

## 6.034 Notes: Section 5.1

### Slide 5.1.1

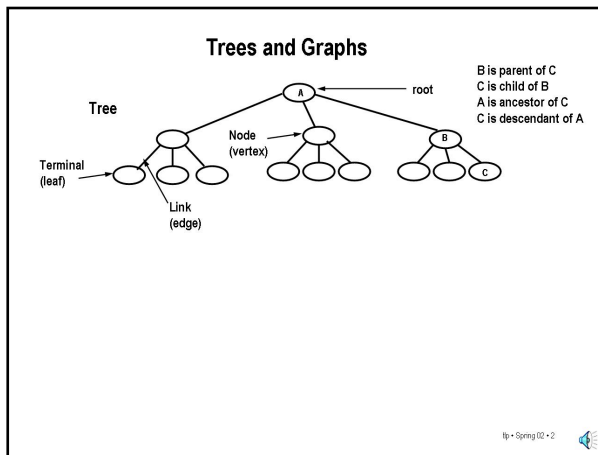
Search plays a key role in many parts of AI. These algorithms provide the conceptual backbone of almost every approach to the systematic exploration of alternatives.

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.

### 6.034 Artificial Intelligence

- Big idea: Search allows exploring alternatives
- Background
- Uninformed vs Informed
- Any Path vs Optimal Path
- Implementation and Performance

lp • Spring 02 • 1



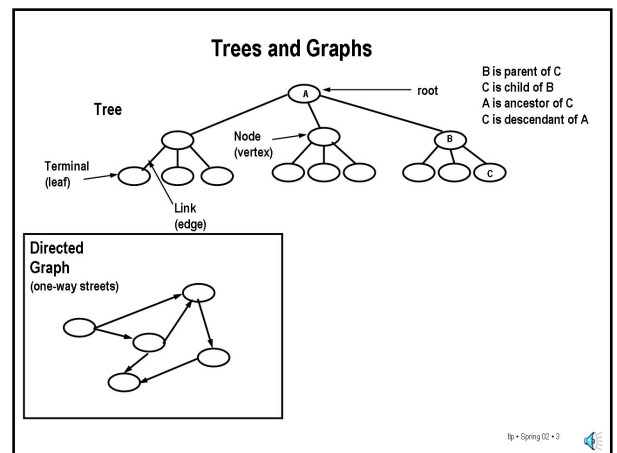
### Slide 5.1.2

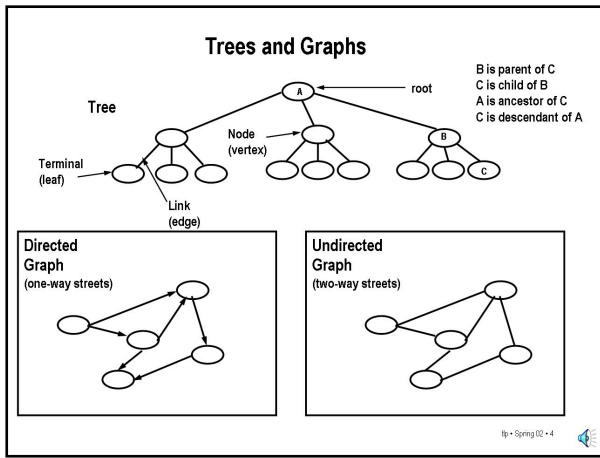
The search methods we will be dealing with are defined on trees and graphs, so we need to fix on some terminology for these structures:

- 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.

### Slide 5.1.3

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).



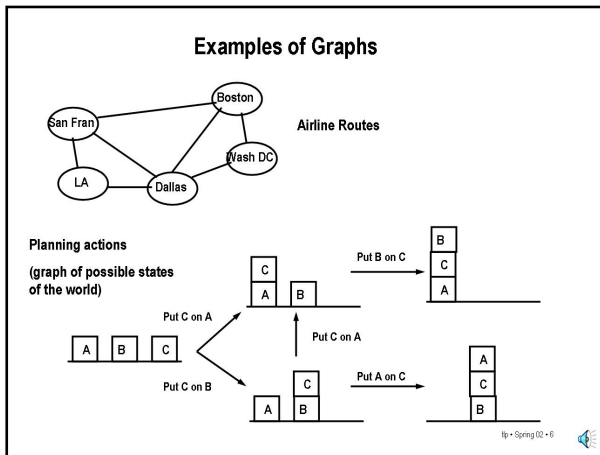
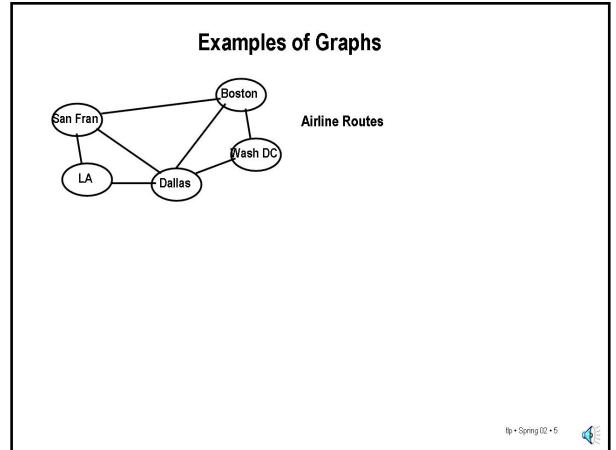


Slide 5.1.4

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.

