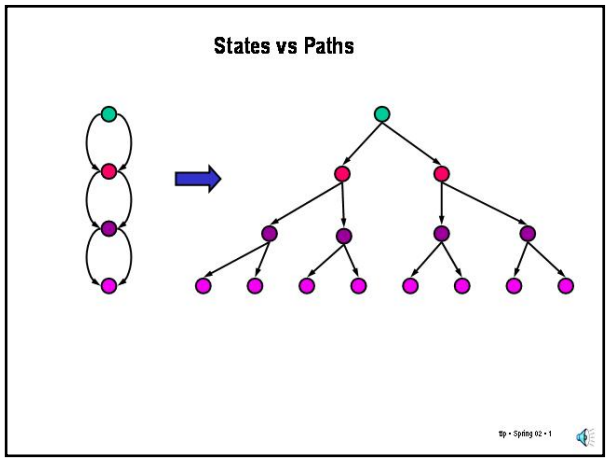
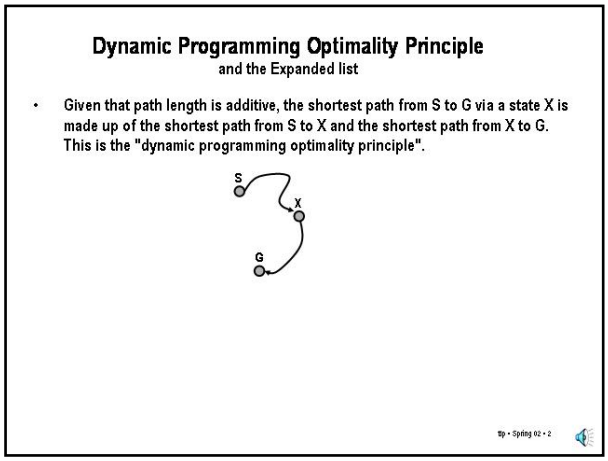
6.034 Notes: Section 2.6

Slide 2.6.1

In our discussion of uniform-cost search and A* so far, we have ignored the issue of revisiting states. We indicated that we could not use a Visited list and still preserve optimality, but can we use something else that will keep the worst-case cost of a search proportional to the number of states in a graph rather than to the number of non-looping paths? The answer is yes. We will start looking at uniform-cost search, where the extension is straightforward and then tackle A*, where it is not.



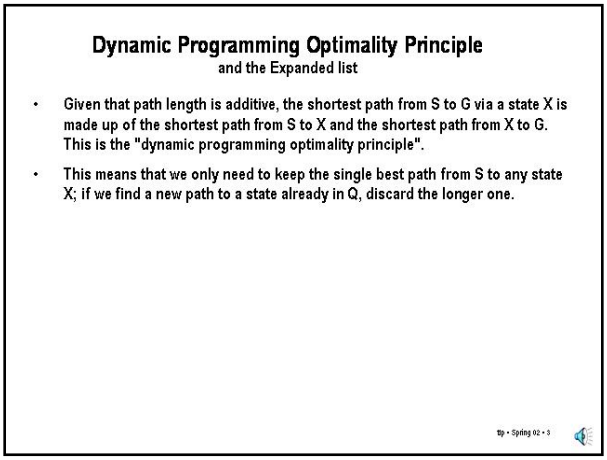


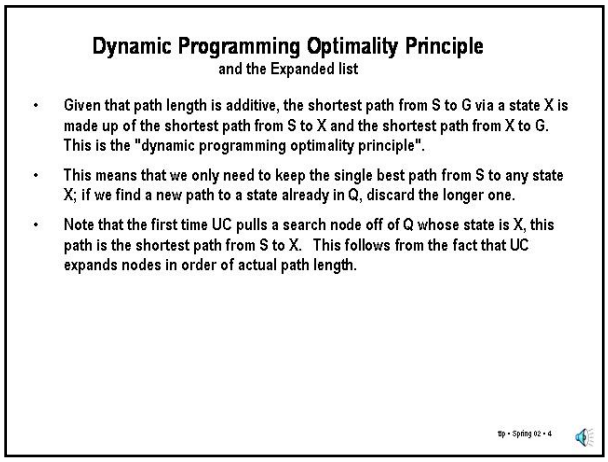
Slide 2.6.2

What will come to our rescue is the so-called "Dynamic Programming Optimality Principle", which is fairly intuitive in this context. Namely, the shortest path from the start to the goal that goes through some state X is made up of the shortest path to X followed by the shortest path from X to G. This is easy to prove by contradiction, but we won't do it here.

Slide 2.6.3

Given this, we know that there is no reason to compute any path except the shortest path to any state, since that is the only path that can ever be part of the answer. So, if we ever find a second path to a previously visited state, we can discard the longer one. So, when adding nodes to Q, check whether another node with the same state is already in Q and keep only the one with shorter path length.





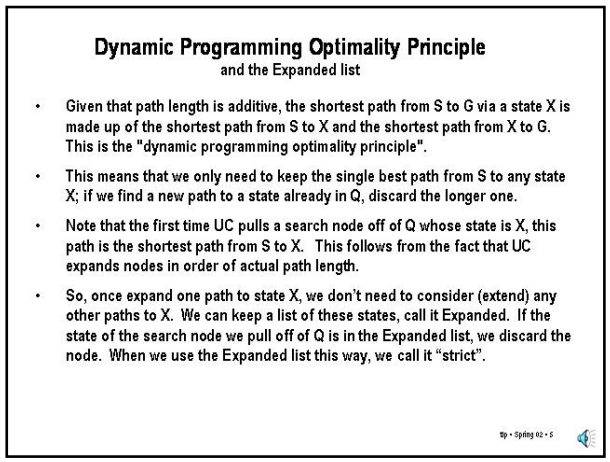
Slide 2.6.4

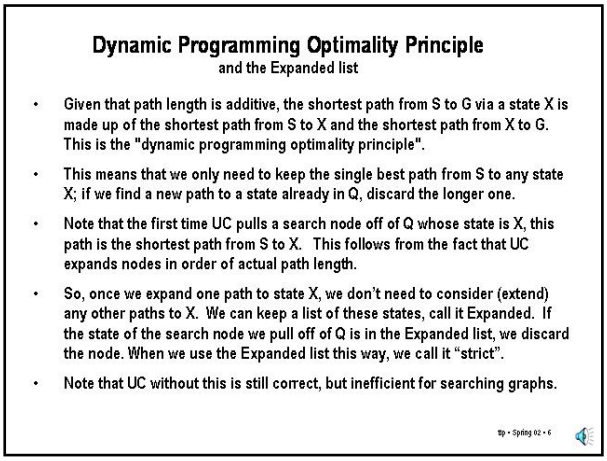
We have observed that uniform-cost search pulls nodes off Q (expands them) in order of their actual path length. So, the **first** time we expand a node whose state is X, that node represents the shortest path to that state. Any subsequent path we find to that state is a waste of effort, since it cannot have a shorter path.

Slide 2.6.5

So, let's remember the states that we have expanded already, in a "list" (or, better, a hash table) that we will call the Expanded list. If we try to expand a node whose state is already on the Expanded list, we can simply discard that path. We will refer to algorithms that do this, that is, no expanded state is re-visited, as using a **strict** Expanded list.

Note that when using a strict Expanded list, any visited state will either be in Q or in the Expanded list. So, when we consider a potential new node we can check whether (a) its state is in Q, in which case we accept it or discard it depending on the length of the new path versus the previous best, or (b) it is in Expanded, in which case we always discard it. If the node's state has never been visited, we add the node to Q.



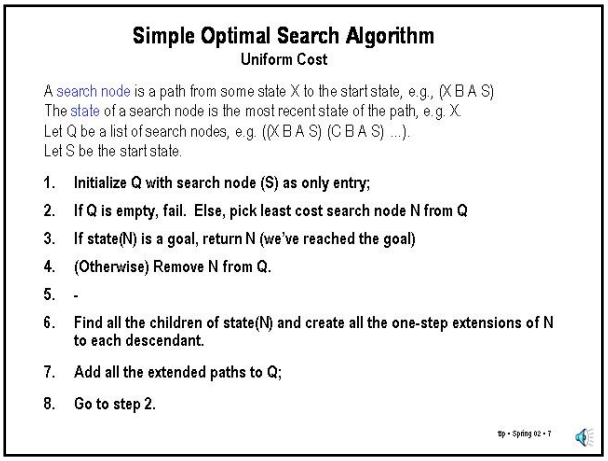


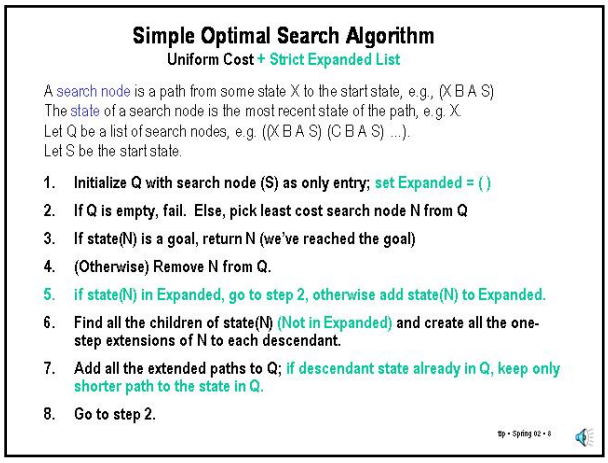
Slide 2.6.6

The correctness of uniform-cost search does not depend on using an expanded list or even on discarding longer paths to the same state (the Q will just be longer than necessary). We can use UC with or without these optimizations and it is still correct. Exploiting the optimality principle by discarding longer paths to states in Q and not re-visiting expanded states can, however, make UC much more efficient for densely connected graphs.

