Search plays a key role in many parts of AI. These algorithms provide the conceptual backbone of almost every approach to the systematic exploration of alternatives.

We will start with some background, terminology and basic implementation strategies and then cover four classes of search algorithms, which differ along two dimensions: First, is the difference between uninformed (also known as blind) search and then informed (also known as heuristic) searches. Informed searches have access to task-specific information that can be used to make the search process more efficient. The other difference is between any path searches and optimal searches. Optimal searches are looking for the best possible path while any-path searches will just settle for finding some solution.

The search methods we will be dealing with are defined on trees and graphs, so we need to fix on some terminology for these structures:

- A tree is made up of nodes and links (circles and lines) connected so that there are no loops (cycles). Nodes are sometimes referred to as vertices and links as edges (this is more common in talking about graphs).
- A tree has a root node (where the tree “starts”). Every node except the root has a single parent (aka direct ancestor). More generally, an ancestor node is a node that can be reached by repeatedly going to a parent node. Each node (except the terminal (aka leaf) nodes) has one or more children (aka direct descendants). More generally, a descendant node is a node that can be reached by repeatedly going to a child node.

A graph is also a set of nodes connected by links but where loops are allowed and a node can have multiple parents. We have two kinds of graphs to deal with: directed graphs, where the links have direction (akin to one-way streets).
And, **undirected** graphs where the links go both ways. You can think of an undirected graph as shorthand for a graph with directed links going each way between connected nodes.

Graphs are everywhere; for example, think about road networks or airline routes or computer networks. In all of these cases we might be interested in finding a path through the graph that satisfies some property. It may be that any path will do or we may be interested in a path having the fewest "hops" or a least cost path assuming the hops are not all equivalent, etc.

However, graphs can also be much more abstract. Think of the graph defined as follows: the nodes denote descriptions of a state of the world, e.g. which blocks are on top of what in a blocks scene, and where the links represent actions that change from one state to the other. A path through such a graph (from a start node to a goal node) is a "plan of action" to achieve some desired goal state from some known starting state. It is this type of graph that is of more general interest in AI.

One general approach to problem solving in AI is to reduce the problem to be solved to one of searching a graph. To use this approach, we must specify what are the states, the actions and the goal test.

A state is supposed to be complete, that is, to represent all (and preferably only) the relevant aspects of the problem to be solved. So, for example, when we are planning the cheapest round-the-world flight plan, we don't need to know the address of the airports; knowing the identity of the airport is enough. The address will be important, however, when planning how to get from the hotel to the airport. Note that, in general, to plan an air route we need to know the airport, not just the city, since some cities have multiple airports.

We are assuming that the actions are deterministic, that is, we know exactly the state after the action is performed. We also assume that the actions are discrete, so we don't have to represent what happens while the action is happening. For example, we assume that a flight gets us to the scheduled destination and that what happens
during the flight does not matter (at least when planning the route).

Note that we’ve indicated that (in general) we need a test for the goal, not just one specific goal state. So, for example, we might be interested any city in Germany rather than specifically Frankfurt. Or, when proving a theorem, all we care is about knowing one fact in our current data base of facts. Any final set of facts that contains the desired fact is a proof.

In principle, we could also have multiple starting states, for example, if we have some uncertainty about the starting state but, for now, we are not addressing issues of uncertainty either in the starting state or in the result of the actions.

Note that trees are a subclass of directed graphs (even when not shown with arrows on the links). Trees don't have cycles and every node has a single parent (or is the root). Cycles are bad for searching, since, obviously, you don't want to go round and round getting nowhere.

When asked to search a graph, we can construct an equivalent problem of searching a tree by doing two things: turning undirected links into two directed links; and, more importantly, making sure we never consider a path with a loop or, even better, by never visiting the same node twice.

You can see an example of this converting from a graph to a tree here. If we assume that S is the start of our search and we are trying to find a path to G, then we can walk through the graph and make connections from every node to every connected node that would not create a cycle (and stop whenever we hit G). Note that such a tree has a leaf node for every non-looping path in the graph starting at S.

Also note, however, that even though we avoided loops, some nodes (the colored ones) are duplicated in the tree, that is, they were reached along different non-looping paths. This means that a complete search of this tree might do extra work.

The issue of how much effort to place in avoiding loops and avoiding extra visits to nodes is an important one that we will revisit later when we discuss the various search algorithms.

One important distinction that will help us keep things straight is that between a state and a search node.

A state is an arrangement of the real world (or at least our model of it). We assume that there is an underlying ‘real’ state graph that we are searching (although it might not be explicitly represented in the computer; it may be implicitly defined by the actions). We assume that you can arrive at the same real world state by multiple routes, that is, by different sequences of actions.

