So far, we have looked at three any-path algorithms, depth-first and breadth-first, which are uninformed, and best-first, which is heuristically guided.

Now, we will look at the first algorithm that searches for optimal paths, as defined by a "path length" measure. This uniform cost algorithm is uninformed about the goal, that is, it does not use any heuristic guidance.
Slide 2.4.3

This is the simple algorithm we have been using to illustrate the various searches. As before, we will see that the key issues are picking paths from Q and adding extended paths back in.

Slide 2.4.4

We will continue to use the algorithm but (as we will see) the use of the Visited list conflicts with optimal searching, so we will leave it out for now and replace it with something else later.

Slide 2.4.5

Why can’t we use a Visited list in connection with optimal searching? In the earlier searches, the use of the Visited list guaranteed that we would not do extra work by re-visiting or re-expanding states. It did not cause any failures then (except possibly of intuition).

Slide 2.4.6

But, using the Visited list can cause an optimal search to overlook the best path. A simple example will illustrate this.

---

**Simple Search Algorithm**

A search node is a path from some node to the start state, e.g., \( \langle X \text{BA} S \rangle \)

The state of a search node is the most recent state of the path, e.g., \( X \).

Let Q be a list of search nodes, e.g., \( \langle X \text{BA} S \rangle \langle \text{CBA} S \rangle \ldots \)

Let S be the start state.

1. Initialize Q with search node (S) as only entry; set Visited = (S)
2. If Q is empty, fail. Else, pick some path \( N \) from Q
3. If state(\( N \)) is a goal, return \( N \) (we’ve reached a goal)
4. (Otherwise) Remove \( N \) from Q
5. Find all the children of state(\( N \)) not in Visited and create all the one-step extensions of \( N \) to each descendant.
6. Add all the extended paths to Q; add children of state(\( N \)) to Visited
7. Go to step 2.

**Critical decisions:**

- Step 2: picking \( N \) from Q
- Step 6: adding extensions of \( N \) to Q

---

**Why not a Visited list?**

- For the any-path algorithm, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with optimal searches can cause us to miss the best path.
Clearly, the shortest path (as determined by sum of link costs) to G is (S A D G) and an optimal search had better find it.

However, on expanding S, A and D are Visited, which means that the extension from A to D would never be generated and we would miss the best path. So, we can't use a Visited list; nevertheless, we still have the problem of multiple paths to a state leading to wasted work. We will deal with that issue later, since it can get a bit complicated. So, first, we will focus on the basic operation of optimal searches.

The first, and most basic, algorithm for optimal searching is called uniform-cost search. Uniform-cost is almost identical in implementation to best-first search. That is, we always pick the best node on Q to expand. The only, but crucial, difference is that instead of assigning the node value based on the heuristic value of the node's state, we will assign the node value as the "path length" or "path cost", a measure obtained by adding the "length" or "cost" of the links making up the path.

To reiterate, uniform-cost search uses the total length (or cost) of a path to decide which one to expand. Since we generally want the least-cost path, we will pick the node with the smallest path cost/length. By the way, we will often use the word "length" when talking about these types of searches, which makes intuitive sense when we talk about the pictures of graphs. However, we mean any cost measure (like length) that is positive and greater than 0 for the link between any two states.
The path length is the \( \text{SUM} \) of the length associated with the links in the path. For example, the path from \( S \) to \( A \) to \( C \) has total length 4, since it includes two links, each with edge 2.

The path from \( S \) to \( B \) to \( D \) to \( G \) has length 8 since it includes links of length 5 (\( S \)-\( B \)), 1 (\( B \)-\( D \)) and 2 (\( D \)-\( G \)).

Similarly for \( S \)-\( A \)-\( D \)-\( C \).

Given this, let’s simulate the behavior of uniform-cost search on this simple directed graph. As usual we start with a single node containing just the start state \( S \). This path has zero length. Of course, we choose this path for expansion.
This generates two new entries on Q: the path to A has length 2 and the one to B has length 5. So, we pick the path to A to expand.

