
# Lecture 2.2: Search II

# Roadmap

**Designing state spaces**

Uniform cost search

Preview of A* search

# 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($s$): ...

Advantage over backtracking search: Process every state exactly once.

Corollary: Runtime is $O(Nb)$, where $N$ is total number of states, $b$ is branching factor (actions per state).

# Dynamic programming

**Key idea: state**

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

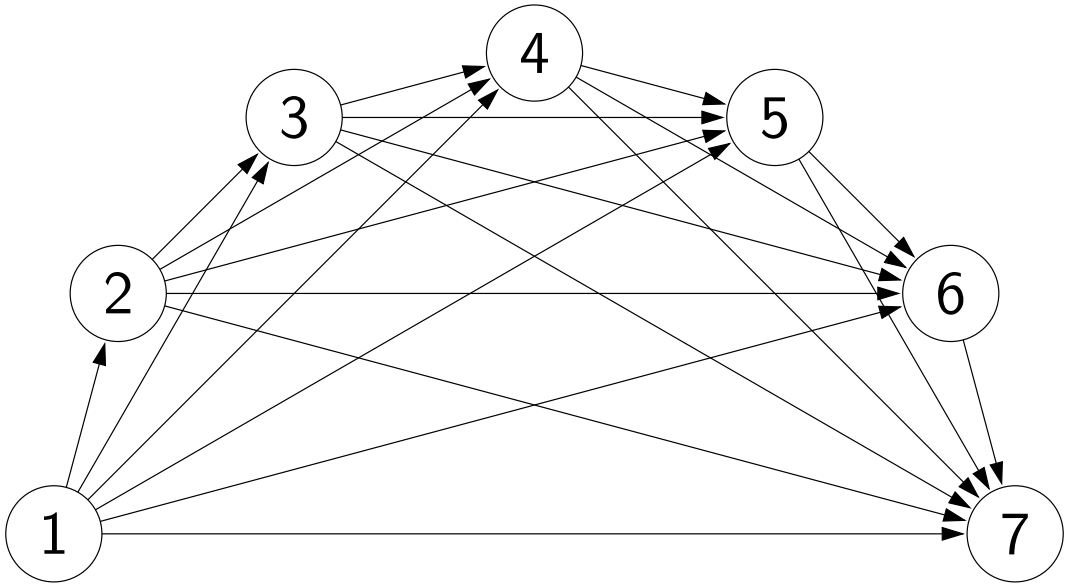
| | |
|---|---|
| past actions (all cities) | 1 3 4 6 |
| state (current city) | 1 3 4 6 |

- Recall that dynamic programming visits every state once, which means its runtime is largely controlled by how many states there are.
- 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.
- Recall that in general, 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 $j$.