A search node, on the other hand, is a data structure in the search algorithm, which constructs an explicit tree of nodes while searching. Each node refers to some state, but not uniquely. Note that a node also corresponds to a path from the start state to the state associated with the node. This follows from the fact that the search algorithm is generating a tree. So, if we return a node, we’re returning a path.
So, let's look at the different classes of search algorithms that we will be exploring. The simplest class is that of the uninformed, any-path algorithms. In particular, we will look at depth-first and breadth-first search. Both of these algorithms basically look at all the nodes in the search tree in a specific order (independent of the goal) and stop when they find the first path to a goal state.

The next class of methods are informed, any-path algorithms. The key idea here is to exploit a task specific measure of goodness to try to either reach the goal more quickly or find a more desirable goal state.

Next, we look at the class of uninformed, optimal algorithms. These methods guarantee finding the "best" path (as measured by the sum of weights on the graph edges) but do not use any information beyond what is in the graph definition.

Finally, we look at informed, optimal algorithms, which also guarantee finding the best path but which exploit heuristic ("rule of thumb") information to find the path faster than the uninformed methods.
The search strategies we will look at are all instances of a common search algorithm, which is shown here. The basic idea is to keep a list (Q) of nodes (that is, partial paths), then to pick one such node from Q, see if it reaches the goal and otherwise extend that path to its neighbors and add them back to Q. Except for details, that’s all there is to it.

Note, by the way, that we are keeping track of the states we have reached (visited) and not entering them in Q more than once. This will certainly avoid us ever looping, no matter how the underlying graph is connected, since we can only ever reach a state once. We will explore the impact of this decision later.

The key questions, of course, is which entry to pick off of Q and how precisely to add the new paths back on the Q. Different choices for these operations produce the various search strategies.

At this point, we are ready to actually look at a specific search. For example, depth-first search always looks at the deepest node in the search tree first. We can get that behavior by:

- picking the first element of Q as the node to test and extend.
- adding the new (extended) paths to the FRONT of Q, so that the next path to be examined will be one of the extensions of the current path to one of the descendants of that node’s state.

One good thing about depth-first search is that Q never gets very big. We will look at this in more detail later, but it’s fairly easy to see that the size of the Q depends on the depth of the search tree and not on its breadth.

Breadth-first is the other major type of uninformed (or blind) search. The basic approach is to once again pick the first element of Q to examine BUT now we place the extended paths at the back of Q. This means that the next path pulled off of Q will typically not be a descendant of the current one, but rather one at the same level in tree.

Note that in breadth-first search, Q gets very big because we postpone looking at longer paths (that go to the next level) until we have finished looking at all the paths at one level.

We’ll look at how to implement other search strategies in just a bit. But, first, lets look at some of the more subtle issues in the implementation.
One subtle point is where in the algorithm one tests for success (that is, the goal test). There are two plausible points: one is when a path is extended and it reaches a goal, the other is when a path is pulled off of Q. We have chosen the latter (testing in step 3 of the algorithm) because it will generalize more readily to optimal searches. However, testing on extension is correct and will save some work for any-path searches.

At this point, we need to agree on more terminology that will play a key role in the rest of our discussion of search. Let's start with the notion of *Visited* as opposed to *Expanded*. We say a state is visited when a path that reaches that state (that is, a node that refers to that state) gets added to Q. So, if the state is anywhere in any node in Q, it has been visited. Note that this is true even if no path to that state has been taken off of Q.

A state M is *Expanded* when a path to that state is pulled off of Q. At that point, the descendants of M are visited and the paths to those descendants added to the Q.

Please try to get this distinction straight, it will save you no end of grief.
In our description of the simple search algorithm, we made use of a Visited list. This is a list of all the states corresponding to any node ever added to Q. As we mentioned earlier, avoiding nodes on the visited list will certainly keep us from looping, even if the graph has loops in it. Note that this mechanism is stronger than just avoiding loops locally in every path; this is a global mechanism across all paths. In fact, it is more general than a loop check on each path, since by definition a loop will involve visiting a state more than once.

But, in addition to avoiding loops, the Visited list will mean that our search will never expand a state more than once. The basic idea is that we do not need to search for a path from any state to the goal more than once. If we did not find a path the first time we tried it, one is not going to materialize the second time. And, it saves work, possibly an enormous amount, not to look again. More on this later.