Slide 2.6.7

So, now, we need to modify our simple algorithm to implement uniform-cost search to take advantage of the Optimality Principle. We start with our familiar algorithm...



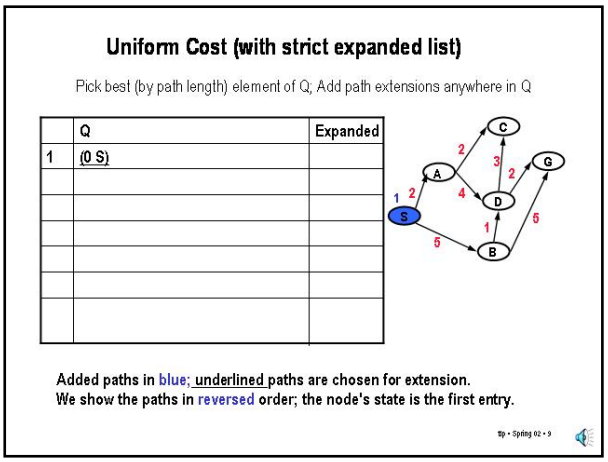


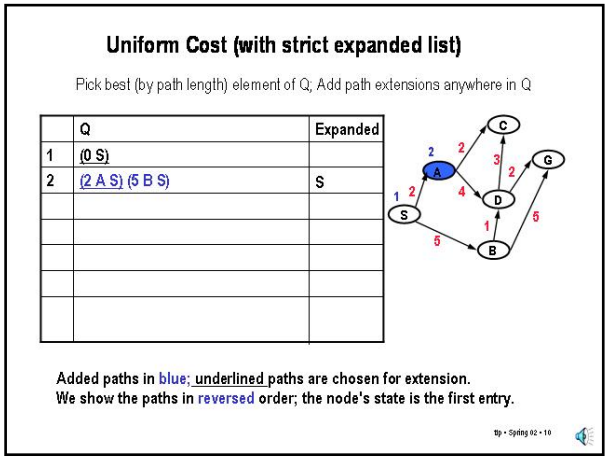
Slide 2.6.8

... and modify it. First we initialize the Expanded list in step 1. Since this is uniform-cost search, the algorithm picks the best element of Q, based on path length, in step 2. Then, in step 5, we check whether the state of the new node is on the Expanded list and if so, we discard it. Otherwise, we add the state of the new node to the Expanded list. In step 6, we avoid visiting nodes that are Expanded since that would be a waste of time. In step 7, we check whether there is a node in Q corresponding to each newly visited state, if so, we keep only the shorter path to that state.

Slide 2.6.9

Let's step through the operation of this algorithm on our usual example. We start with a node for S, having a 0-length path, as usual.



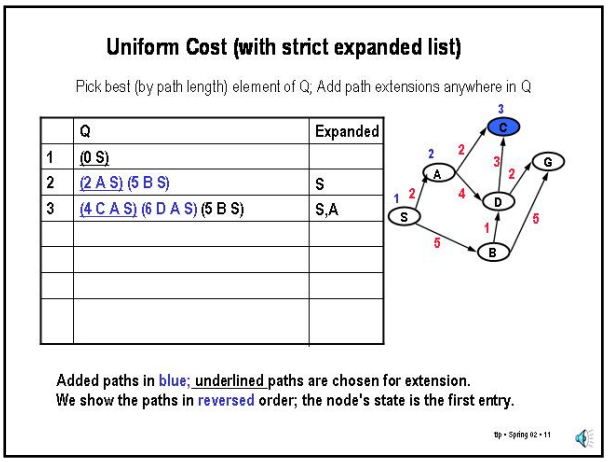


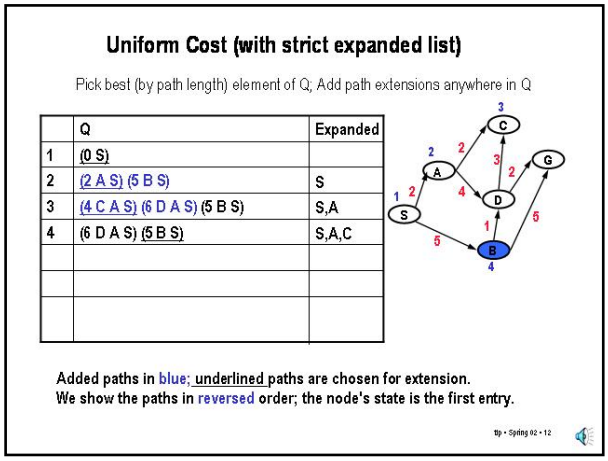
Slide 2.6.10

We expand the S node, add its descendants to Q and add the state S to the Expanded list.

Slide 2.6.11

We then pick the node at A to expand since it has the shortest length among the nodes in Q. We get the two extensions of the A node, which gives us paths to C and D. Neither of the two new nodes' states is already present in Q or in Expanded so we add them both to Q. We also add A to the Expanded list.



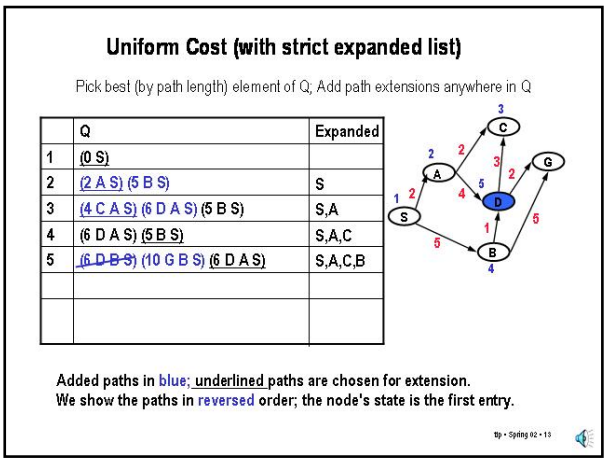


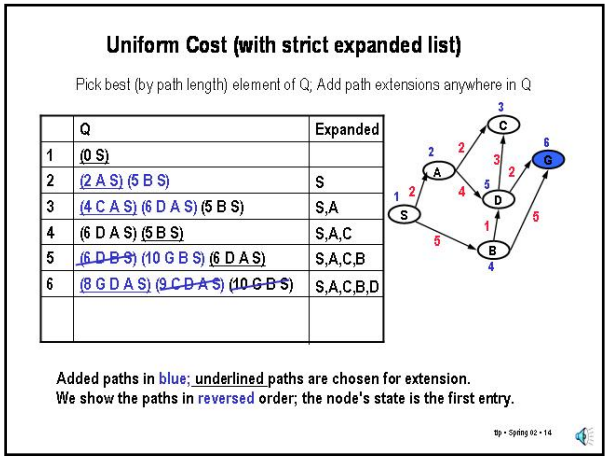
Slide 2.6.12

We pick the node at C to expand, but C has no descendants. So, we add C to Expanded but there are no new nodes to add to Q.

Slide 2.6.13

We select the node with the shortest path in Q, which is the node at B with path length 5 and generate the new descendant nodes, one to D and one to G. Note that at this point we have generated two paths to D - (S A D) and (S B D) both with length 6. We're free to keep either one but we do not need both. We will choose to discard the new node and keep the one already in Q.



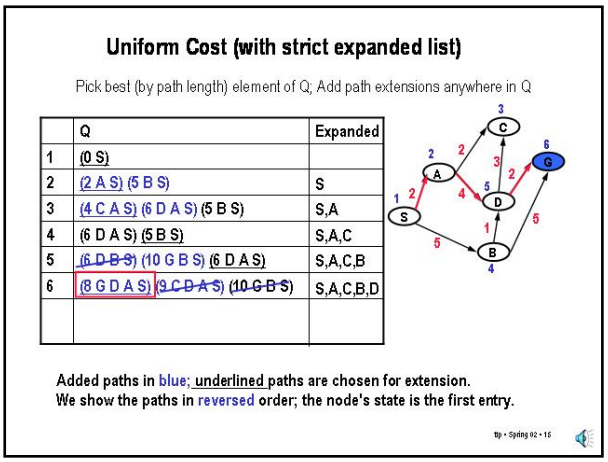


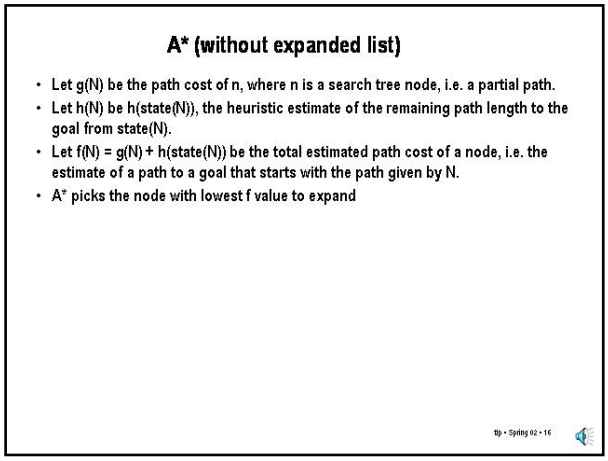
Slide 2.6.14

The node corresponding to the (S A D) path is now the shortest path, so we expand it and generate two descendants, one going to C and one going to G. The new C node can be discarded since C is on the Expanded list. The new G node shares its state with a node already on Q, but it corresponds to a shorter path - so we discard the older node in favor of the new one. So, at this point, Q only has one remaining node.

Slide 2.6.15

This node corresponds to the optimal path that is returned. It is easy to show that the use of an Expanded list, as well as keeping only the shortest path to any state in Q, preserve the optimality guarantee of uniform-cost search and can lead to substantial performance improvements. Will this hold true for A* as well?



