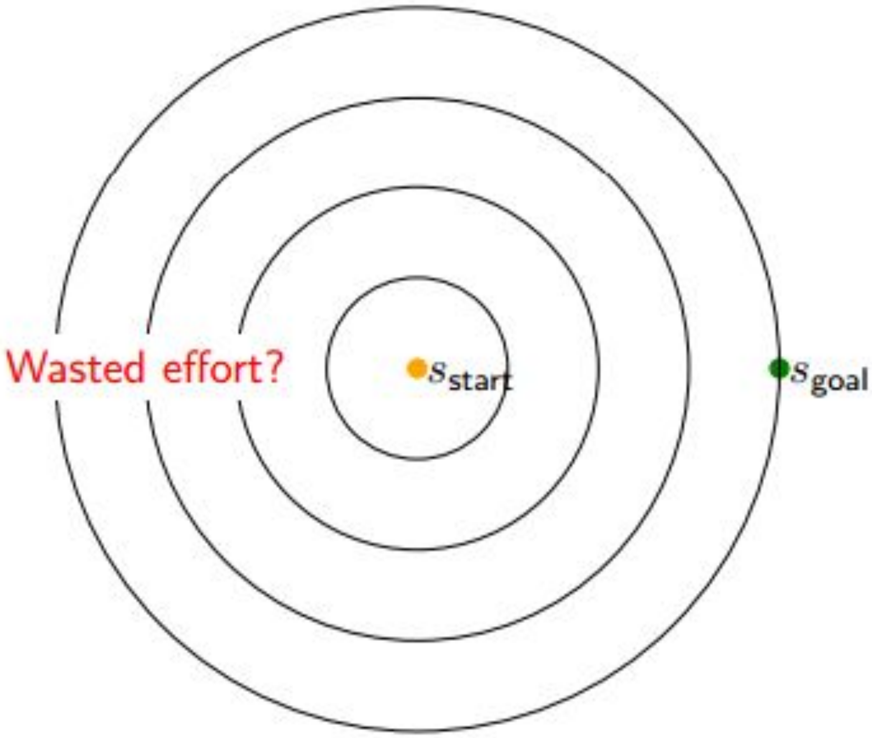
Hard to define subproblems in undirected graphs

UCS cannot handle negative edge weights



Best path is [1,2,3,4,5] with cost of 8, but UCS will output [1,3,4,5] with cost of 13 because 3 is set as ‘explored’ before 2.

Improve UCS: A* Search



Recap of A* Search

- Modify the cost of edges and run UCS on the new graph
  - $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s)$
- $h(s)$ is a heuristic that is our estimate of FutureCost(s)
- If $h(s)$ is consistent then the modified edge weights will return min cost path
  - Consistent: $\text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$
- One can find a good consistent h by performing relaxation
- If c is min cost on original graph, c' is min cost on modified graph, then $c' = c + h(s\_goal) - h(s\_start)$

Relaxation

A good way to come up with a reasonable heuristic is to solve an easier (less constrained) version of the problem

For example, we can remove the constraint that we visit more odd cities than even cities.

h(s) = h((i, d)) = length of shortest path from city i to city N

Note on Relaxation

The main point of relaxation is to attain a problem that **can be solved more efficiently**.

In our case, the modified shortest path problem has O(N) states instead of O(N^2) can thus can be solved more efficiently

Checking consistency

- $\text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$ (Triangle Inequality)
  - Suppose $s = (i, d)$ and $\text{Succ}(s, a) = (j, d')$
  - Note that $h((i, d)) - h((j, d')) \leq c(i, j) = \text{Cost}(s, a)$
- $h((N, d)) = 0$