A word on implementation: Although we speak of a “Visited list”, it is never a good idea to keep track of visited states using a list, since we will continually be checking to see if some particular state is on the list, which will require scanning the list. Instead, we want to use some mechanism that takes roughly constant time. If we have a data structure for the states, we can simply include a “flag” bit indicating whether the state has been visited. In general, one can use a hash table, a data structure that allows us to check if some state has been visited in roughly constant time, independent of the size of the table. Still, no matter how fast we make the access, this table will still require additional space to store. We will see later that this can make the cost of using a Visited list prohibitive for very large problems.

Another key concept to keep straight is that of a heuristic value for a state. The word heuristic generally refers to a “rule of thumb”, something that's helpful but not guaranteed to work.

A heuristic function has similar connotations. It refers to a function (defined on a state - not on a path) that may be helpful in guiding search but which is not guaranteed to produce the desired outcome. Heuristic searches generally make no guarantees on shortest paths or best anything (even when they are called best-first). Nevertheless, using heuristic functions may still provide help by speeding up, at least on average, the process of finding a goal.
Slide 2.2.17

If we can get some estimate of the "distance" to a goal from the current node and we introduce a preference for nodes closer to the goal, then there is a good chance that the search will terminate more quickly. This intuition is clear when thinking about "airline" (as-the-crow-flies) distance to guide a search in Euclidean space, but it generalizes to more abstract situations (as we will see).

Slide 2.2.18

Best-first (also known as "greedy") search is a heuristic (informed) search that uses the value of a heuristic function defined on the states to guide the search. This will not guarantee finding a "best" path, for example, the shortest path to a goal. The heuristic is used in the hope that it will steer us to a quick completion of the search or to a relatively good goal state.

Best-first search can be implemented as follows: pick the "best" path (as measured by heuristic value of the node's state) from all of Q and add the extensions somewhere on Q. So, at any step, we are always examining the pending node with the best heuristic value.

Note that, in the worst case, this search will examine all the same paths that depth or breadth first would examine, but the order of examination may be different and therefore the resulting path will generally be different. Best-first has a kind of breadth-first flavor and we expect that Q will tend to grow more than in depth-first search.

Slide 2.2.19

Note that best-first search requires finding the best node in Q. This is a classic problem in computer science and there are many different approaches that are appropriate in different circumstances. One simple method is simply to scan the Q completely, keeping track of the best element found. Surprisingly, this simple strategy turns out to be the right thing to do in some circumstances. A more sophisticated strategy, such as keeping a data structure called a "priority queue", is more often the correct approach. We will pursue this issue further when we talk about optimal searches.

Slide 2.2.20

This table summarizes the key cost and performance properties of the different any-path search methods. We are assuming that our state space is a tree and so we cannot revisit states and a Visited list is useless.

Note that since this analysis is done for searching a tree with uniform branching factor b and depth d. Therefore, the size of this search space grows exponentially with the depth. So, it should not be surprising that methods that guarantee finding a path will require exponential time in this situation. These estimates are not intended to be tight and precise; instead they are intended to convey a feeling for the tradeoffs.

Note that we could have phrased these results in terms of V, the number of vertices (nodes) in the tree, and then everything would have worst case behavior that is linear in V. We phrase it the way we do because in many applications, the number of nodes depends in an exponential way on some depth parameter, for example, the length of an action plan, and thinking of the cost as linear in the number of nodes is misleading. However, in the algorithms literature, many of these algorithms are described as requiring time linear in the number of nodes.
There are two points of interest in this table. One is the fact that depth-first search requires much less space than the other searches. This is important, since space tends to be the limiting factor in large problems (more on this later). The other is that the time cost of best-first search is higher than that of the others. This is due to the cost of finding the best node in Q, not just the first one. We will also look at this in more detail later.

**Slide 2.2.21**

Remember that we are assuming in this slide that we are searching a tree, so states cannot be visited more than once - so the Visited list is completely superfluous when searching trees. However, if we were to use a Visited list (even implemented as a constant-time access hash table), the only thing that seems to change in this table is that the worst-case space requirements for all the searches go up (and way up for depth-first search). That does not seem to be very useful! Why would we ever use a Visited list?

**Cost and Performance of Any-Path Methods**

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Fewest states?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-First</td>
<td>(b^{d+1})</td>
<td>(b^{d+1})</td>
<td>No</td>
<td>Yes*</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>(b^{d+1})</td>
<td>(b^{d+1})</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Best-First</td>
<td>(b^{d+2})</td>
<td>(b^{d+2})</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

*If there are no infinitely long paths in the graph  
**Best-First needs more time to locate the best node in Q.

Worst case time is proportional to number of nodes added to Q.  
Worst case space is proportional to maximal length of Q (and Visited list).