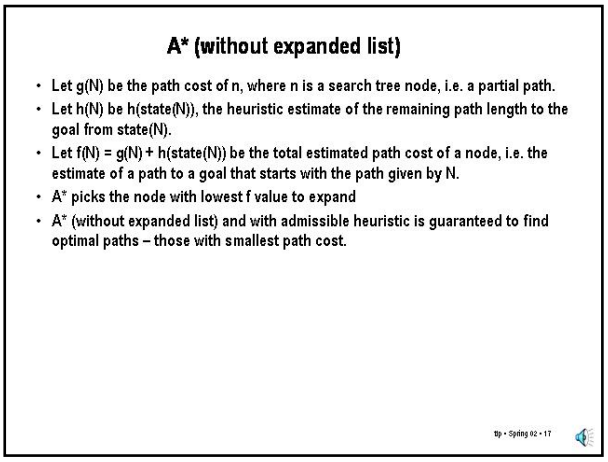


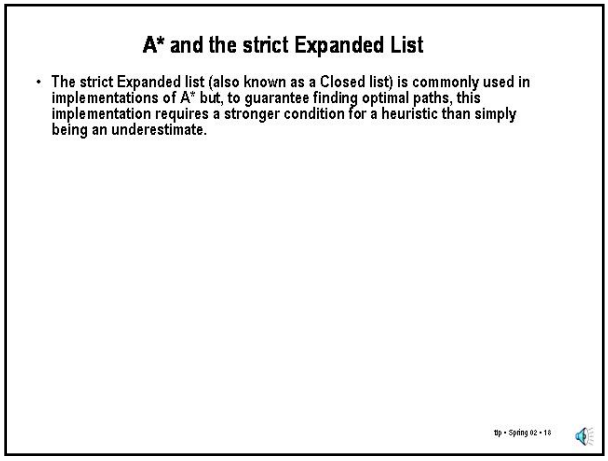
Slide 2.6.16

First, let's review A* and the notation that we have been using. The important notation to remember is that the function g represents actual path length along a partial path to a node's state. The function h represents the heuristic value at a node's state and f is the total estimated path length (to a goal) and is the sum of the actual length (g) and the heuristic estimate (h). A* picks the node with the smallest value of f to expand.

Slide 2.6.17

A*, without using an Expanded list or discarding nodes in Q but using an admissible heuristic -- that is, one that underestimates the distance to the goal -- is guaranteed to find optimal paths.



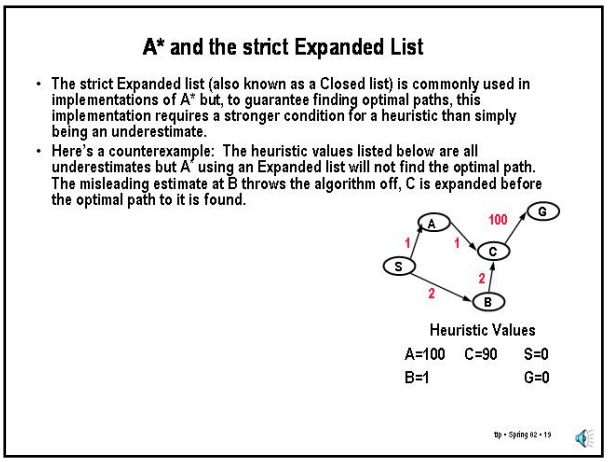


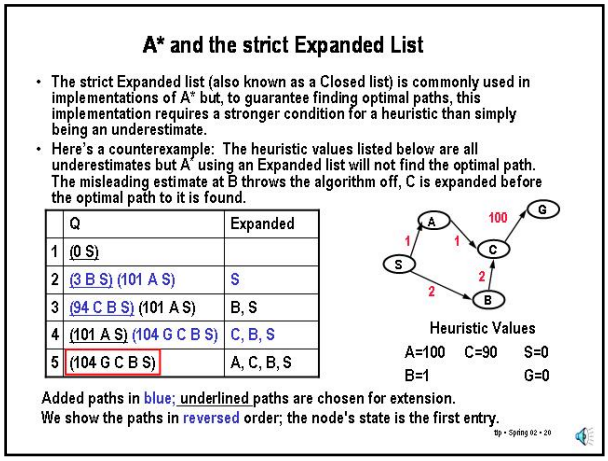
Slide 2.6.18

If we use the search algorithm we used for uniform-cost search with a strict Expanded list for A*, adding in an admissible heuristic to the path length, then we can no longer guarantee that it will always find the optimal path. We need a stronger condition on the heuristics used than being an underestimate.

Slide 2.6.19

Here's an example that illustrates this point. The exceedingly optimistic heuristic estimate at B "lures" the A* algorithm down the wrong path.



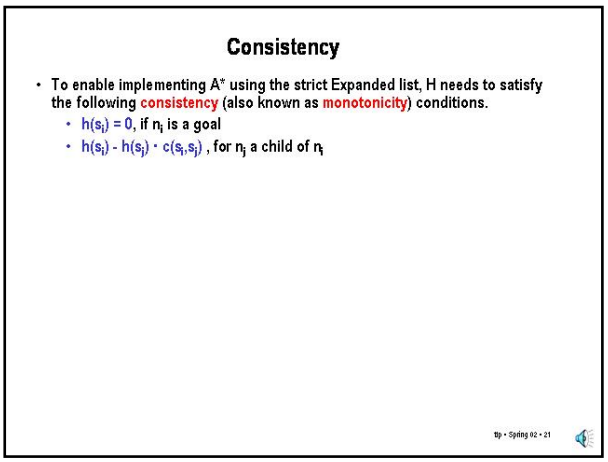


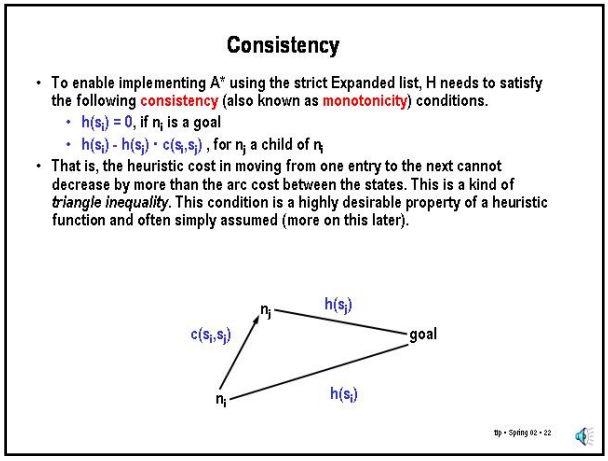
Slide 2.6.20

You can see the operation of A* in detail here, confirming that it finds the incorrect path. The correct partial path via A is blocked when the path to C via B is expanded. In step 4, when A is finally expanded, the new path to C is not put on Q, because C has already been expanded.

Slide 2.6.21

The stronger conditions on a heuristic that enables us to implement A* just the same way we implemented uniform-cost search with a strict Expanded list are known as the **consistency** conditions. They are also called monotonicity conditions by others. The first condition is simple, namely that goal states have a heuristic estimate of zero, which we have already been assuming. The next condition is the critical one. It indicates that the difference in the heuristic estimate between one state and its descendant must be less than or equal to the actual path cost on the edge connecting them.



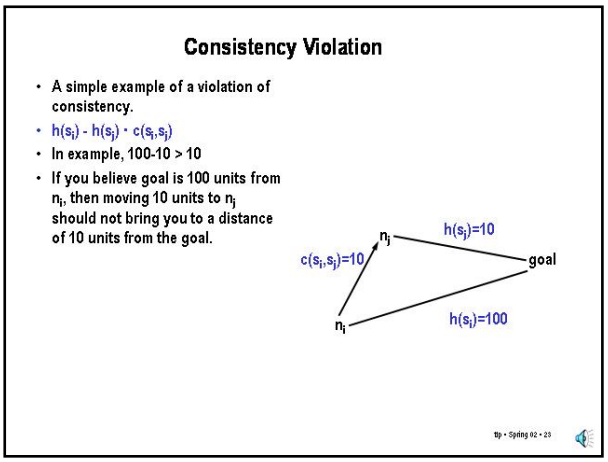


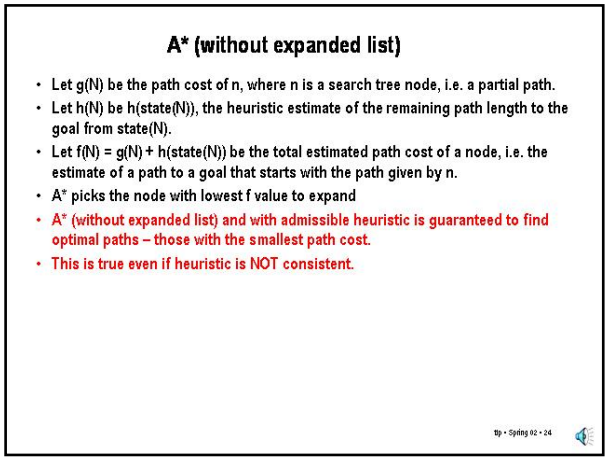
Slide 2.6.22

The best way of visualizing the consistency condition is as a "triangle inequality", that is, one side of the triangle is less than or equal the sum of the other two sides, as seen on the diagram here.

Slide 2.6.23

Here is a simple example of a (gross) violation of consistency. If you believe goal is 100 units from n_i, then moving 10 units to n_j should not bring you to a distance of 10 units from the goal. These heuristic estimates are not consistent.