Observation: future costs only depend on current city

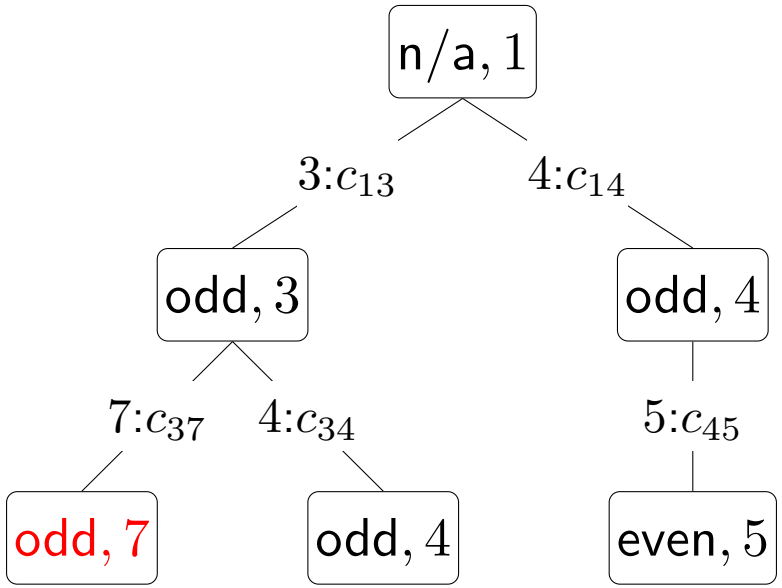
# Handling additional constraints

**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)



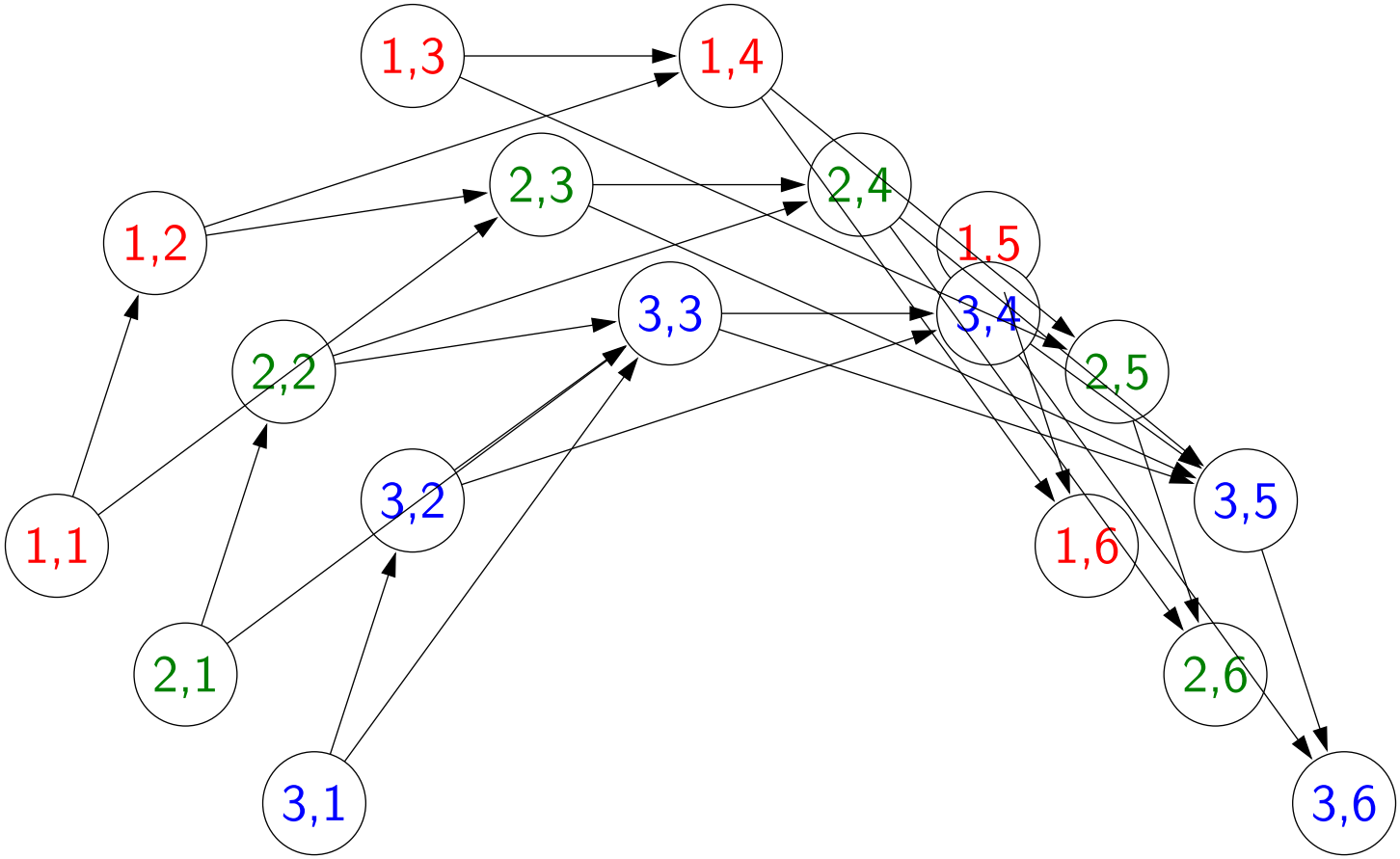
- Let's add a constraint that says we can't visit three odd cities in a row. If we only keep track of the current city, and we try to move to a next city, we cannot enforce this constraint because we don't know what the previous city was. So let's add the previous city into the state.
- This will work, 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.

# 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.

# 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
- To handle complex constraints, need to add additional information to our state
- To speed up dynamic programming, find a minimal state space

Dynamic programming only works for acyclic graphs...what if there are cycles?

# Roadmap

Designing state spaces

**Uniform cost search**

Preview of A* search

# Ordering the states

Observation: suffixes of optimal path are optimal

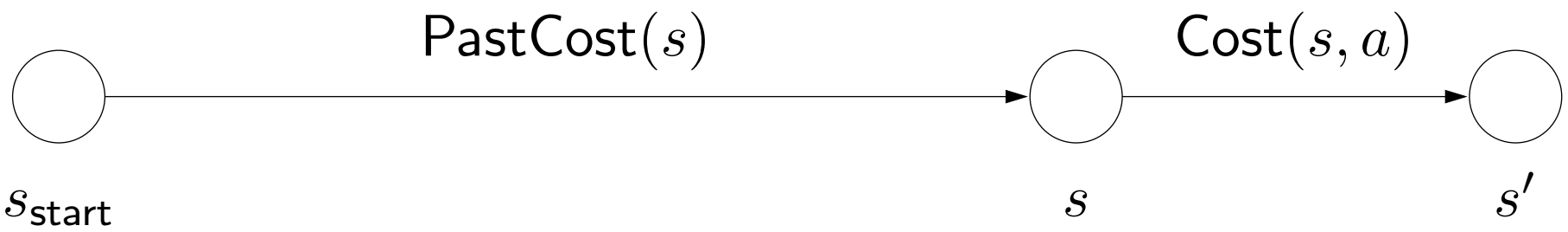

Dynamic programming: if graph is acyclic, DP makes sure we compute FutureCost$(s')$ before FutureCost$(s)$

If graph is cyclic, then we need another mechanism to order states...