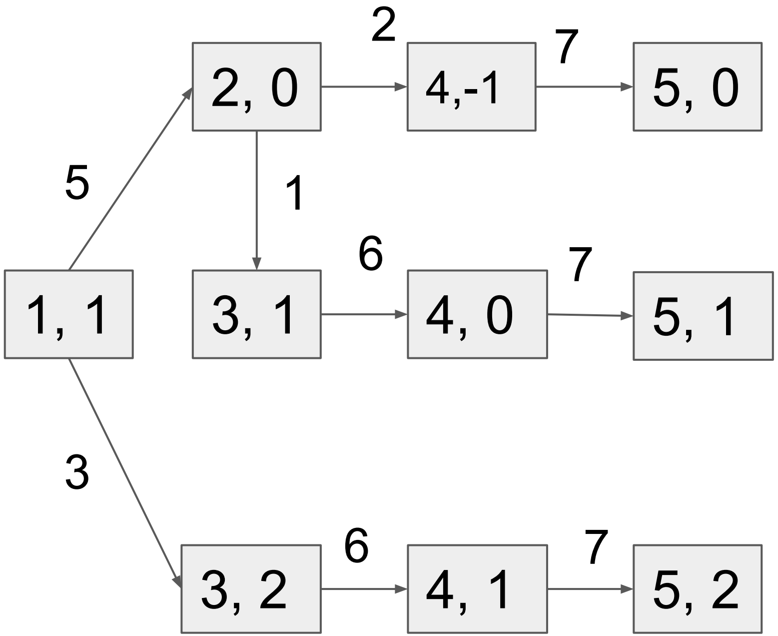
How to compute h?

We can reverse the direction of all edges, and then perform UCS starting from city N, and our goal state is city 1.

This takes O(n log n) time, where n is the number of states whose distance to city N is no farther than the distance of city 1 to city N