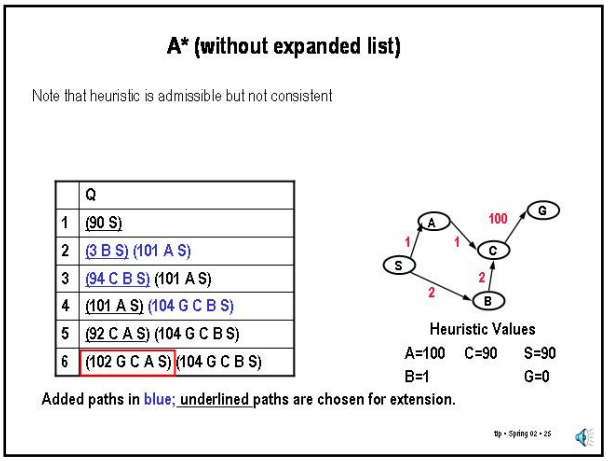


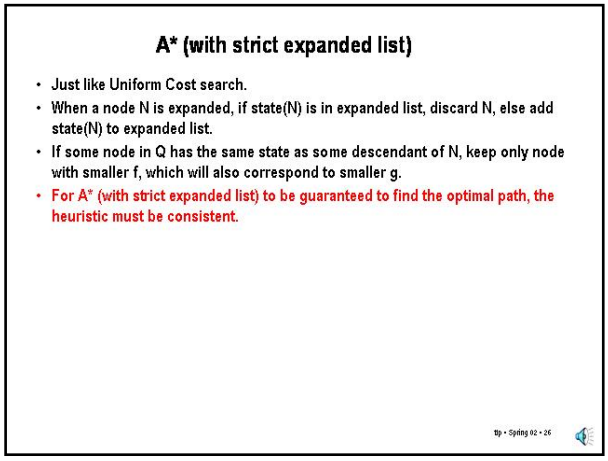
Slide 2.6.24

I want to stress that consistency of the heuristic is only necessary for optimality when we want to discard paths from consideration, for example, because a state has already been expanded. Otherwise, plain A* without using an expanded only requires only that the heuristic be admissible to guarantee optimality.

Slide 2.6.25

This illustrates that A* without an Expanded list has no trouble coping with the example we saw earlier that showed the pitfalls of using a strict Expanded list. This heuristic is not consistent but it is an underestimate and that is all that is needed for A* without an Expanded list to guarantee optimality.



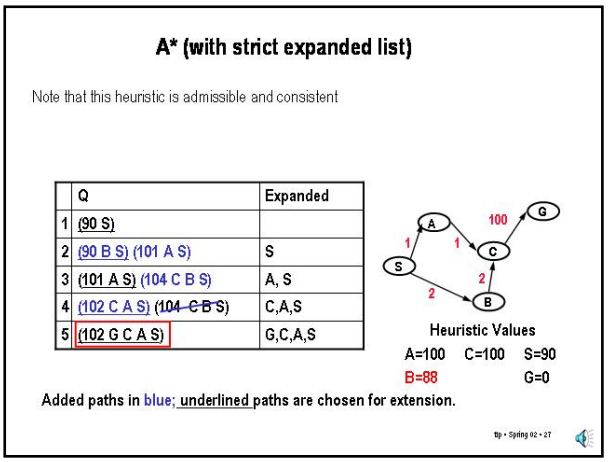


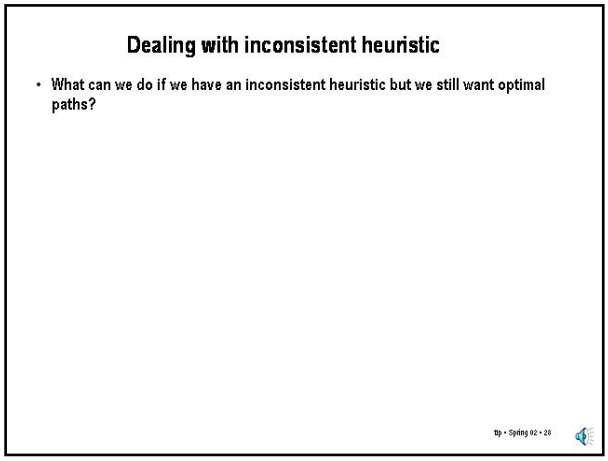
Slide 2.6.26

The extension of A* to use a strict expanded list is just like the extension to uniform-cost search. In fact, it is the identical algorithm except that it uses f values instead of g values. But, we stress that for this algorithm to guarantee finding optimal paths, the heuristic must be consistent.

Slide 2.6.27

If we modify the heuristic in the example we have been considering so that it is consistent, as we have done here by increasing the value of h(B), then A* (even when using a strict Expanded list) will work.





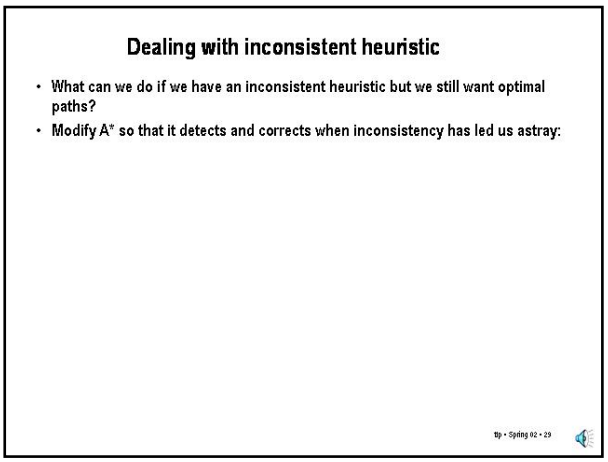
Slide 2.6.28

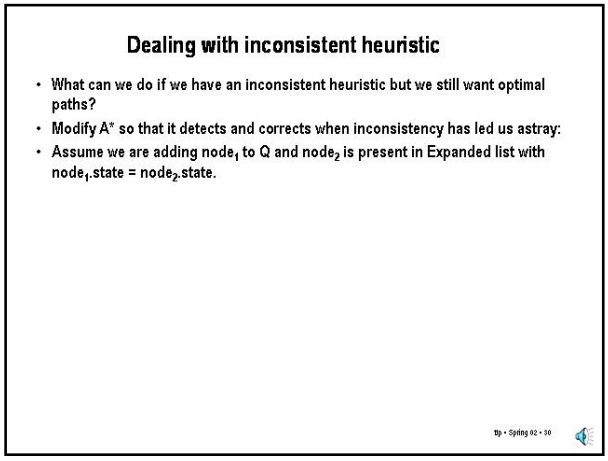
People sometimes simply assume that the consistency condition holds and implement A* with a strict Expanded list (also called a Closed list) in the simple way we have shown before. But, this is not the only (or best) option. Later we will see that A* can be adapted to retain optimality in spite of a heuristic that is not consistent - there will be a performance price to be paid however.

Slide 2.6.29

The key step needed to enable A* to cope with inconsistent heuristics is to detect when an overly optimistic heuristic estimate has caused us to expand a node prematurely, that is, before the shortest path to that node has been found. This is basically analogous to what we have been doing when we find a shorter path to a state already in Q, except we need to do it to states in the Expanded list. In this modified algorithm, the use of the Expanded list is not strict: we allow re-visiting states on the Expanded list.

To implement this, we will keep in the Expanded list not just the expanded states but the actual node that was expanded. In particular, this records the actual path length at the time of expansion





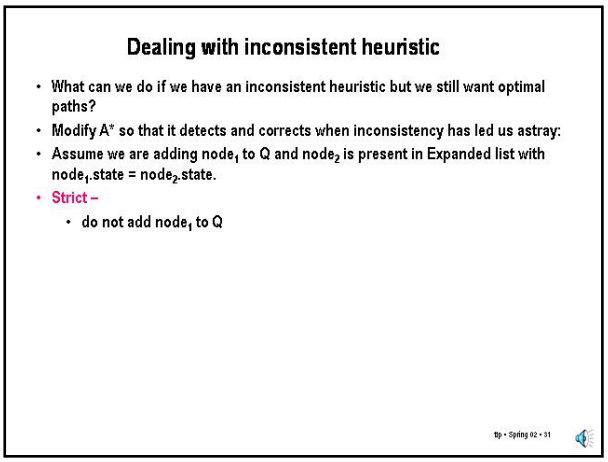
Slide 2.6.30

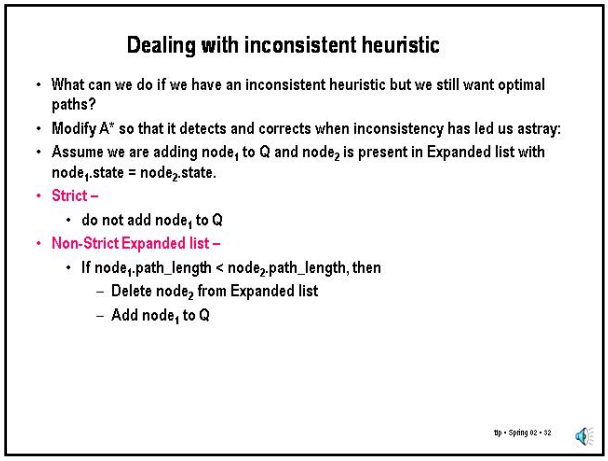
Let's consider in detail the operation of the Expanded list if we want to handle inconsistent heuristics while guaranteeing optimal paths.

Assume that we are adding a node, call it $node_1$, to Q when using an Expanded list. So, we check to see if a node with the same state is present in the Expanded list and we find $node_2$ which matches.

Slide 2.6.31

With a strict Expanded list, we simply discard $node_1$; we do not add it to Q.