- Recall that we used dynamic programming to compute the future cost of each state $s$, the cost of the minimum cost path from $s$ to a end state.
- Dynamic programming relies on the absence of cycles, so that there is always a clear order in which to compute all the future costs. If the future costs of all the successors of a state $s$ are computed, then we could compute the future cost of $s$ by taking the minimum.
- Note that FutureCost$(s)$ will always be computed after FutureCost$(s')$ if there is an edge from $s$ to $s'$. In essence, the future costs will be computed according to a topological ordering of the nodes.
- However, when there are cycles, no topological ordering exists, so we need another way to order the states.
- The particular DP implementation I showed in class therefore seems to reason "backwards" about future cost, but that is not fundamental to the algorithm. We can analogously define PastCost$(s)$, the cost of the minimum cost path from the start state to $s$. If instead of having access to the successors via Succ$(s, a)$, we had access to predecessors (think of reversing the edges in the state graph), then we could define a dynamic program to compute all the PastCost$(s)$.

# Ordering the states

Observation: prefixes of optimal path are optimal




Constant edge costs (BFS, DFS-ID): compute PastCost($s$) before PastCost($s'$)

- Past vs. future not important–symmetrical

If edges have different costs, how to order states?

- Similarly, when we had constant costs for all actions, breadth-first search or DFS with iterative deepening explored states in order of their past cost.
- This relies on the same idea: once you have explored all states at depth $d$, and one of them leads to state $s$, then you know $s$ has depth $d+1$. The depth-$d$ nodes are the "predecessors" of $s$, rather than successors.
- However, if costs are different for different actions, breadth-first order does not have these same guarantees.

# Uniform cost search (UCS)

**Key idea: state ordering**

UCS enumerates states in order of increasing past cost.

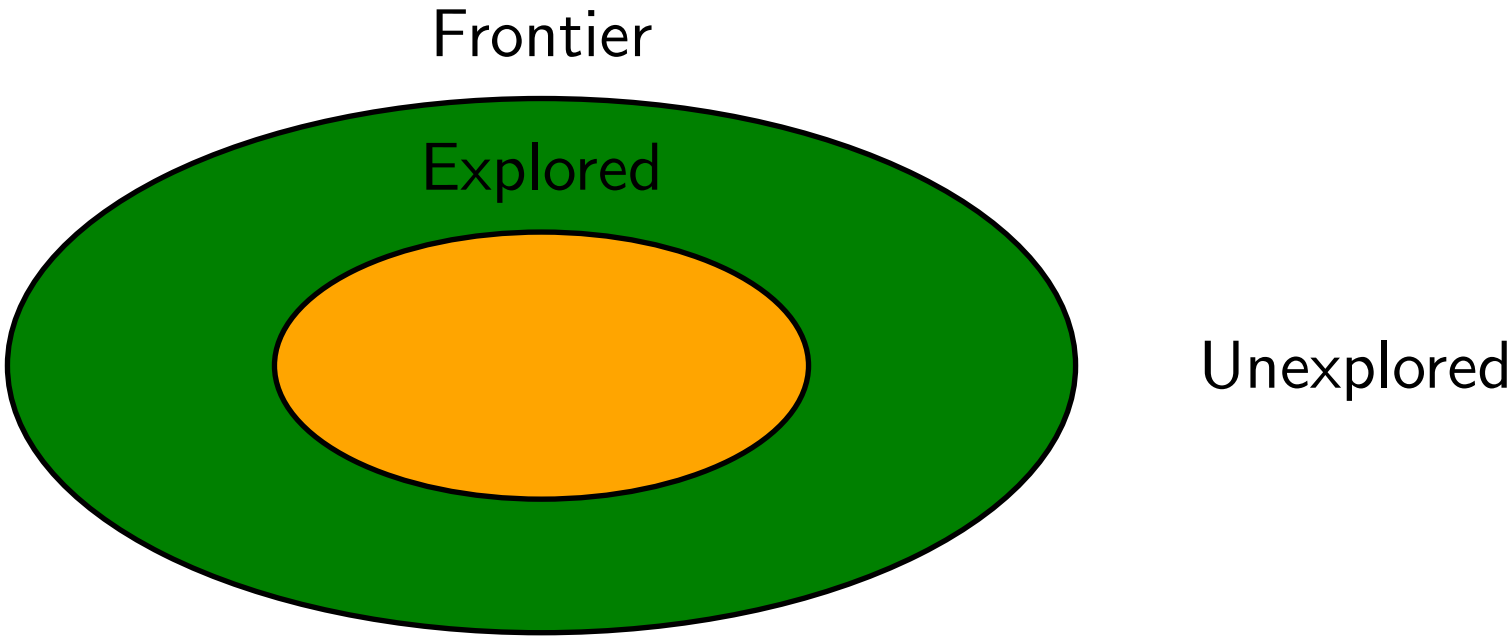

**Assumption: non-negativity**

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

UCS in action:

- The key idea that uniform cost search (UCS) uses is to compute the past costs in order of increasing past cost. To make this efficient, we need to make an important assumption that all action costs are non-negative.
- This assumption is reasonable in many cases, but doesn't allow us to handle cases where actions have payoff. To handle negative costs (positive payoffs), we need the Bellman-Ford algorithm. When we talk about value iteration for MDPs, we will see a form of this algorithm.
- Note: those of you who have studied algorithms should immediately recognize UCS as Dijkstra's algorithm. Logically, the two are indeed equivalent. There is an important implementation difference: UCS takes as input a **search problem**, which implicitly defines a large and even infinite graph, whereas Dijkstra's algorithm (in the typical exposition) takes as input a fully concrete graph. The implicitness is important in practice because we might be working with an enormous graph (a detailed map of world) but only need to find the path between two close by points (Stanford to Palo Alto).
- Another difference is that Dijkstra's algorithm is usually thought of as finding the shortest path from the start state to every other node, whereas UCS is explicitly about finding the shortest path to an end state. This difference is sharpened when we look at the A* algorithm next time, where knowing that we're trying to get to the goal can yield a much faster algorithm. The name uniform cost search refers to the fact that we are exploring states of the same past cost uniformly (the video makes this visually clear); in contrast, A* will explore states which are biased towards the end state.

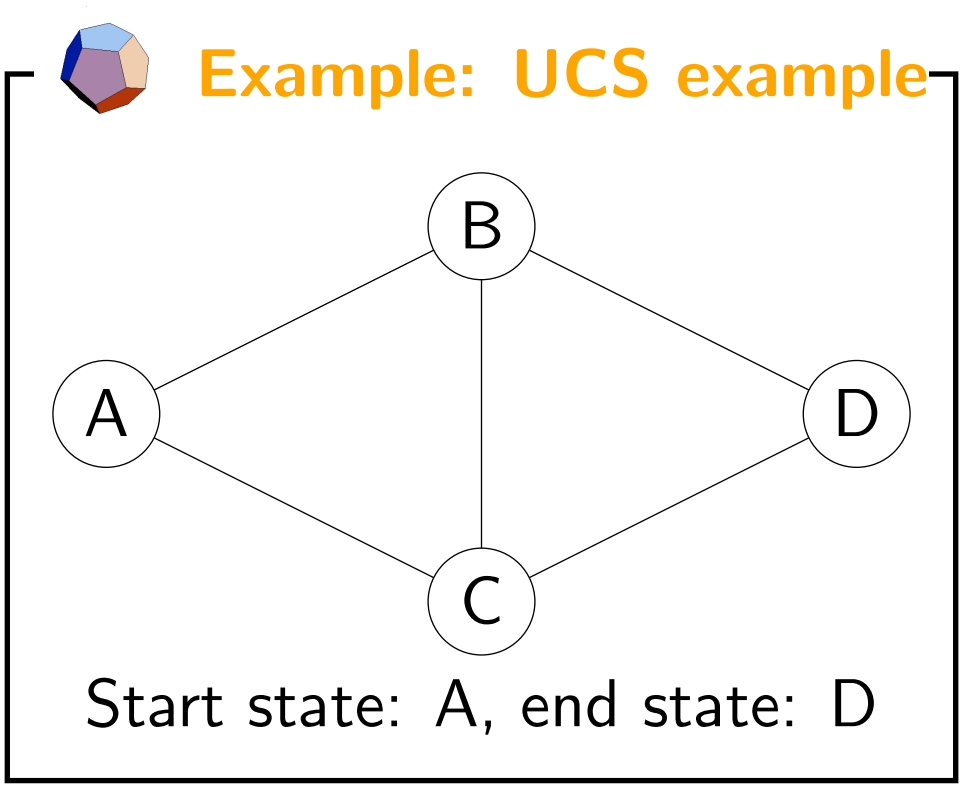
# High-level strategy




- 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

- The general strategy of UCS is to maintain three sets of nodes: explored, frontier, and unexplored. Throughout the course of the algorithm, we will move states from unexplored to frontier, and from frontier to explored.
- The key invariant is that we have computed the minimum cost paths to all the nodes in the explored set. So when the end state moves into the explored set, then we are done.

# Uniform cost search example




[whiteboard]

Minimum cost path:

$$A \rightarrow B \rightarrow C \rightarrow D \text{ with cost } 3$$

- Before we present the full algorithm, let's walk through a concrete example.
- Initially, we put A on the frontier. We then take A off the frontier and mark it as explored. We add B and C to the frontier with past costs 1 and 100, respectively.
- Next, we remove from the frontier the state with the minimum past cost (priority), which is B. We mark B as explored and consider successors A, C, D. We ignore A since it's already explored. The past cost of C gets updated from 100 to 2. We add D to the frontier with initial past cost 101.
- Next, we remove C from the frontier; its successors are A, B, D. A and B are already explored, so we only update D's past cost from 101 to 3.
- Finally, we pop D off the frontier, find that it's a end state, and terminate the search.

