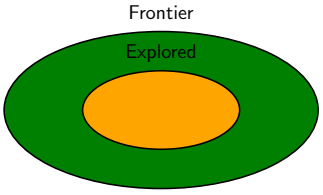
Search: uniform cost search

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 (UCS)

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 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)

[semi-live solution: `Uniform Cost Search`]

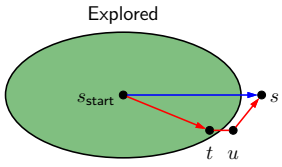
- Implementation note: we use `util.PriorityQueue` which supports `removeMin` and `update`. Note that `frontier.update(state, pastCost)` returns whether `pastCost` improves the existing estimate of the past cost of `state`.

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 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 PastCost(t) + Cost(t, a) = p_t + 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	≥ 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_{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.

Summary

- Tree search: memory efficient, suitable for huge state spaces but exponential worst-case running time
- State: summary of past actions sufficient to choose future actions optimally
- Graph search: dynamic programming and uniform cost search construct optimal paths (exponential savings!)
- Next time: learning action costs, searching faster with A*

- We started out with the idea of a search problem, an abstraction that provides a clean interface between modeling and algorithms.
- Tree search algorithms are the simplest: just try exploring all possible states and actions. With backtracking search and DFS with iterative deepening, we can scale up to huge state spaces since the memory usage only depends on the number of actions in the solution path. Of course, these algorithms necessarily take exponential time in the worst case.
- To do better, we need to think more about bookkeeping. The most important concept from this lecture is the idea of a **state**, which contains all the information about the past to act optimally in the future. We saw several examples of traveling between cities under various constraints, where coming up with the proper minimal state required a deep understanding of 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.