This generates two new entries on the queue. The new path to C is the shortest path on Q, so we pick it to expand.

Since C has no descendants, we add no new paths to Q and we pick the best of the remaining paths, which is now the path to B.

The path to B is extended to D and G and the path to D from B is tied with the path to D from A. We are using order in Q to settle ties and so we pick the path from B to expand. Note that at this point G has been visited but not expanded.
Slide 2.4.19
Expanding D adds paths to C and G. Now the earlier path to D from A is the best pending path and we choose it to expand.

Slide 2.4.20
This adds a new path to G and a new path to C. The new path to G is the best on the Q (at least tied for best) so we pull it off.

Slide 2.4.21
And we have found our shortest path (S A D G) whose length is 8.

Slide 2.4.22
Note that once again we are not stopping on first visiting (placing on Q) the goal. We stop when the goal gets expanded (pulled off Q).
In uniform-cost search, it is imperative that we only stop when G is expanded and not just when it is visited. Until a path is first expanded, we do not know for a fact that we have found the shortest path to the state.

In the any-path searches we chose to do the same thing, but that choice was motivated at the time simply by consistency with what we HAVE to do now. In the earlier searches, we could have chosen to stop when visiting a goal state and everything would still work fine (actually better).

Note that the first path that visited G was not the eventually chosen optimal path to G. This explains our unwillingness to stop on first visiting G in the example we just did.

It is very important to drive home the fact that what uniform-cost search is doing (if we focus on the sequence of expanded paths) is enumerating the paths in the search tree in order of their path cost. The green numbers next to the tree on the left are the total path cost of the path to that state. Since, in a tree, there is a unique path from the root to any node, we can simply label each node by the length of that path.
Slide 2.4.27

So, for example, the trivial path from S to S is the shortest path.

Slide 2.4.28

Then the path from S to A, with length 2, is the next shortest path.

Slide 2.4.29

Then the path from S to A to C, with length 4, is the next shortest path.

Slide 2.4.30

Then comes the path from S to B, with length 5.
Followed by the path from S to A to D, with length 6.

And the path from S to B to D, also with length 6.

And, finally the path from S to A to D to G with length 8. The other path (S B D G) also has length 8.

This gives us the path we found. Note that the sequence of expansion corresponds precisely to path-length order, so it is not surprising we find the shortest path.
Now, we will turn our attention to what is probably the most popular search algorithm in AI, the A* algorithm. A* is an informed, optimal search algorithm. We will spend quite a bit of time going over A*; we will start by contrasting it with uniform-cost search.

Uniform-cost search as described so far is concerned only with expanding short paths; it pays no particular attention to the goal (since it has no way of knowing where it is). UC is really an algorithm for finding the shortest paths to all states in a graph rather than being focused in reaching a particular goal.

We can bias UC to find the shortest path to the goal that we are interested in by using a heuristic estimate of remaining distance to the goal. This, of course, cannot be the exact path distance (if we knew that we would not need much of a search); instead, it is a stand-in for the actual distance that can give us some guidance.
What we can do is to enumerate the paths by order of the SUM of the actual path length and the estimate of the remaining distance. Think of this as our best estimate of the TOTAL distance to the goal. This makes more sense if we want to generate a path to the goal preferentially to short paths away from the goal.

We call an estimate that always underestimates the remaining distance from any node an admissible (heuristic) estimate.

In order to preserve the guarantee that we will find the shortest path by expanding the partial paths based on the estimated total path length to the goal (like in UC without an expanded list), we must ensure that our heuristic estimate is admissible. Note that straight-line distance is always an underestimate of path-length in Euclidean space. Of course, by our constraint on distances, the constant function 0 is always admissible (but useless).