**Slide 2.2.22**

As we mentioned earlier, the key observation is that with a Visited list, our worst-case time performance is limited by the number of states in the search space (since you visit each state at most once) rather than the number of paths through the nodes in the space, which may be exponentially larger than the number of states, as this classic example shows. Note that none of the paths in the tree have a loop in them, that is, no path visits a state more than once. The Visited list is a way of spending space to limit this time penalty. However, it may not be appropriate for very large search spaces where the space requirements would be prohibitive.

**Slide 2.2.23**

So far, we have been treating time and space in parallel for our algorithms. It is tempting to focus on time as the dominant cost of searching and, for real-time applications, it is. However, for large off-line applications, space may be the limiting factor.

If you do a back of the envelope calculation on the amount of space required to store a tree with branching factor 8 and depth 10, you get a very large number. Many real applications may want to explore bigger spaces.

**Space**

(the final frontier)

- In large search problems, memory is often the limiting factor.
- Imagine searching a tree with branching factor 8 and depth 10. Assume a node requires just 8 bytes of storage. Then, breadth-first search might require up to  
  \((2^8)^{d+1}\times 2^d = 512,000 \text{ bytes} = 6.0 \text{ MB} = 96 \text{ MB}\)
One strategy for enabling such open-ended searches, which may run for a very long time, is Progressive Deepening Search (aka Iterative Deepening Search). The basic idea is to simulate searches with a breadth-like component by a succession of depth-limited depth-first searches. Since depth-first has negligible storage requirements, this is a clean tradeoff of time for space.

Interestingly, PDS is more than just a performance tradeoff. It actually represents a merger of two algorithms that combines the best of both. Let's look at that a little more carefully.

**Slide 2.2.25**

Depth-first search has one strong point - its limited space requirements, which are linear in the depth of the search tree. Aside from that there's not much that can be said for it. In particular, it is susceptible to "going off the deep-end", that is, chasing very deep (possibly infinitely deep) paths. Because of this it does not guarantee, as breadth-first, does to find the shallowest goal states - those requiring the fewest actions to reach.

**Slide 2.2.26**

Breadth-first search on the other hand, does guarantee finding the shallowest goal, but at the expense of space requirements that are exponential in the depth of the search tree.

**Progressive Deepening Search**

*Best of Both Worlds*

- Depth-First Search (DFS) has small space requirements (linear in depth), but has major problems:
  - DFS can run forever in search spaces with infinite length paths
  - DFS does not guarantee of finding shallowest goal
- Breadth-First Search (BFS) guarantees finding shallowest goal, even in the presence of infinite paths, but has one great problem:
  - BFS requires a great deal of space (exponential in depth)

**Progressive Deepening Search**

*Best of Both Worlds*

- Depth-First Search (DFS) has small space requirements (linear in depth), but has major problems:
  - DFS can run forever in search spaces with infinite length paths
  - DFS does not guarantee of finding shallowest goal
- Breadth-First Search (BFS) guarantees finding shallowest goal, even in the presence of infinite paths, but has one great problem:
  - BFS requires a great deal of space (exponential in depth)
- Progressive Deepening Search (PDS) has the advantages of DFS and BFS,
  - PDS has small space requirements (linear in depth)
  - PDS guarantees finding shallowest goal

**Slide 2.2.27**

Progressive-deepening search, on the other hand, has both limited space requirements of DFS and the strong optimality guarantee of BFS. Great! No?

**Progressive Deepening Search**

*Best of Both Worlds*

- Depth-First Search (DFS) has small space requirements (linear in depth), but has major problems:
  - DFS can run forever in search spaces with infinite length paths
  - DFS does not guarantee of finding shallowest goal
- Breadth-First Search (BFS) guarantees finding shallowest goal, even in the presence of infinite paths, but has one great problem:
  - BFS requires a great deal of space (exponential in depth)
- Progressive Deepening Search (PDS) has the advantages of DFS and BFS,
  - PDS has small space requirements (linear in depth)
  - PDS guarantees finding shallowest goal
At first sight, most people find PDS horrifying. Isn’t progressive deepening really wasteful? It looks at the same nodes over and over again...

In small graphs, yes it is wasteful. But, if we really are faced with an exponentially growing space (in the depth), then it turns out that the work at the deepest level dominates the total cost.

It is easy to see this for binary trees, where the number of nodes at level $d$ is about equal to the number of nodes in the rest of the tree. The worst-case time for BFS at level $d$ is proportional to the number of nodes at level $d$, while the worst case time for PDS at that level is proportional to the number of nodes in the whole tree which is almost exactly twice those at the deepest level. So, in the worst case, PDS (for binary trees) does no more than twice as much work as BFS, while using much less space.

This is a worst case analysis, it turns out that if we try to look at the expected case, the situation is even better.