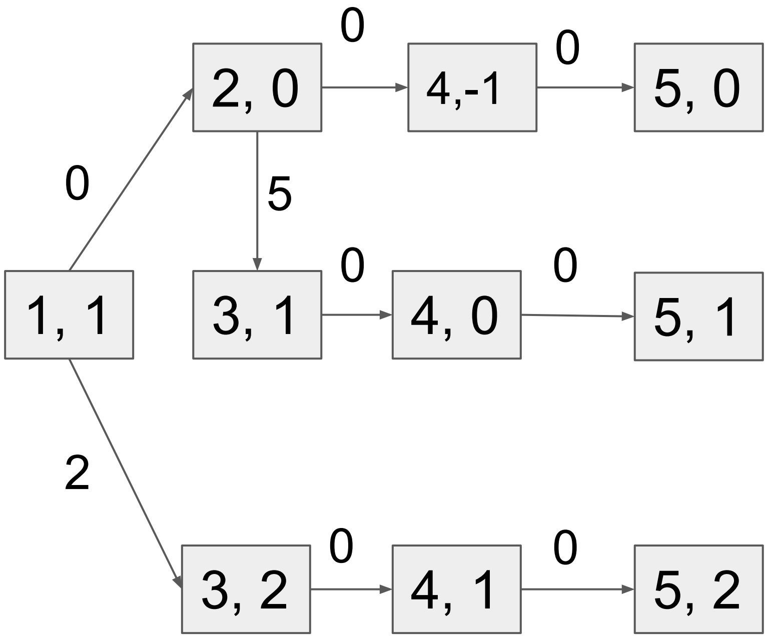
city	1	2	3	4	5
h	14	9	13	7	0

Original State Graph



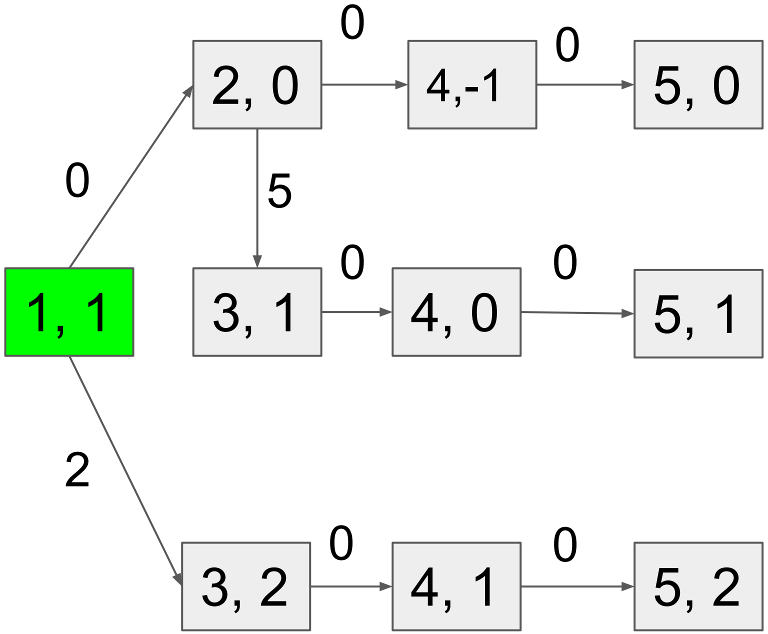
Cost(s, a)

Modified State Graph



Cost’(s, a)
= Cost(s, a) + h(Succ(s, a)) - h(s)

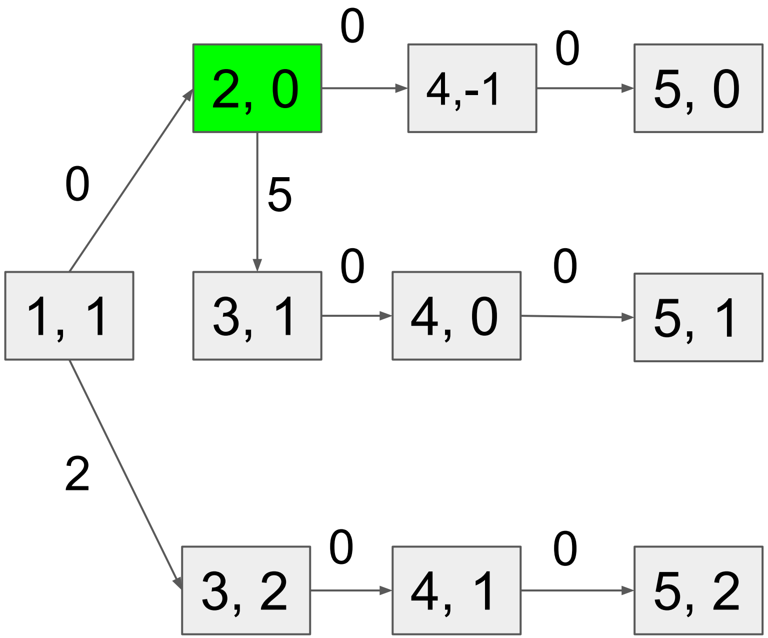
Simulation of UCS (A*)



Explored:
(1, 1) : 0

Frontier:
(2, 0) : 0
(3, 2) : 2

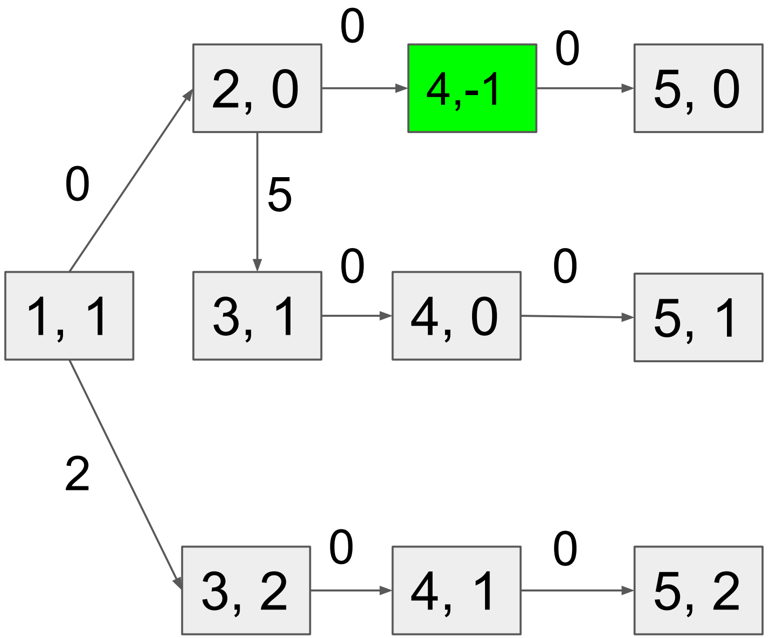
Simulation of UCS (A*)