UC using an admissible heuristic is known as A* (A star). It is a very popular search method in AI.
Let's look at a quick example of the straight-line distance underestimate for path length in a graph. Consider the following simple graph, which we are assuming is embedded in Euclidean space, that is, think of the states as city locations and the length of the links are proportional to the driving distance between the cities along the best roads.

Then, we can use the straight-line (airline) distances (shown in red) as an underestimate of the actual driving distance between any city and the goal. The best possible driving distance between two cities cannot be better than the straight-line distance. But, it can be much worse.

Here we see that the straight-line estimate between B and G is very bad. The actual driving distance is much longer than the straight-line underestimate. Imagine that B and G are on different sides of the Grand Canyon, for example.

It may help to understand why an underestimate of remaining distance may help reach the goal faster to visualize the behavior of UC in a simple example.

Imagine that the states in a graph represent points in a plane and the connectivity is to nearest neighbors. In this case, UC will expand nodes in order of distance from the start point. That is, as time goes by, the expanded points will be located within expanding circular contours centered on the start point. Note, however, that points heading away from the goal will be treated just the same as points that are heading towards the goal.
Slide 2.5.12

If we add in an estimate of the straight-line distance to the goal, the points expanded will be bounded contours that keep constant the sum of the distance from the start and the distance to the goal, as suggested in the figure. What the underestimate has done is to "bias" the search towards the goal.

Slide 2.5.13

Let’s walk through an example of A*, that is, uniform-cost search using a heuristic which is an underestimate of remaining cost to the goal. In this example we are focusing on the use of the underestimate. The heuristic we will be using is similar to the earlier one but slightly modified to be admissible.

We start at S as usual.

Slide 2.5.14

And expand to A and B. Note that we are using the path length + underestimate and so the S-A path has a value of 4 (length 2, estimate 2). The S-B path has a value of 8 (5 + 3). We pick the path to A.

Slide 2.5.15

Expand to C and D and pick the path with shorter estimate, to C.
Slide 2.5.16

C has no descendants, so we pick the shorter path (to D).

Slide 2.5.17

Then a path to the goal has the best value. However, there is another path that is tied, the S-B path. It is possible that this path could be extended to the goal with a total length of 8 and we may prefer that path (since it has fewer states). We have assumed here that we will ignore that possibility, in some other circumstances that may not be appropriate.

Slide 2.5.18

So, we stop with a path to the goal of length 8.

Slide 2.5.19

It is important to realize that not all heuristics are admissible. In fact, the rather arbitrary heuristic values we used in our best-first example are not admissible given the path lengths we later assigned. In particular, the value for D is bigger than its distance to the goal and so this set of distances is not everywhere an underestimate of distance to the goal from every node. Note that the (arbitrary) value assigned for S is also an overestimate but this value would have no ill effect since at the time S is expanded there are no alternatives.
Although it is easy and intuitive to illustrate the concept of a heuristic by using the notion of straight-line distance to the goal in Euclidean space, it is important to remember that this is by no means the only example.

Take solving the so-called 8-puzzle, in which the goal is to arrange the pieces as in the goal state on the right. We can think of a move in this game as sliding the "empty" space to one of its nearest vertical or horizontal neighbors. We can help steer a search to find a short sequence of moves by using a heuristic estimate of the moves remaining to the goal.

One admissible estimate is simply the number of misplaced tiles. No move can get more than one misplaced tile into place, so this measure is a guaranteed underestimate and hence admissible.

We can do better if we note that, in fact, each move can at best decrease by one the "Manhattan" (aka Taxicab, aka rectilinear) distance of a tile from its goal.

So, the sum of these distances for each misplaced tile is also an underestimate. Note that it is always a better (larger) underestimate than the number of misplaced tiles. In this example, there are 7 misplaced tiles (all except tile 2), but the Manhattan distance estimate is 17 (4 for tile 1, 0 for tile 2, 2 for tile 3, 3 for tile 4, 1 for tile 5, 3 for tile 6, 1 for tile 7 and 3 for tile 8).