One can derive an estimate of the ratio of the work done by progressive deepening to that done by a single depth-first search: $(b+1)/(b-1)$. This estimate is for the average work (averaging over all possible searches in the tree). As you can see from the table, this ratio approaches one as the branching factor increases (and the resulting exponential explosion gets worse).
Progressive Deepening Search

- Compare the ratio of average time spent on PDS with average time spent on a single DFS with the full depth tree:
  \[(\text{Avg time for PDS})(\text{Avg time for DFS}) = (b+1)/(b-1)\]
- Progressive deepening is an effective strategy for difficult searches.

<table>
<thead>
<tr>
<th>b</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
</tr>
<tr>
<td>25</td>
<td>1.08</td>
</tr>
<tr>
<td>100</td>
<td>1.02</td>
</tr>
</tbody>
</table>

6.034 Notes: Section 2.3

Slide 2.3.1

We will now step through the any-path search methods looking at their implementation in terms of the simple algorithm. We start with depth-first search using a visited list.

The table in the center shows the contents of Q and of the visited list at each time through the loop of the search algorithm. The nodes in Q are indicated by reversed paths, blue is used to indicate newly added nodes (paths). On the right is the graph we are searching and we will label the state of the node that is being extended at each step.

Slide 2.3.2

The first step is to initialize Q with a single node corresponding to the start state (S in this case) and the visited list with the start state.

For many difficult searches, progressive deepening is in fact the only way to go. There are also progressive deepening versions of the optimal searches that we will see later, but that’s beyond our scope.
We pick the first element of Q, which is that initial node, remove it from Q, extend its path to its descendant states (if they have not been Visited) and add the resulting nodes to the front of Q. We also add the states corresponding to these new nodes to the Visited list. So, we get the situation on line 2.

Note that the descendant nodes could have been added to Q in the other order. This would be absolutely valid. We will typically add nodes to Q in such a way that we end up visiting states in alphabetical order, when no other order is specified by the algorithm. This is purely an arbitrary decision.

We then pick the first node on Q, whose state is A, and repeat the process, extending to paths that end at C and D and placing them in front of Q.

We pick the first node, whose state is C, and note that there are no descendants of C and so no new nodes to add.

We pick the first node of Q, whose state is D, and consider extending to states C and G, but C is on the Visited list so we do not add that extension. We do add the path to G to the front of the queue.

We pick the first node of Q, whose state is G, the intended goal state, so we stop and return the path.
Slide 2.3.7
The final path returned goes from S to A, then to D and then to G.

Slide 2.3.8
Tracing out the content of Q can get a little monotonous, although it allows one to trace the performance of the algorithms in detail. Another way to visualize simple searches is to draw out the search tree, as shown here, showing the result of the first expansion in the example we have been looking at.

Slide 2.3.9
In this view, we introduce a left to right bias in deciding which nodes to expand - this is purely arbitrary. It corresponds exactly to the arbitrary decision of which nodes to add to Q first. Giving this bias, we decide to expand the node whose state is A, which ends up visiting C and D.

Slide 2.3.10
We now expand the node corresponding to C, which has no descendants, so we cannot continue to go deeper. At this point, one talks about having to back up or backtrack to the parent node and expanding any unexpanded descendant nodes of the parent. If there were none at that level, we would continue to keep backing up to its parent and so on until an unexpanded node is found. We declare failure if we cannot find any remaining unexpanded nodes. In this case, we find an unexpanded descendant of A, namely D.
Slide 2.3.11
So, we expand D. Note that states C and G are both reachable from D. However, we have already visited C, so we do not add a node corresponding to that path. We add only the new node corresponding to the path to G.

Slide 2.3.12
We now expand G and stop.
This view of depth-first search is the more common one (rather than tracing Q). In fact, it is in this view that one can visualize why it is called depth-first search. The red arrow shows the sequence of expansions during the search and you can see that it is always going as deep in the search tree as possible. Also, we can understand another widely used name for depth-first search, namely backtracking search. However, you should convince yourself that this view is just a different way to visualize the behavior of the Q-based algorithm.

Slide 2.3.13
We can repeat the depth-first process without the Visited list and, as expected, one sees the second path to C added to Q, which was blocked by the use of the Visited list. I'll leave it as an exercise to go through the steps in detail.
Note that in the absence of a Visited list, we still require that we do not form any paths with loops, so if we have visited a state along a particular path, we do not re-visit that state again in any extensions of the path.

Slide 2.3.14
Let's look now at breadth-first search. The difference from depth-first search is that new paths are added to the back of Q. We start as with depth-first with the initial node corresponding to S.
Slide 2.3.15
We pick it and add paths to A and B, as before.