Slide 2.6.32

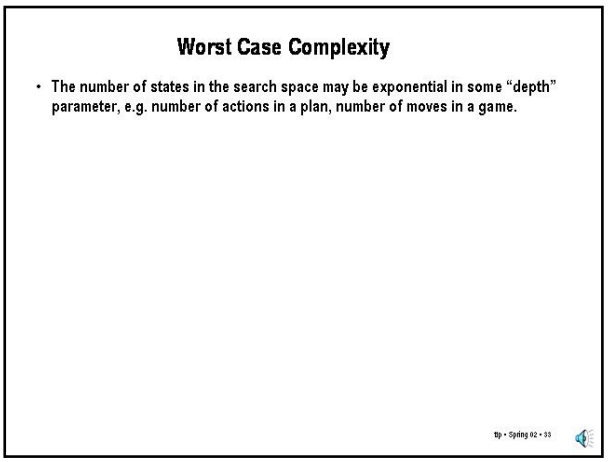
With a non-strict Expanded list, the situation is a bit more complicated. We want to make sure that $node_1$ has not found a better path to the state than $node_2$. If a better path has been found, we remove the old node from Expanded (since it does not represent the optimal path) and add the new node to Q.

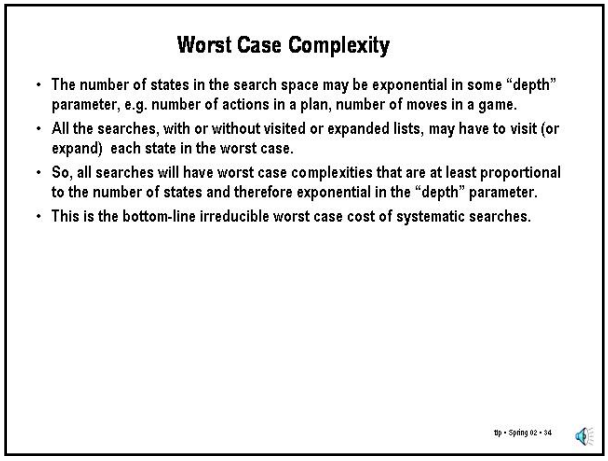
Slide 2.6.33

Let's think a bit about the worst case complexity of A*, in terms of the number of nodes expanded (or visited).

As we've mentioned before, it is customary in AI to think of search complexity in terms of some "depth" parameter of the domain such as the number of steps in a plan of action or the number of moves in a game. The state space for such domains (planning or game playing) grows exponentially in the "depth", that is, because at each depth level there is some branching factor (e.g., the possible actions) and so the number of states grows exponentially with the depth.

We could equally well speak instead of the number of states as a fixed parameter, call it N, and state our complexity in terms of N. We just have to keep in mind then that in many applications, N grows exponentially with respect to the depth parameter.



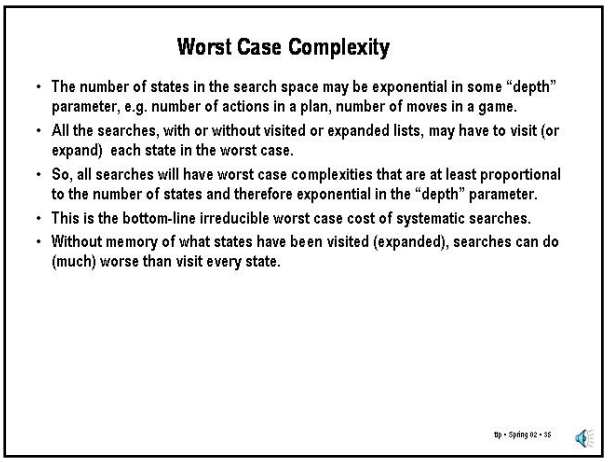


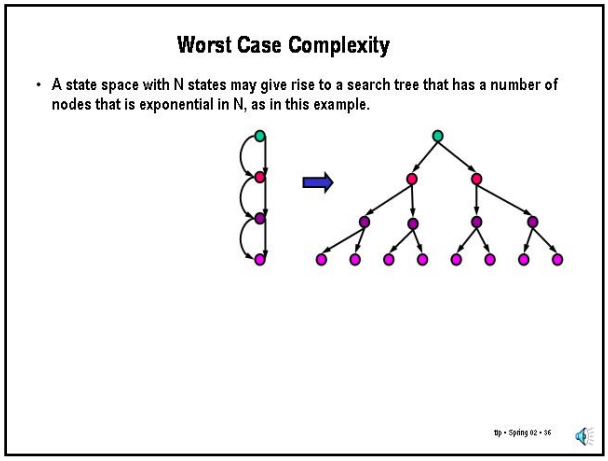
Slide 2.6.34

In the worst case, when the heuristics are not very useful or the nodes are arranged in the worst possible way, all the search methods may end up having to visit or expand all of the states (up to some depth). In practice, we should be able to avoid this worst case but in many cases one comes pretty close.

Slide 2.6.35

The problem is that if we have no memory of what states we've visited or expanded, then the worst case for a densely connected graph can be much, much worse than this. One may end up doing exponentially more work.



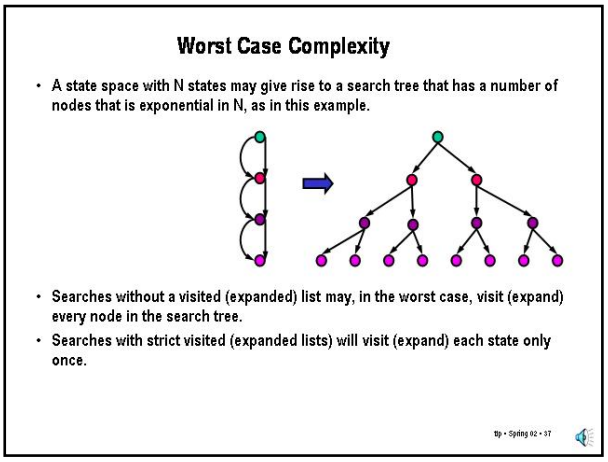


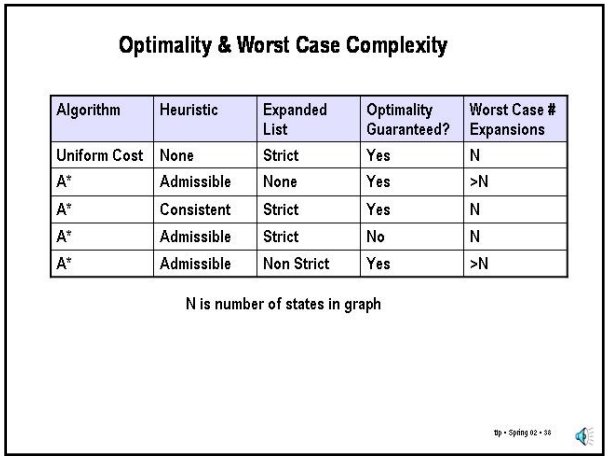
Slide 2.6.36

We've seen this example before. It shows that a state space with N states can generate a search tree with 2^N nodes.

Slide 2.6.37

A search algorithm that does not keep a visited or expanded list will do exponentially more work that necessary. On the other hand, if we use a strict expanded list, we will never expand more than the (unavoidable) N states.



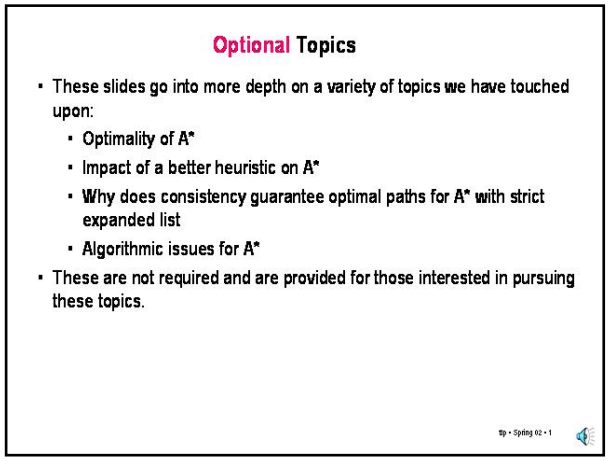


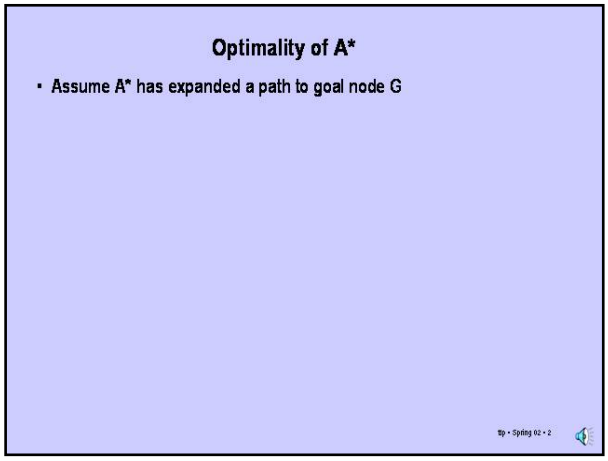
Here we summarize the optimality and complexity of the various algorithms we have been examining.

6.034 Notes: Section 2.7

Slide 2.7.1

This set of slides goes into more detail on some of the topics we have covered in this chapter.