# Uniform cost search (UCS)

**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 IsEnd$(s)$: 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$(s, a)$

[live solution]

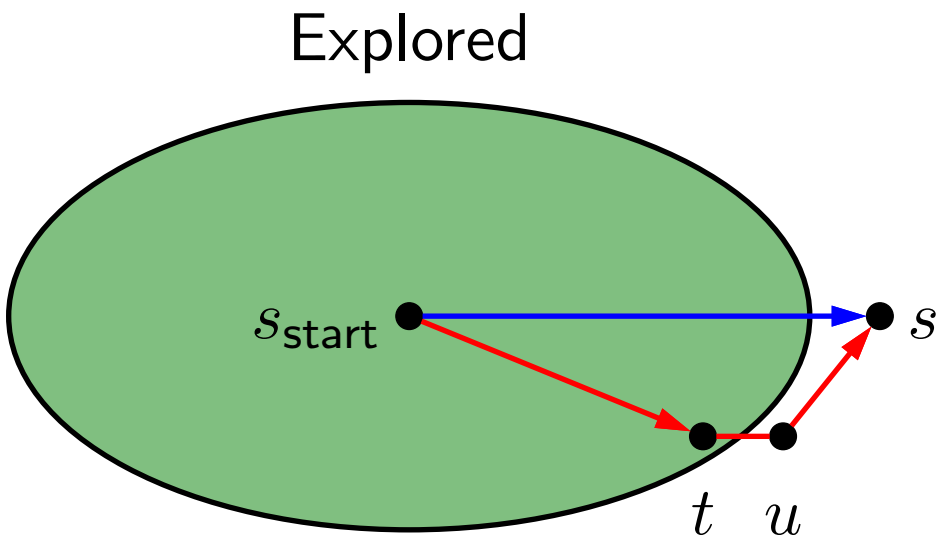
- Implementation note: we use `util.PriorityQueue` which supports `removeMin` and `update`. Note that `frontier.update(state, pastCost)` returns whether `pastCost` improves the existing estimate of the past cost of `state`.

# 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:



- Let $p_s$ be the priority of $s$ when $s$ is popped off the frontier. Since all costs are non-negative, $p_s$ increases over the course of the algorithm.
- Suppose we pop $s$ off the frontier. Let the blue path denote the path with cost $p_s$.
- Consider any alternative red path from the start state to $s$. The red path must leave the explored region at some point; let $t$ and $u = \text{Succ}(t, a)$ be the first pair of states straddling the boundary. We want to show that the red path cannot be cheaper than the blue path via a string of inequalities.
- First, by definition of $\text{PastCost}(t)$ and non-negativity of edge costs, the cost of the red path is at least the cost of the part leading to $u$, which is $\text{PastCost}(t) + \text{Cost}(t, a) = p_t + \text{Cost}(t, a)$, where the last equality is by the inductive hypothesis.
- Second, we have $p_t + \text{Cost}(t, a) \geq p_u$ since we updated the frontier based on $(t, a)$.
- Third, we have that $p_u \geq p_s$ because $s$ was the minimum cost state on the frontier.
- Note that $p_s$ is the cost of the blue path.

# 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 | $\geq 0$ | $O(n \log n)$ |

Note: UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

Note: assume number of actions per state is constant (independent of $n$ and $N$)

- DP and UCS have complementary strengths and weaknesses; neither dominates the other.
- DP can handle negative action costs, but is restricted to acyclic graphs. It also explores all $N$ reachable states from $s_{\text{start}}$, which is inefficient. This is unavoidable due to negative action costs.
- UCS can handle cyclic graphs, but is restricted to non-negative action costs. An advantage is that it only needs to explore $n$ states, where $n$ is the number of states which are cheaper to get to than any end state. However, there is an overhead with maintaining the priority queue.
- One might find it unsatisfying that UCS can only deal with non-negative action costs. Can we just add a large positive constant to each action cost to make them all non-negative? It turns out this doesn't work because it penalizes longer paths more than shorter paths, so we would end up solving a different problem.