Slide 2.3.16
We pick the first node, whose state is A, and extend the path to C and D and add them to Q (at the back) and here we see the difference from depth-first.

Slide 2.3.17
Now, the first node in Q is the path to B so we pick that and consider its extensions to D and G. Since D is already Visited, we ignore that and add the path to G to the end of the Q.

Slide 2.3.18
At this point, having generated a path to G, we would be justified in stopping. But, as we mentioned earlier, we proceed until the path to the goal becomes the first path in Q.
Slide 2.3.19
We now pull out the node corresponding to C from Q but it does not generate any extensions since C has no descendants.

Slide 2.3.20
So we pull out the path to D. Its potential extensions are to previously visited states and so we get nothing added to Q.

Slide 2.3.21
Finally, we get the path to G and we stop.

Slide 2.3.22
Note that we found a path with fewer states than we did with depth-first search, from S to B to G. In general, breadth-first search guarantees finding a path to the goal with the minimum number of states.
Here we see the behavior of breadth-first search in the search-tree view. In this view, you can see why it is called breadth-first -- it is exploring all the nodes at a single depth level of the search tree before proceeding to the next depth level.

We can repeat the breadth-first process without the Visited list and, as expected, one sees multiple paths to C, D and G are added to Q, which were blocked by the Visited test earlier. I'll leave it as an exercise to go through the steps in detail.

Finally, let's look at Best-First Search. The key difference from depth-first and breadth-first is that we look at the whole Q to find the best node (by heuristic value).

We start as before, but now we’re showing the heuristic value of each path (which is the value of its state) in the Q, so we can easily see which one to extract next.

We pick the first node and extend to A and B.
Slide 2.3.27
We pick the node corresponding to A, since it has the best value (= 2) and extend to C and D.

Slide 2.3.28
The node corresponding to C has the lowest value so we pick that one. That goes nowhere.

Slide 2.3.29
Then, we pick the node corresponding to B which has lower value than the path to D and extend to G (not C because of previous Visit).

Slide 2.3.30
We pick the node corresponding to G and rejoice.
Slide 2.3.31
We found the path to the goal from S to B to G.

6.034 Notes: Section 2.4

Slide 2.4.1
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.

Slide 2.4.2
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.
Slide 2.4.7
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.

Slide 2.4.8
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.

Slide 2.4.9
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.

Slide 2.4.10
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.
Slide 2.4.11
The path length is the 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.

Slide 2.4.12
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).

Slide 2.4.13
Similarly for S-A-D-C.

Slide 2.4.14
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.
Slide 2.4.15
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.

Slide 2.4.16
This generates two new entries on the queue. The new path to C is the shortest path on the Q, so we pick it to expand.

Slide 2.4.17
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.

Slide 2.4.18
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 Q.

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).
Slide 2.4.23
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.

Why not stop on first visiting a goal?
- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.
- This contrasts with the non-optimal searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.

Slide 2.4.24
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.

Why not stop on first visiting a goal?
- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.
- This contrasts with the Any Path searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.
- In the previous example, a path to G was generated at step 5, but it was a different, shorter, path at step 7 that we accepted.

Uniform Cost
Another (easier?) way to see it

Total path cost
UC enumerates paths in order of total path cost!

Slide 2.4.26
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.
Slide 2.4.31
Followed by the path from S to A to D, with length 6.

Slide 2.4.32
And the path from S to B to D, also with length 6.

Slide 2.4.33
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.

Slide 2.4.34
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.
Slide 2.5.1

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.

Slide 2.5.2

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.

Slide 2.5.3

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.
Slide 2.5.4
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.

Slide 2.5.5
We call an estimate that always **underestimates** the remaining distance from any node an **admissible** (heuristic) estimate.

Slide 2.5.6
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 insure 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).

Slide 2.5.7
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).

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 from $S$ to $X$ and 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 revisiting 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 \textbf{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.

\begin{itemize}
  \item The strict Expanded list (also known as a Closed list) is commonly used in implementations of A* but, to guarantee finding optimal paths, this implementation requires a stronger condition for a heuristic than simply being an underestimate.
  \item Here's a counterexample: The heuristic values listed below are all underestimates but A* using an Expanded list will not find the optimal path. The misleading estimate at B throws the algorithm off. C is expanded before the optimal path to it is found.
\end{itemize}

\begin{center}
\begin{tabular}{|c|c|}
\hline
Q & Expanded \\
\hline
1 & (B, S) \\
2 & (101 A S) & S \\
3 & (104 C B S) & (101 A S) & B, S \\
4 & (101 A S) & (104 G B S) & C, B, S \\
5 & (104 G C B S) & A, C, B, S \\
\hline
\end{tabular}
\end{center}

\small
\begin{itemize}
  \item Added paths in blue; underlined paths are chosen for extension.
  \item We show the paths in reversed order; the node's state is the first entry.
\end{itemize}
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.