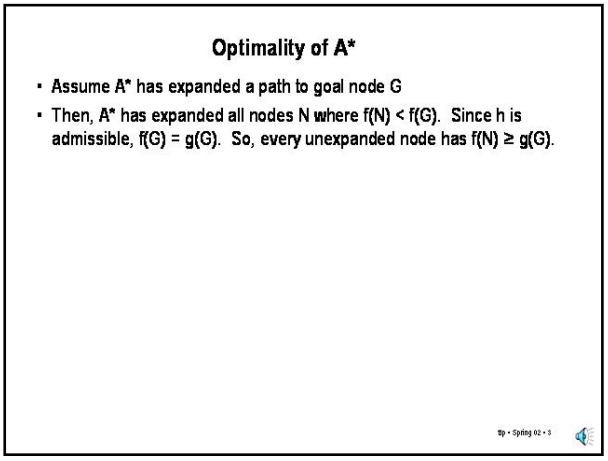
Slide 2.7.2

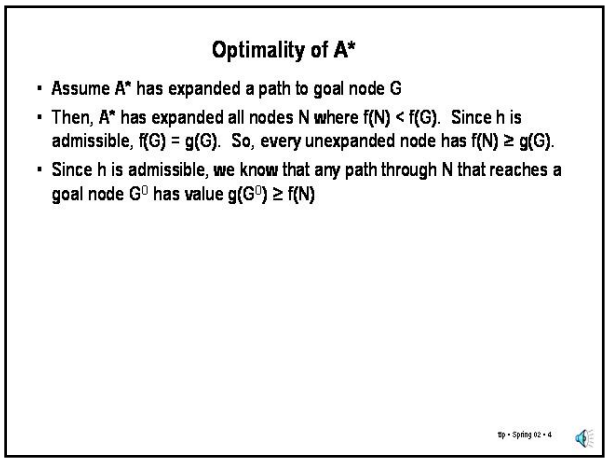
First topic:

Let's go through a quick proof that A* actually finds the optimal path. Start by assuming that A* has selected a node G.

Slide 2.7.3

Then, we know from the operation of A* that it has expanded all nodes N whose cost f(N) is strictly less than the cost of G. We also know that since the heuristic is admissible, its value at a goal node must be 0 and thus, f(G) = g(G)+h(G) = g(G). Therefore, every unexpanded node N must have f(N) greater or equal to the actual path length to G.



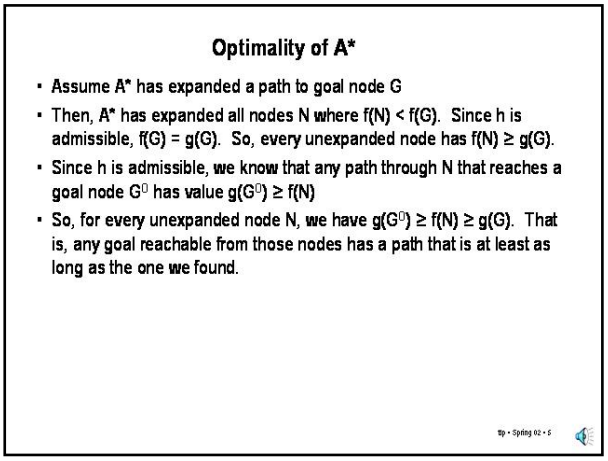


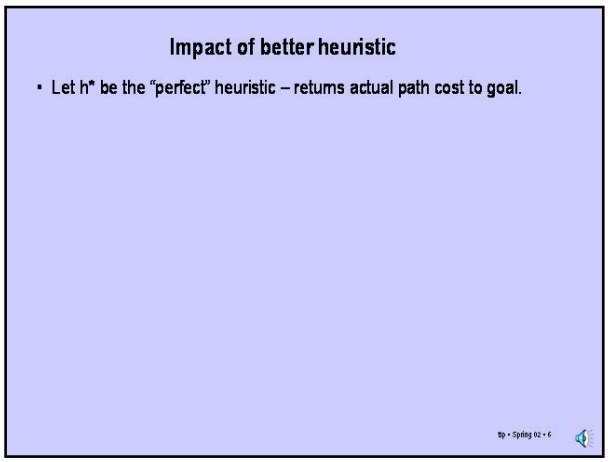
Slide 2.7.4

Since h is admissible, we know that any path through an unexpanded node N that reaches some alternate goal node G' must have a total cost estimate f(N) that is not larger than the actual cost to G', that is, g(G').

Slide 2.7.5

Combining these two statements we see that the path length to any other goal node G' must be greater or equal to the path length of the goal node A* found, that is, G.





Slide 2.7.6

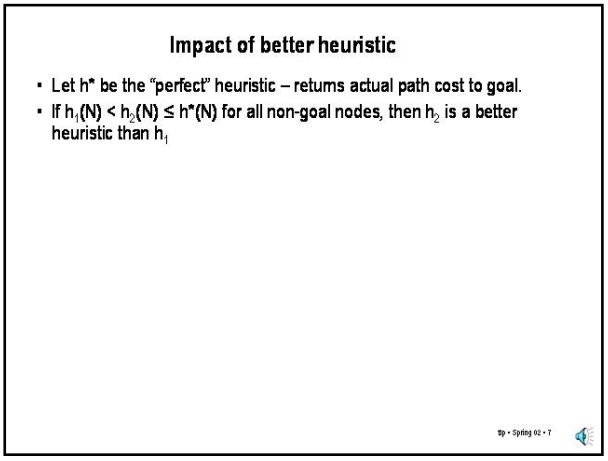
Next topic:

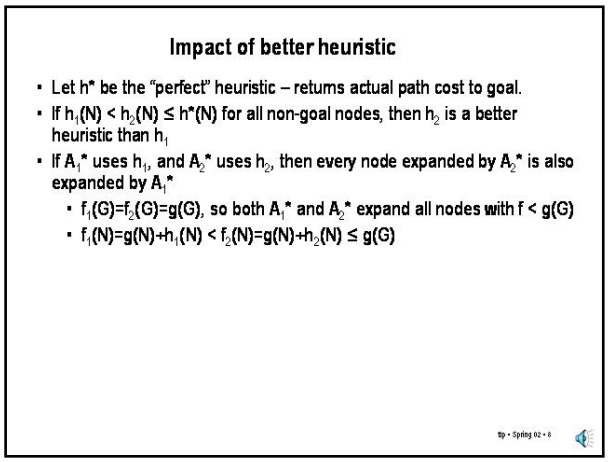
We can also show that a better heuristic in general leads to improved performance of A* (or at least no decrease). By performance, we mean number of nodes expanded. In general, there is a tradeoff in how much effort we do to compute a better heuristic and the improvement in the search time due to reduced number of expansions.

Let's postulate a "perfect" heuristic which computes the actual optimal path length to a goal. Call this heuristic h*.

Slide 2.7.7

Then, assume we have a heuristic h_1 that is always numerically less than another heuristic h_2, which is (by admissibility) less than or equal to h*.





Slide 2.7.8

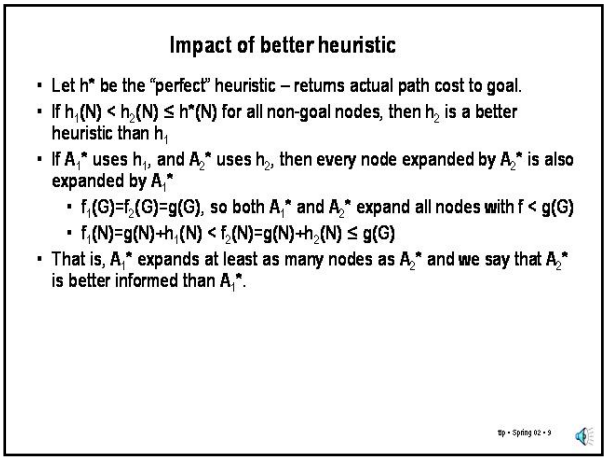
The key observation is that if we have two versions of A*, one using h_1 and the other using h_2, then every node expanded by the second one is also expanded by the first.

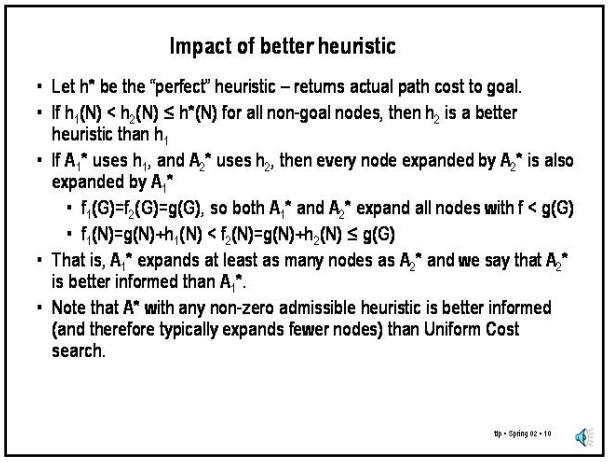
This follows from the observation we have made earlier that at a goal, the heuristic estimates all agree (they are all 0) and so we know that both versions will expands all nodes whose value of f is less than the actual path length of G.

Now, every node expanded by A^*_2, will have a path cost no greater than the actual cost to the goal G. Such a node will have a smaller cost using h_1 and so it will definitely be expanded by A^*_1 as well.

Slide 2.7.9

So, A^*_1 expands at least as many nodes as A^*_2. We say that **A^*_2 is better informed than A^*_1** to refer to this situation.





Slide 2.7.10

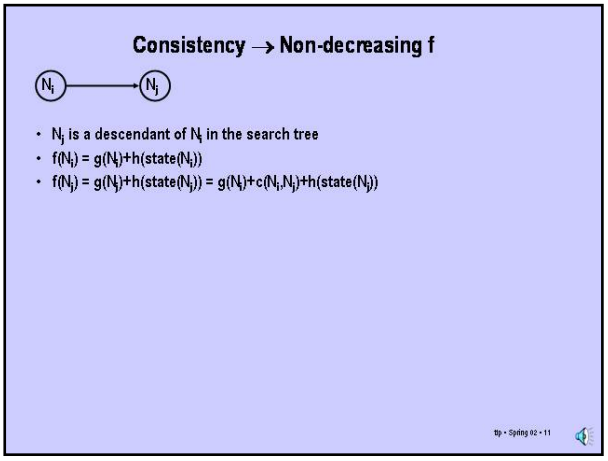
Since uniform-cost search is simply A* with a heuristic of 0, we can say that A* is generally better informed than UC and we expect it to expand fewer nodes. But, A* will expend additional effort computing the heuristic value -- a good heuristic can more than pay back that extra effort.