Explored:
(1, 1) : 0
(2, 0) : 0

Frontier:
(4, -1) : 0
(3, 2) : 2
(3, 1) : 5

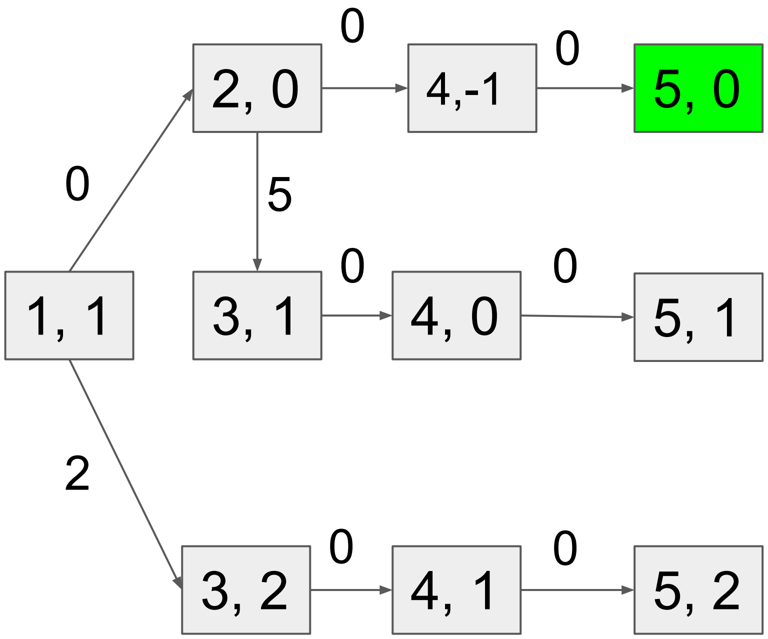
Simulation of UCS (A*)



Explored:
(1, 1) : 0
(2, 0) : 0
(4, -1) : 0

Frontier:
(5, 0) : 0
(3, 2) : 2
(3, 1) : 5

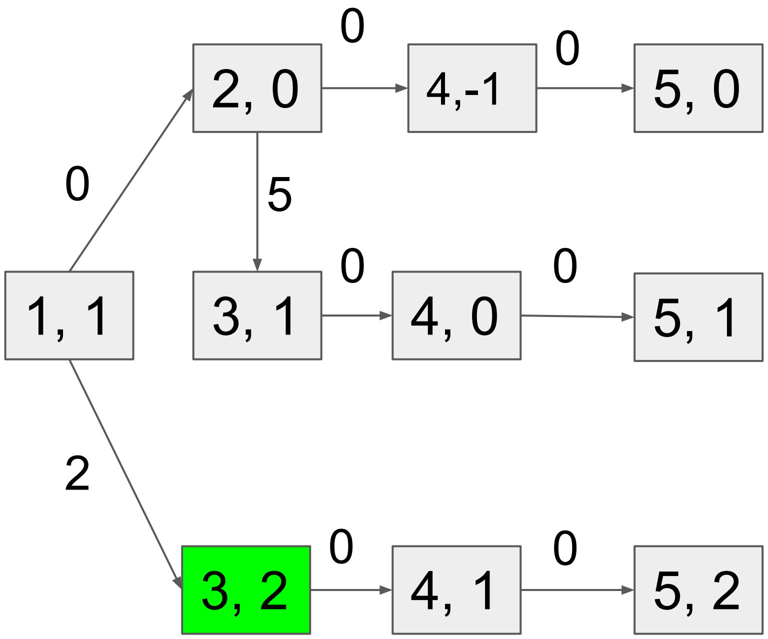
Simulation of UCS (A*)