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.

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.

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.
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.

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.

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. This records both the actual path length at the time of expansion but also the heuristic estimate used (which will come in handy later). If a new node is generated whose actual path length to an expanded state is shorter than the one when the state was expanded, then we remove the node for that state from the Expanded list and we add the new node to Q. This will enable us to reconsider any descendants of that state using the updated path length.

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.
We can take an additional step to improve the behavior of A* in the presence of inconsistent heuristics. This step is optional; it is not required to cope with inconsistency, but it can help with efficiency.

The key observation is that inconsistency arises when a heuristic estimate is too low. We saw that for consistency, the heuristic estimate at a node in the search tree should not decrease by more than the edge cost from the ancestor node. If we have a heuristic where the estimate at a state is lower than that bound, then, if we accept it, we have lost information about the estimated distance to the goal. Recall that a smaller heuristic estimate is always a worse estimate, assuming that they are both admissible. So, what we can do is simply refuse to accept the lower estimate, just stick with the higher estimate. This idea goes by the name of Pathmax.

Note, however, that using Pathmax does not completely fix the problem of inconsistency. This is actually a subtle point and you can find many lectures and papers that miss it. In general, you will still need to detect situations where you find a node that reaches an expanded state with an improved estimate. In those cases, we still need to remove the state from expanded and add the new node to Q.

### Slide 2.6.31
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 node₁, 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 node₂ which matches.

---

**Expanded List**

- Assume we are adding node₁ to Q and node₂ is present in Expanded list with node₂.state = node₁.state
- **Strict**
  - do not add node₁ to Q

---

**Expanded List**

- Assume we are adding node₁ to Q and node₂ is present in Expanded list with node₂.state = node₁.state
- **Strict**
  - do not add node₁ to Q

---

**Expanded List**

- Assume we are adding node₁ to Q and node₂ is present in Expanded list with node₂.state = node₁.state
- **Strict**
  - do not add node₁ to Q

---

**Expanded List**

- Assume we are adding node₁ to Q and node₂ is present in Expanded list with node₂.state = node₁.state
- **Strict**
  - do not add node₁ to Q

---

With a strict Expanded list, we simply discard node₁; we do not add it to Q.
Slide 2.6.33
With a non-strict list, the situation is a bit more complicated. We want to make sure that node1 has not found a better path to the state than node2. 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 2.6.34
When we are using Pathmax, it may be the case that even though the new node has a worse path than the old node it may have a better estimate, since with Pathmax the estimates will depend on the path taken to a state. In that case, we want to still remove the old node from Expanded but we want to add a new node to Q that has the path from the old node (node2) together with the estimate from the new node (node1). This new, hybrid node represents the best information we have about the optimal path to that state as well as the best estimate of remaining distance to the goal.

Slide 2.6.35
If the new node is not in Expanded but its state has been Visited, then there is already a node with that state in Q, we can follow essentially the same logic to decide whether to add it to Q.

Slide 2.6.36
Here we see our simple example using Pathmax and a non-strict Expanded list in the presence of an inconsistent heuristic. In step 2 we see that, when we generate a path to B, we need to modify the value of h(B) drastically since the heuristic estimate for B is way off. The estimate at S is 90 and the edge length to B is 2, so the lowest consistent value for h(B) is 88. So, f(B) is h(B) + c(S,B) which is 90 (not 3).

A similar, though less drastic, situation occurs at step 5. When going from A to C, the minimum consistent value for h(C) is 99, not 90 as given. So, we use an f(C) value of 101, since the path length to C is 2.

Note that in step 6 we add a new node corresponding to G and since it has a better path length than the existing node for G, we discard the old node and keep the new one.
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.

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.

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.

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 2.6.41
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Heuristic</th>
<th>Expanded List</th>
<th>Optimality Guaranteed?</th>
<th>Worst Case # Expansions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Cost</td>
<td>None</td>
<td>Strict</td>
<td>Yes</td>
<td>$N$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Admissible</td>
<td>None</td>
<td>Yes</td>
<td>$&gt;N$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Consistent</td>
<td>Strict</td>
<td>Yes</td>
<td>$N$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Admissible</td>
<td>Strict</td>
<td>No</td>
<td>$N$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Admissible</td>
<td>Non Strict</td>
<td>Yes</td>
<td>$&gt;N$</td>
</tr>
</tbody>
</table>

$N$ is number of states in graph

Slide 2.6.42
Here we summarize the optimality and complexity of the various algorithms we have been examining.