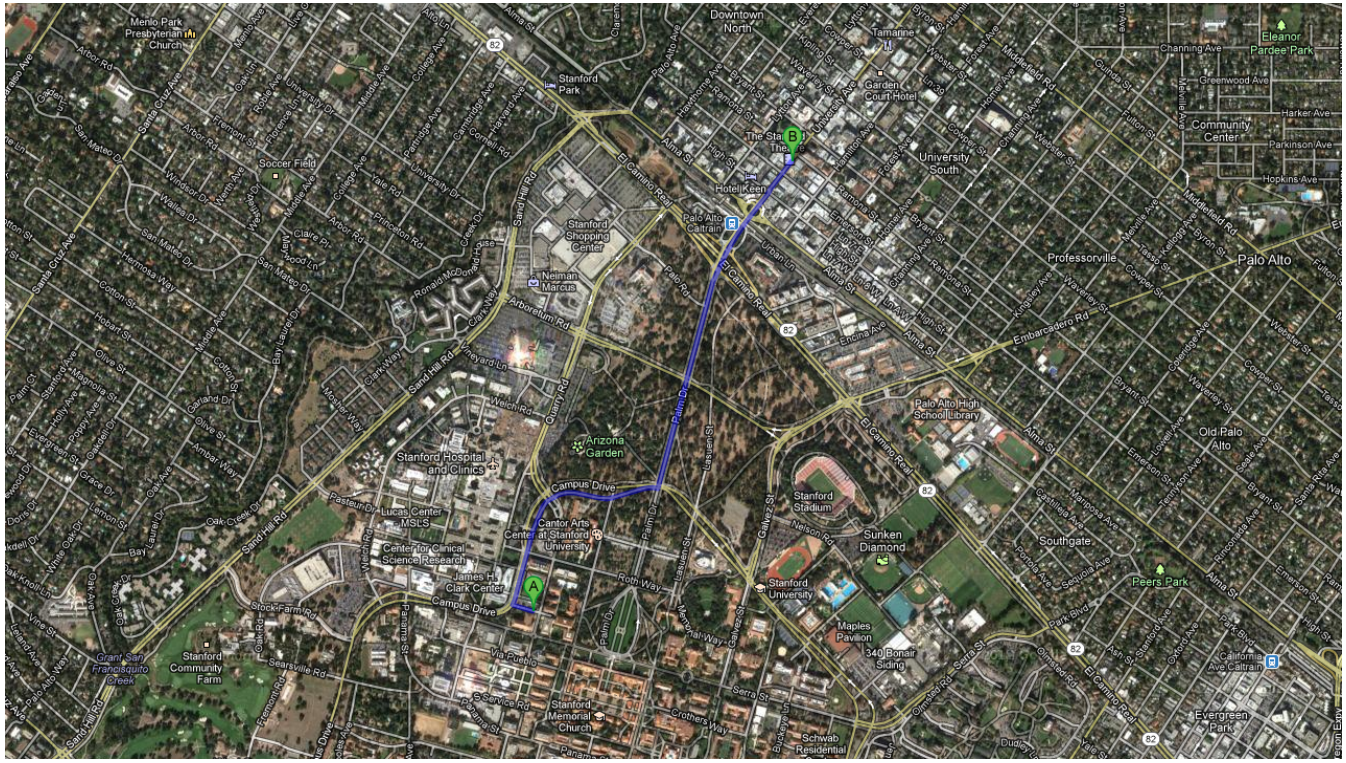
# Roadmap

Designing state spaces

Uniform cost search

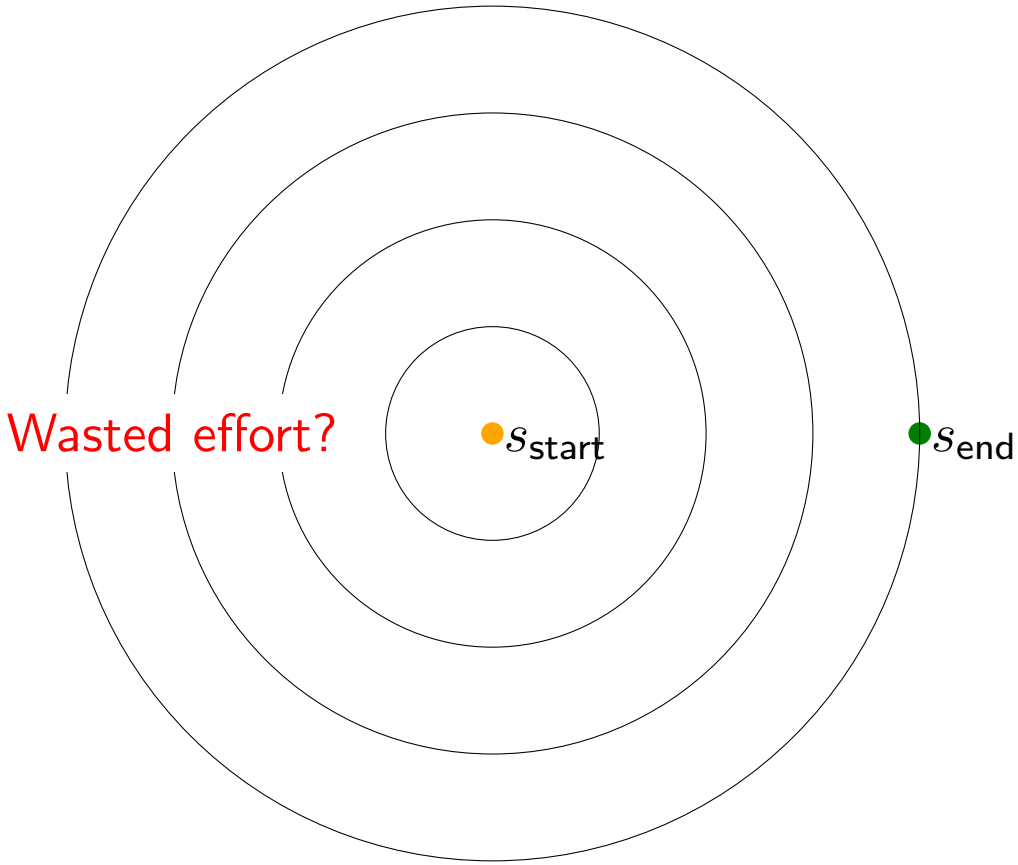
**Preview of A* search**

# Application: route finding




How would find the fastest way to walk from here to Downtown Palo Alto?