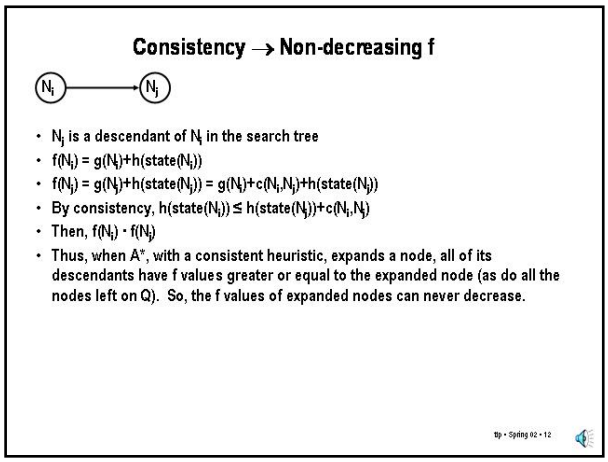
Slide 2.7.11

New topic:

Why does consistency allow us to guarantee that A* will find optimal paths? The key insight is that consistency ensures that the f values of expanded nodes will be non-decreasing over time.

Consider two nodes N_i and N_j such that the latter is a descendant of the former in the search tree. Then, we can write out the values of f as shown here, involving the actual path length $g(N_i)$, the cost of the edge between the nodes $c(N_i, N_j)$ and the heuristic values of the two corresponding states.





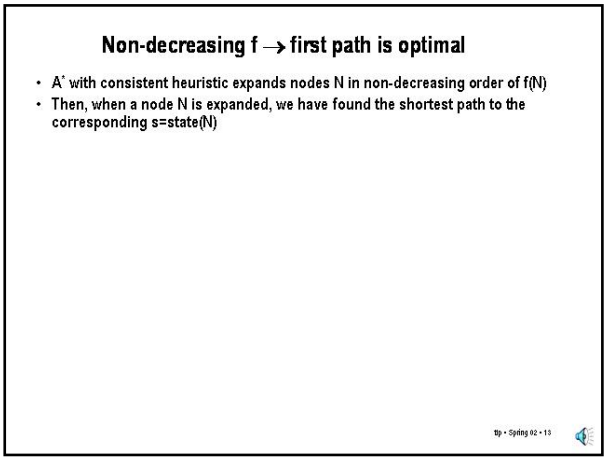
Slide 2.7.12

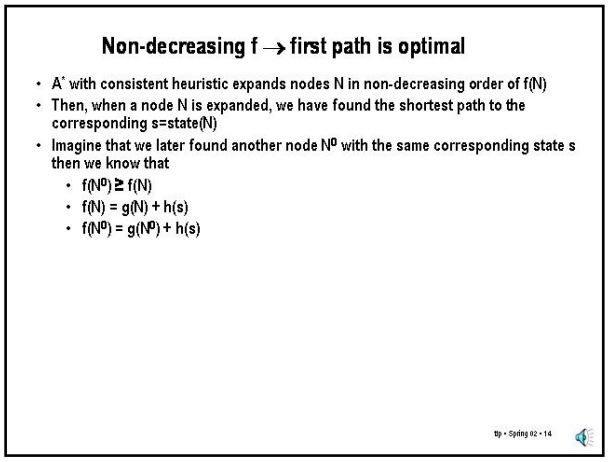
By consistency of the heuristic estimates, we know that the heuristic estimate cannot decrease more than the edge cost. So, the value of f in the descendant node cannot go down; it must stay the same or go up.

By this reasoning we can conclude that whenever A* expands a node, the new nodes' f values must be greater or equal to that of the expanded node. Also, since the expanded node must have had an f value that was a minimum of the f values in Q, this means that no nodes in Q after this expansion can have a lower f value than the most recently expanded node. That is, if we track the series of f values of expanded nodes over time, this series is non-decreasing.

Slide 2.7.13

Now we can show that if we have nodes expanded in non-decreasing order of f, then the first time we expand a node whose state is s, then we have found the optimal path to the state. If you recall, this was the condition that enabled us to use the strict Expanded list, that is, we never need to re-visit (or re-expand) a state.



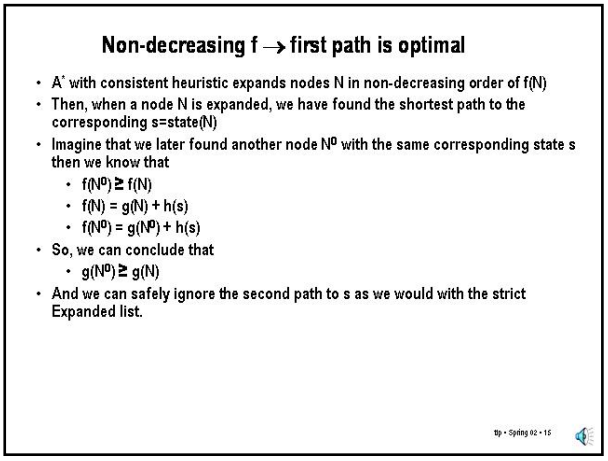


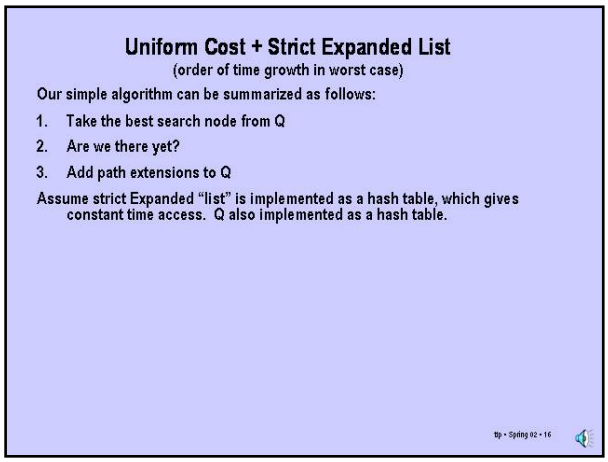
Slide 2.7.14

To prove this, let's assume that we later found another node N' that corresponds to the same state as a previously expanded node N. We have shown that the f value of N' is greater or equal that of N. But, since the heuristic values of these nodes must be the same - since they correspond to the same underlying graph state - the difference in f values must be accounted by a difference in actual path length.

Slide 2.7.15

So, we can conclude that the second path cannot be shorter than the first path we already found, and so we can ignore the new path!





Slide 2.7.16

Final topic:

Let's analyze the behavior of uniform-cost search with a strict Expanded List. This algorithm is very similar to the well known Dijkstra's algorithm for shortest paths in a graph, but we will keep the name we have been using. This analysis will apply to A* with a strict Expanded list, since in the worst case they are the same algorithm.

To simplify our approach to the analysis, we can think of the algorithm as boiled down to three steps.

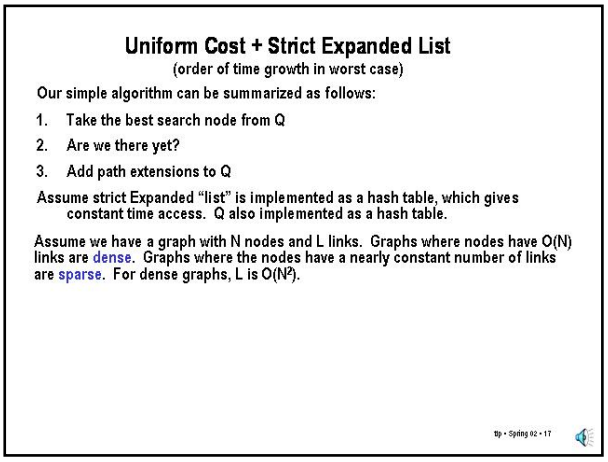
1. Pulling paths off of Q,
2. Checking whether we are done and
3. Adding the relevant path extensions to Q.

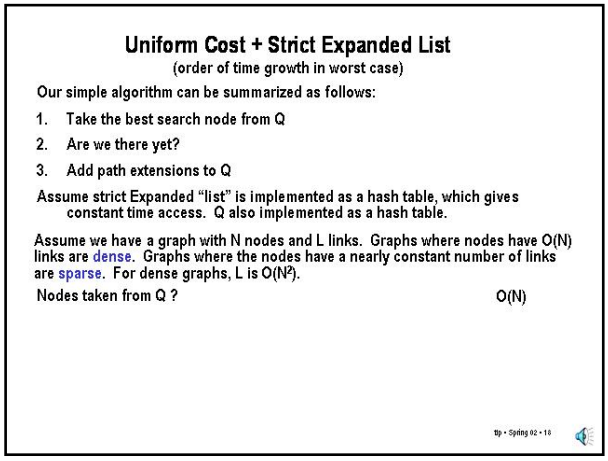
In what follows, we assume that the Expanded list is not a "real" list but some constant-time way of checking that a state has been expanded (e.g., by looking at a mark on the state or via a hash-table).

We also assume that Q is implemented as a hash table, which has constant time access (and insertion) cost. This is so we can find whether a node with a given state is already on Q.

Slide 2.7.17

Later, it will become important to distinguish the case of "sparse" graphs, where the states have a nearly constant number of neighbors and "dense" graphs where the number of neighbors grows with the number of states. In the dense case, the total number of edges is $O(N^2)$, which is substantial.