Worst Case Complexity
- A state space with $N$ states may give rise to a search tree that has a number of nodes that is exponential in $N$, as in this example.
- Searches without a visited (expanded) list may, in the worst case, visit (expand) every node in the search tree.
- Searches with strict visited (expanded lists) will visit (expand) each state only once.

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.

Optional Topics
- These slides go into more depth on a variety of topics we have touched upon:
  - Optimality of $A^*$
  - Impact of a better heuristic on $A^*$
  - Why does consistency guarantee optimal paths for $A^*$ with strict expanded list?
  - Pathmax
  - Algorithmic issues for $A^*$
- These are not required and are provided for those interested in pursuing these topics.
Let's go through a quick proof that A* actually finds the optimal path. Start by assuming that A* has selected a node G.

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.

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, $g(G')$.

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.
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 $h^*$.

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 $h^*$.

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.

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.
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 to 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.

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!

**New topic:**

If \( N_j \) is a descendant of \( N_i \) in the search tree, Pathmax tells us that we should use the larger of \( h(N_j) \) and \( h(N_i) \) for the total estimated path length at \( N_j \). That is, we should not let the value of \( f \) drop along a path. This basically follows from enforcing consistency on the heuristic estimates, as we will now see.

Technically, what Pathmax is doing is using the constraint on the heuristic values imposed by consistency to pick a new value for a state's heuristic estimate when the one provided is obviously too low -- that is, when it drops more than the edge cost when moving from one state to an adjacent state. From this, the conclusion that the \( f \) values should not drop along the path follows.

To see this, recall that the value of \( f \) at \( N_j \) is simply the path length to the parent node, \( g(N_j) \), plus the edge cost from the parent to the descendant, \( C(s_i, s_j) \), plus the heuristic estimate at the descendant. Substituting the Pathmax expression for the heuristic estimate and propagating the max (which is ok since all the quantities are positive), we arrive at the conclusion that we want to use the larger of the \( f \) values.
Pathmax does not completely correct the problem of having an inconsistent heuristic; we will still need to occasionally re-visit a previously expanded node. The reason is that pathmax effectively makes the heuristic estimate at a state depend on the path we use to reach it. You can look at our proof of why consistency implies that the first expanded path to a state is optimal and see that it assumes crucially that heuristic estimates depend only on the state and not the path to the state.

Slide 2.7.19
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.

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 2.7.21
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.

---

Uniform Cost + Strict Expanded List
(order of time growth in worst case)
Our simple algorithm can be summarized as follows:
1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q
Assume strict Expanded “list” is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.
Assume we have a graph with N nodes and L links. Graphs where nodes have O(N) links are dense. Graphs where the nodes have a nearly constant number of links are sparse. For dense graphs, L is O(N).
Nodes taken from Q ? O(N)
Cost of picking a node from Q using linear scan? O(N)

---

Slide 2.7.22
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.

---

Uniform Cost + Strict Expanded List
(order of time growth in worst case)
Our simple algorithm can be summarized as follows:
1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q
Assume strict Expanded “list” is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.
Assume we have a graph with N nodes and L links. Graphs where nodes have O(N) links are dense. Graphs where the nodes have a nearly constant number of links are sparse. For dense graphs, L is O(N).
Nodes taken from Q ? O(N)
Cost of picking a node from Q using linear scan? O(N)
Attempts to add nodes to Q (many are rejected)? O(L)
Cost of adding a node to Q ? O(1)

---

Slide 2.7.23
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 2.7.24
Adding to the Q, assuming it is a hash table, as we have been assuming here, can be done in constant time.
Slide 2.7.25
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 2.7.26
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 2.7.27
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 2.7.28
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.

A* (order of time growth in worst case)
Our simple algorithm can be summarized as follows:
1. Take the best search node from $Q$
2. Are we there yet?
3. Add path extensions to $Q$
Assume strict Expanded “list” is implemented as a hash table, which gives constant time access. $Q$ also implemented as a hash table.
Assume we have a graph with $N$ nodes and $L$ links. Graphs where the nodes have $O(N)$ links are dense. Graphs where the nodes have a nearly constant number of links are sparse. For dense graphs, $L$ is $O(N^2)$.

<table>
<thead>
<tr>
<th>Nodes taken from $Q$?</th>
<th>$O(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost of picking a node from $Q$ using linear scan?</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Attempts to add nodes to $Q$ (many are rejected)?</td>
<td>$O(L)$</td>
</tr>
<tr>
<td>Cost of adding a node to $Q$?</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Total cost: $O(N^2 + L)$
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.