# Can uniform cost search be improved?




Problem: UCS orders states by cost from $s_{\text{start}}$ to $s$

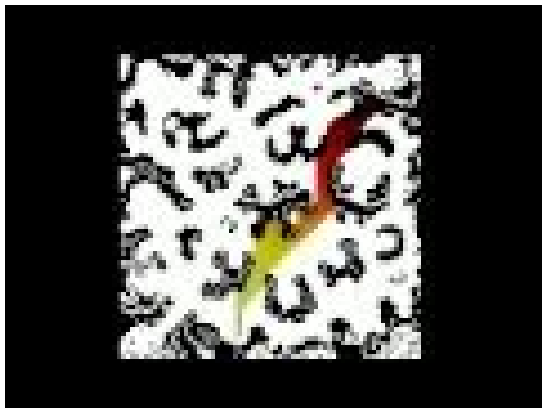
Goal: take into account cost from $s$ to $s_{\text{end}}$

- Now our goal is to make UCS faster. If we look at the UCS algorithm, we see that it explores states based on how far they are away from the start state. As a result, it will explore many states which are close to the start state, but in the opposite direction of the end state.
- Intuitively, we'd like to bias UCS towards exploring states which are closer to the end state, and that's exactly what A* does.

# A* algorithm

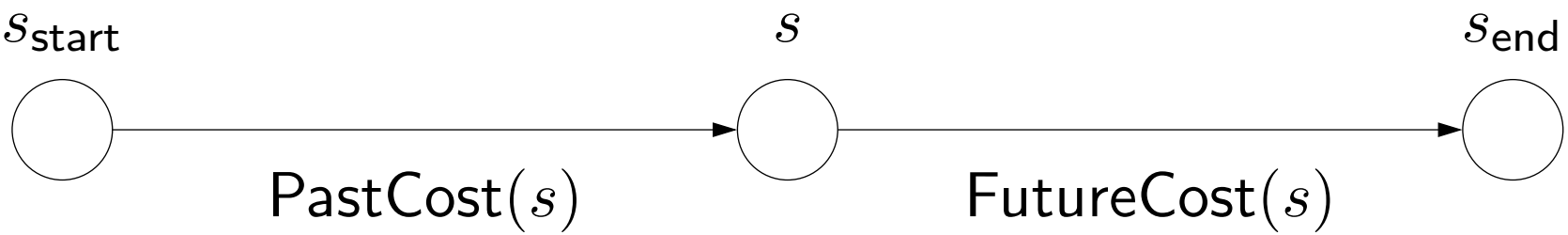
UCS in action:

A* in action:



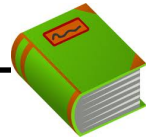
# Exploring states

UCS: explore states in order of PastCost$(s)$




Ideal: explore in order of PastCost$(s)$ + FutureCost$(s)$

A*: explore in order of PastCost$(s)$ + $h(s)$



**Definition: Heuristic function**

A heuristic $h(s)$ is any estimate of FutureCost$(s)$.

- First, some terminology: PastCost($s$) is the minimum cost from the start state to $s$, and FutureCost($s$) is the minimum cost from $s$ to an end state. Without loss of generality, we can just assume we have one end state. (If we have multiple ones, create a new official goal state which is the successor of all the original end states.)
- Recall that UCS explores states in order of PastCost($s$). It'd be nice if we could explore states in order of PastCost($s$) + FutureCost($s$), which would definitely take the end state into account, but computing FutureCost($s$) would be as expensive as solving the original problem.
- A* relies on a **heuristic** $h(s)$, which is an estimate of FutureCost($s$). For A* to work, $h(s)$ must satisfy some conditions, but for now, just think of $h(s)$ as an approximation. We will soon show that A* will explore states in order of PastCost($s$) + $h(s)$. This is nice, because now states which are estimated (by $h(s)$) to be really far away from the end state will be explored later, even if their PastCost($s$) is small.

# A* search

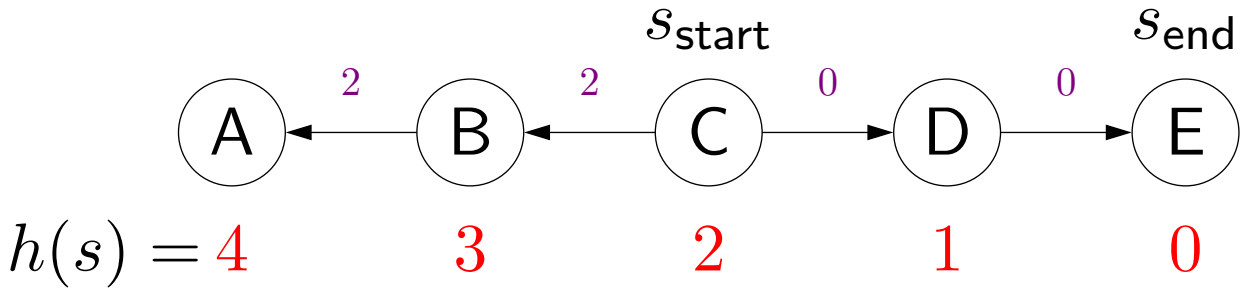
**Algorithm: A* search [Hart/Nilsson/Raphael, 1968]**

Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)$$

Intuition: add a penalty for how much action $a$ takes us away from the end state

