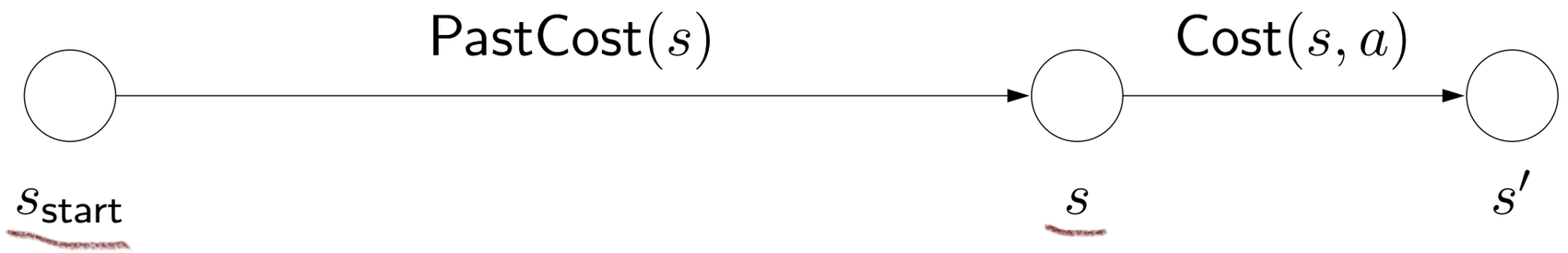
Search: uniform cost search

الگوریتم جستجوی هزینه یکنواخت

Ordering the states

Observation: prefixes of optimal path are optimal



Key: if graph is acyclic, dynamic programming makes sure we compute PastCost(s) before PastCost(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.
- 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).
- Dynamic programming relies on the absence of cycles, so that there is always a clear order in which to compute all the past costs. If the past costs of all the predecessors of a state s are computed, then we could compute the past cost of s by taking the minimum.
- Note that PastCost(s) will always be computed before PastCost(s') if there is an edge from s to s'. In essence, the past 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.

Uniform cost search (UCS)

Key idea: state ordering

UCS enumerates states in order of increasing past cost.

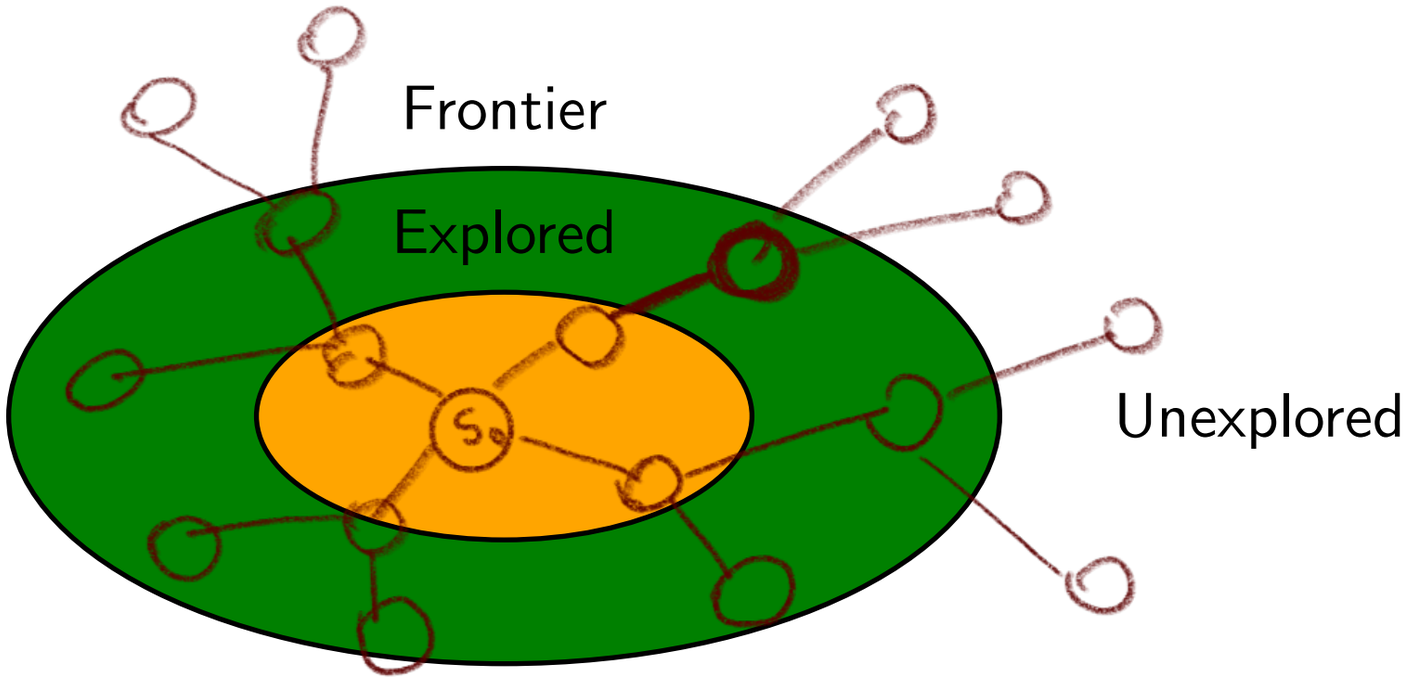
Assumption: non-negativity

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.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

الگوی کلی الگوریتم جستجوی گرافی

قرار دهید

f := PastCost

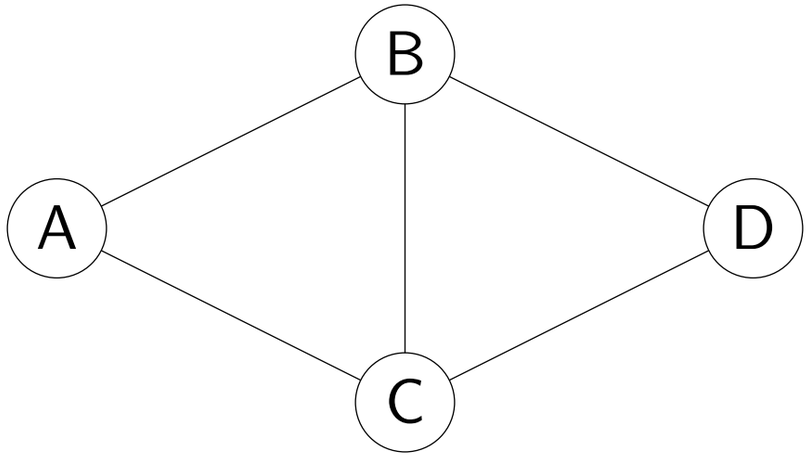
UCS ⇐

f ≡ depth ،

BFS ⇐

Uniform cost search example

Example: UCS example



Start state: A, end state: D

[whiteboard]

Minimum cost path:

$A \rightarrow B \rightarrow C \rightarrow D$ 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.