Explored:
(1, 1) : 0
(2, 0) : 0
(4, -1) : 0
(5, 0) : 0

Frontier:
(3, 2) : 2
(3, 1) : 5

Simulation of UCS (A*)



Explored:

(1, 1) : 0

(2, 0) : 0

(4, -1) : 0

(5, 0) : 0

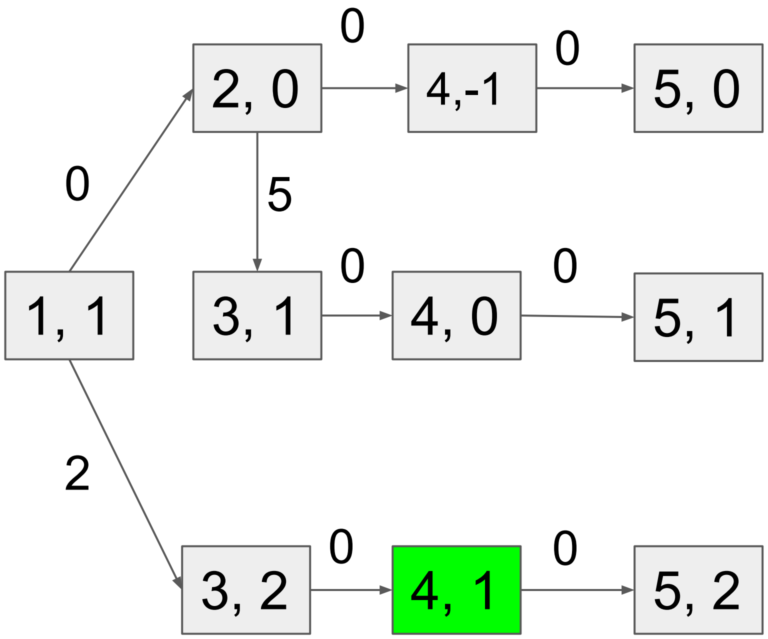
(3, 2) : 2

Frontier:

(4, 1) : 2

(3, 1) : 5

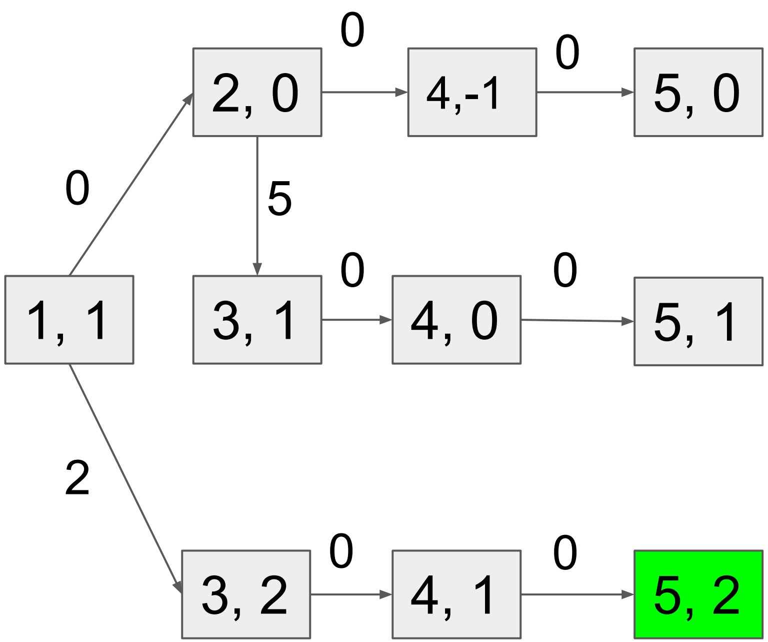
Simulation of UCS (A*)



Explored:
(1, 1) : 0
(2, 0) : 0
(4, -1) : 0
(5, 0) : 0
(3, 2) : 2
(4, 1): 2

Frontier:
(5, 2) : 2
(3, 1) : 5

Simulation of UCS (A*)



Explored:
(1, 1) : 0
(2, 0) : 0
(4, -1) : 0
(5, 0) : 0
(3, 2) : 2
(4, 1) : 2
(5, 2) : 2

Frontier:
(3, 1) : 5

STOP!