Example:




$$\text{Cost}'(C,B) = \text{Cost}(C,B) + h(B) - h(C) = 1 + (3-2) = 2$$

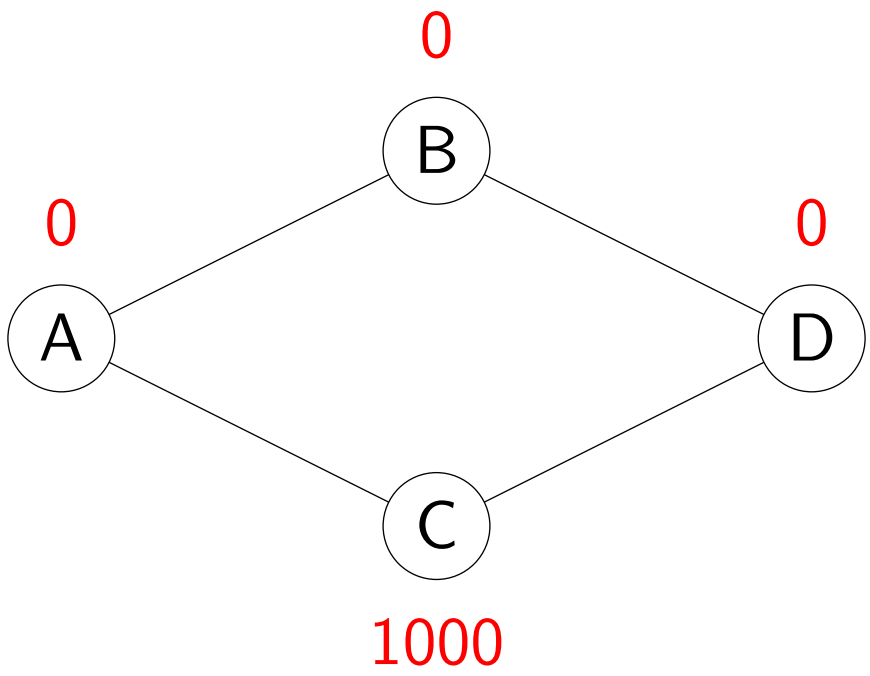
- Here is the full A* algorithm: just run UCS with modified edge costs.
- You might feel tricked because we promised you a shiny new algorithm, but actually, you just got a refurbished version of UCS. (This is a slightly unorthodox presentation of A*. The normal presentation is modifying UCS to prioritize by PastCost$(s) + h(s)$ rather than PastCost$(s)$.) But I think the modified edge costs view shows a deeper connection to UCS, and we don't even have to modify the UCS code at all.
- How should we think of these modified edge costs? It's the same edge cost Cost$(s, a)$ plus an additional term. This term is difference between the estimated future cost of the new state Succ$(s, a)$ and that of the current state $s$. In other words, we're measuring how much farther from the end state does action $a$ take us. If this difference is positive, then we're penalizing the action $a$ more. If this difference is negative, then we're favoring this action $a$.
- Let's look at a small example. All edge costs are $1$. Let's suppose we define $h(s)$ to be the actual FutureCost$(s)$, the minimum cost to the end state. In general, this is not the case, but let's see what happens in the best case. The modified edge costs are $2$ for actions moving away from the end state and $0$ for actions moving towards the end state.
- In this case, UCS with original edge costs $1$ will explore all the nodes. However, A* (UCS with modified edge costs) will explore only the three nodes on the path to the end state.

# An example heuristic

Will any heuristic work?

No.

Counterexample:




Doesn't work because of **negative modified edge costs**!

- So far, we've just said that $h(s)$ is just an approximation of FutureCost$(s)$. But can it be any approximation?
- The answer is no, as the counterexample clearly shows. The modified edge costs would be 1 (A to B), 1002 (A to C), 5 (B to D), and -999 (C to D). UCS would go to B first and then to D, finding a cost 6 path rather than the optimal cost 3 path through C.
- If our heuristic is lying to us (bad approximation of future costs), then running A* (UCS on modified costs) could lead to a suboptimal solution. Note that the reason this heuristic doesn't work is the same reason UCS doesn't work when there are negative action costs.

# Next time

- What makes a heuristic $h(s)$ good?

- How can I come up with a good heuristic?

# Summary

- State: summary of past actions sufficient to choose future actions optimally
- Smaller state spaces mean faster runtime for dynamic programming (and UCS)
- Uniform cost search: handles cycles (but not negative edge costs)
- A* search: improves on UCS by using prior knowledge about where the end state is
- Next time: more on A* search, learning action costs

- Today we thought more deeply about what a **state** really is. We want our state to capture all the information about the past needed to act optimally in the future. But we also don't want it to carry extra information, because that increases the runtime of search algorithms like dynamic programming and uniform cost search.
- With an appropriately defined state, we can apply either dynamic programming or UCS, which have complementary strengths. The former handles negative action costs and the latter handles cycles. Both require space proportional to the number of states, so we need to make sure that we did a good job with the modeling of the state.
- Finally, we got a preview of A* search. The key idea of A* is to improve on UCS by incorporating prior knowledge about what actions take us closer to the goal.