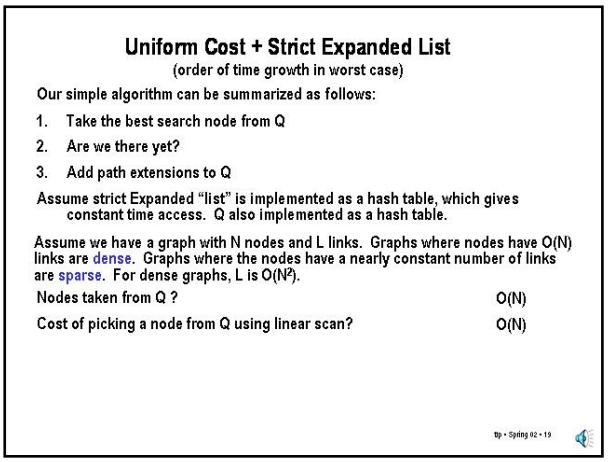


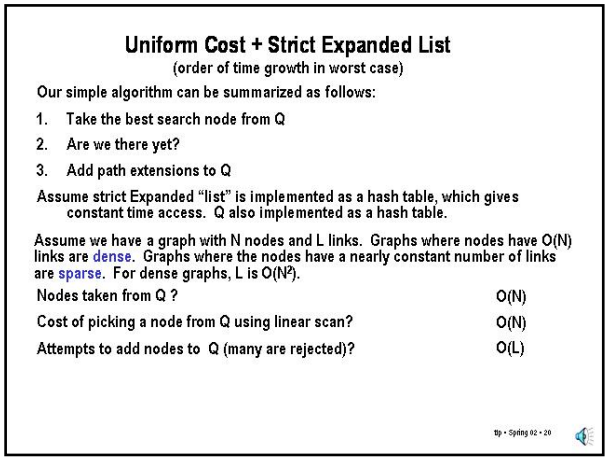
Slide 2.7.18

So, let's ask the question, how many nodes are taken from Q (expanded) over the life of the algorithm (in the worst case)? Here we assume that when we add a node to Q, we check whether a node already exists for that state and keep only the node with the shorter path. Given this and the use of a strict Expanded list, we know that the worst-case number of expansions is N, the total number of states.

Slide 2.7.19

What's the cost of expanding a node? Assume we scan Q to pick the best paths. Then the cost is of the order of the number of paths in Q, which is O(N) also, since we only keep the best path to a state.



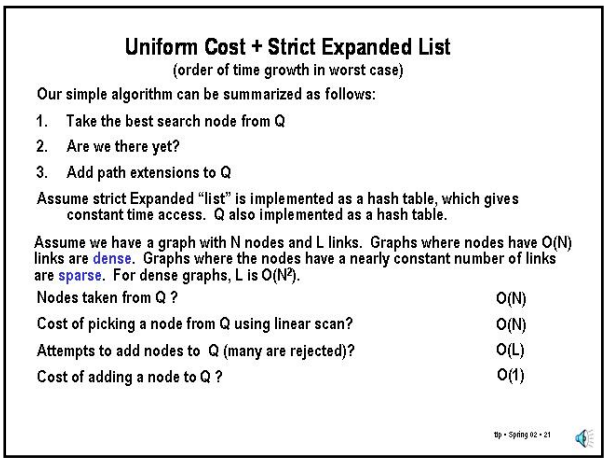


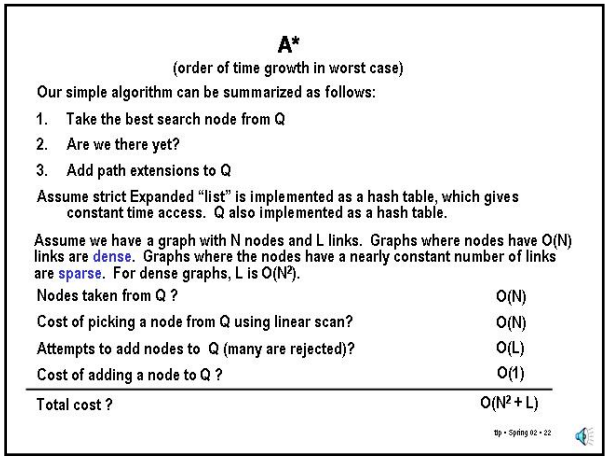
Slide 2.7.20

How many times do we (attempt to) add paths to Q? Well, since we expand every state at most once and since we only add paths to direct neighbors (links) of that state, then the total number is bounded by the total number of links in the graph.

Slide 2.7.21

Adding to the Q, assuming it is a hash table, as we have been assuming here, can be done in constant time.



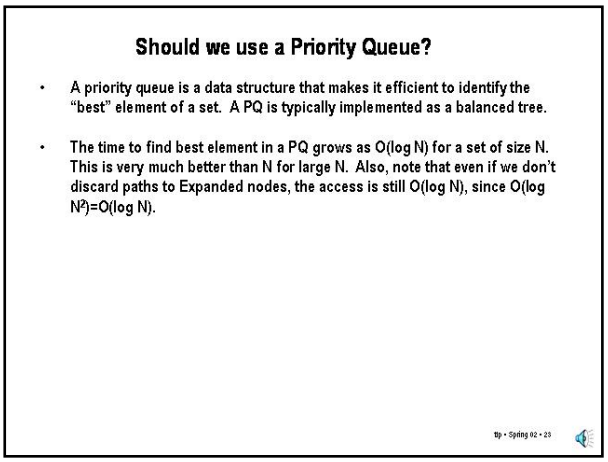


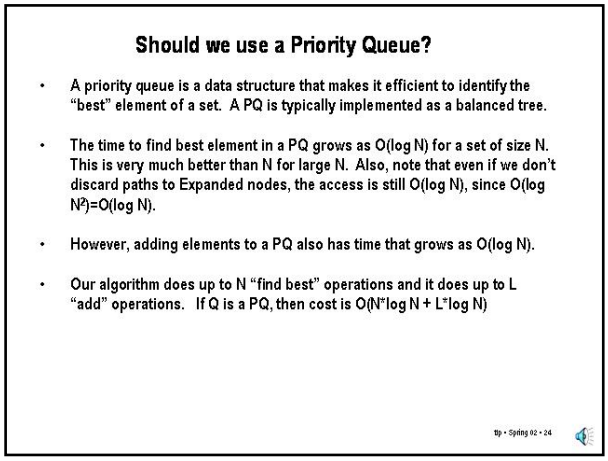
Slide 2.7.22

Putting it all together gives us a total cost on the order of $O(N^2+L)$ which, since L is at worst $O(N^2)$ is essentially $O(N^2)$.

Slide 2.7.23

If you know about priority queues, you might think that they are natural as implementation of Q, since one can efficiently find the best element in such a queue.



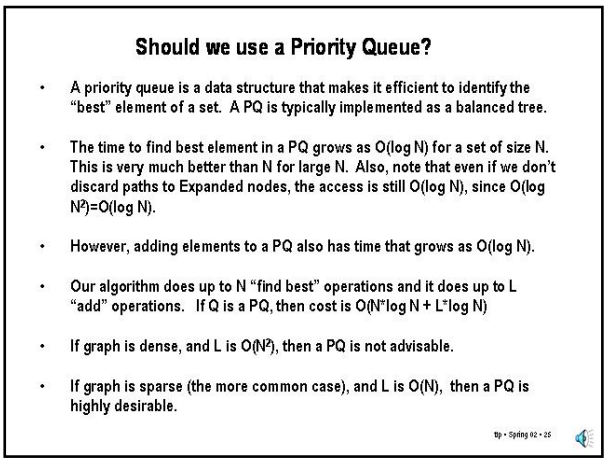


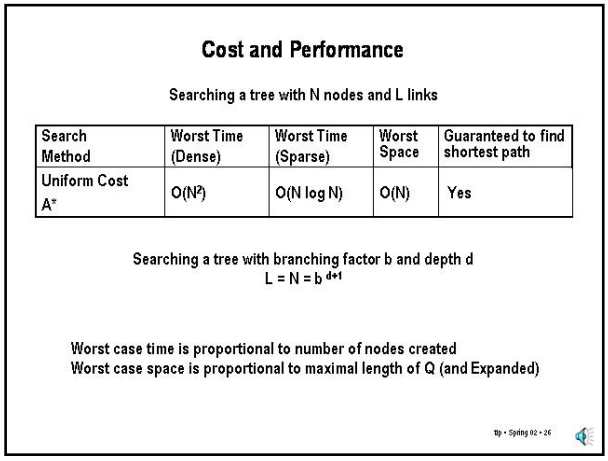
Slide 2.7.24

Note, however, that adding elements to such a Q is more expensive than adding elements to a list or a hash table. So, whether it's worth it depends on how many additions are done. As we said, this is order of L, the number of links.

Slide 2.7.25

For a dense graph, where L is $O(N^2)$, then the priority queue will not be worth it. But, for a sparse graph it will.





Slide 2.7.26

Here we summarize the worst-case performance of UC (and A*, which is the same). Note, however, that we expect A* with a good heuristic to outperform UC in practice since it will expand at most as many nodes as UC. The worst case cost (with an uninformative heuristic) remains the same.

By the way, in talking about space we have focused on the number of entries in Q but have not mentioned the length of the paths. One might think that this would actually be the dominant factor. But, recall that we are unrolling the graph into the search tree and each node only needs to have a link to its unique ancestor in the tree and so a node really requires constant space.

As before, you can think of the performance of these algorithms as a low-order polynomial (N^2) or as an intractable exponential, depending on how one describes the search space.