Actual Cost is 2 + h(1) - h(5) = 2 + 14 - 0= 16

Comparison of States visited

UCS

Explored:	Frontier:
(1, 1) : 0	(5, 1) : 19
(3, 2) : 3	
(2, 0) : 5	
(3, 1) : 6	
(4, -1) : 7	
(4, 1) : 9	
(4, 0) : 12	
(5, 0) : 14	
(5, 2) : 16	

UCS(A*)

Explored:	Frontier:
(1, 1) : 0	(3, 1) : 5
(2, 0) : 0	
(4, -1) : 0	
(5, 0) : 0	
(3, 2) : 2	
(4, 1) : 2	
(5, 2) : 2	

Why Search?

- Search as a verb
  - How do we choose the next state?
  - How do we handle various types of graphs?
  - How do we prove optimality of our paths?
  - When do we terminate the search?

- Search as a noun?

Search as a Problem-Solving Paradigm

So far, we've been thinking fairly procedurally about search, and how we *implement* search algorithms.

In an algorithms class, we'd talk a lot more about designing and analyzing various search algorithms.

In Artificial Intelligence, often the hard part is *modeling* problems so that we can use known algorithms.

Implementation vs. Abstraction

Fundamental notion in computer science

Examples: HashMap, sparse vectors, NumPy array, …

Separation of concerns via an *interface*

Implementers handle the “behind the scenes” work

Clients must figure out how to model their problem using the abstraction

In this class, you will get to wear both hats!

What makes search a good abstraction?

Explicit search problems: graphs show up everywhere!

- Transportation networks, e.g. roads, flights, shipping
- Information and communication networks, e.g. web search, IR, networking
- Biological networks, e.g. protein-protein interaction networks

Implicit search problems

- Problem solving as search for an answer
- Problem solving as *sequential decision making*: decisions have consequences and lead to new subproblems

Using the Search Abstraction

What are the key ingredients we need in order to model our problem using the search abstraction?

State: What’s really relevant in our problem? What information is extraneous? How do we *represent* the entities of interest in our problem?

Actions: How can we parameterize the space of allowable decisions?

Rewards: What do we really care about achieving?

Termination: When are we done?

Answering these questions gives us a better understanding of our problem!

The State-Action Paradigm

The notions of *states* and *actions* are fundamental to computer science

Early CS theory: Finite Automata

- DFAs, NFAs, Turing machines played a fundamental role in the origins of CS as a field
- Still used to design compilers!

Object Oriented Programming

- Objects have *state* and *methods* that can mutate state
- Bundling states and actions together allows for *encapsulation*
- Key differentiator between OOP and functional programming!

Reinforcement Learning, Games, Sequential Decision Making Problems

Where to go from here?

Advanced search algorithms:

- All-pairs shortest paths
- Negative weights?
- Sparse graphs and graphs with other special structure
- Parallel/distributed algorithms
- Cache-efficient algorithms and programming the memory hierarchy

Modeling problems as search:

- Rubik’s Cubes, mazes, other spatial puzzles
- 2-player games, chess puzzles
- Automatic theorem proving
- Reinforcement learning and sequential decision making
- Problems in your area of research or other things you’re interested in!

Questions?

Summary

- States Representation/Modelling
  - make state representation as compact as possible, remove unnecessary information
- DP
  - underlying graph cannot have cycles
  - visit all reachable states, but no log overhead
- UCS
  - actions cannot have negative cost
  - visit only a subset of states, log overhead
- A*
  - ensure that relaxed problem can be solved more efficiently