Slide 5.1.5

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.



Slide 5.1.6

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.

Slide 5.1.7

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 in 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

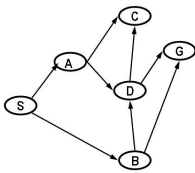
- ### Problem Solving Paradigm
- What are the **states**? (All relevant aspects of the problem)
    - Arrangement of parts (to plan an assembly)
    - Positions of trucks (to plan package distribution)
    - City (to plan a trip)
    - Set of facts (e.g. to prove geometry theorem)
  - What are the **actions** (operators)? (Deterministic and discrete)
    - Assemble two parts
    - Move a truck to a new position
    - Fly to a new city
    - Apply a theorem to derive new fact
  - What is the **goal test**? (Conditions for success)
    - All parts in place
    - All packages delivered
    - Reached destination city
    - Derived goal fact
- lp • Spring 02 • 7

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.

### Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having  $\leq 1$  parent
- We can turn graph search problems into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping track of visited nodes globally)



fp • Spring 02 • 8

### Slide 5.1.8

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.

### Slide 5.1.9

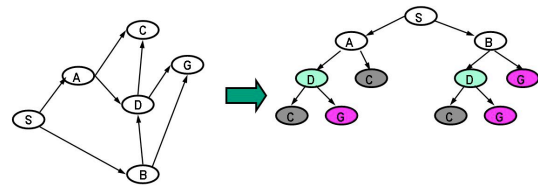
You can see an example of this converting from a graph to a tree here. If we assume that S is the start of our search and we are trying to find a path to G, then we can walk through the graph and make connections from every node to every connected node that would not create a cycle (and stop whenever 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.

### Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having  $\leq 1$  parent
- We can turn graph search problems (from S to G) into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping track of visited nodes globally)



fp • Spring 02 • 9

### Terminology

- **State** – Used to refer to the vertices of the underlying **graph** that is being searched, that is, states in the problem domain, for example, a city, an arrangement of blocks or the arrangement of parts in a puzzle.
- **Search Node** – Refers to the vertices of the search **tree** which is being generated by the search algorithm. Each node refers to a state of the world; **many nodes may refer to the same state**. Importantly, a node implicitly represents a path (from the start state of the search to the state associated with the node). Because search nodes are part of a search tree, they have a unique ancestor node (except for the root node).

fp • Spring 02 • 10

### Slide 5.1.10

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.

## 6.034 Notes: Section 5.2


## Slide 5.2.1

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.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.

lp • Spring 02 • 1 

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.

lp • Spring 02 • 2 


## Slide 5.2.2

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.


## Slide 5.2.3

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.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.

lp • Spring 02 • 3 

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	A*	Uses path "length" measure and heuristic. Finds "shortest" path.

lp • Spring 02 • 4 

## Slide 5.2.4

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.

**Slide 5.2.5**

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 keep us from 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.

**Simple Search Algorithm**

A *search node* is a path from some state  $X$  to the start state, e.g.,  $(X B A S)$

The *state* of a search node is the most recent state of the path, e.g.  $X$ .

Let  $Q$  be a list of search nodes, e.g.  $\{(X B A S) (C B A S) \dots\}$ .

Let  $S$  be the start state.

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

lp • Spring 02 • 5

**Simple Search Algorithm**

A *search node* is a path from some state  $X$  to the start state, e.g.,  $(X B A S)$

The *state* of a search node is the most recent state of the path, e.g.  $X$ .

Let  $Q$  be a list of search nodes, e.g.  $\{(X B A S) (C B A S) \dots\}$ .

Let  $S$  be the start state.

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

Critical decisions:

Step 2: picking  $N$  from  $Q$

Step 6: adding extensions of  $N$  to  $Q$

lp • Spring 02 • 6

**Slide 5.2.6**

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

**Slide 5.2.7**

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.

**Implementing the Search Strategies**

Depth-first:

Pick first element of  $Q$

Add path extensions to front of  $Q$

lp • Spring 02 • 7

**Implementing the Search Strategies**

Depth-first:

Pick first element of  $Q$

Add path extensions to front of  $Q$

Breadth-first:

Pick first element of  $Q$

Add path extensions to end of  $Q$

lp • Spring 02 • 8

**Slide 5.2.8**

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, let's look at some of the more subtle issues in the implementation.

## Slide 5.2.9

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.

## Testing for the Goal

- This algorithm stops (in step 3) when  $\text{state}(N) = G$  or, in general, when  $\text{state}(N)$  satisfies the goal test.
- We could have performed this test in step 6 as each extended path is added to Q. This would catch termination earlier and be perfectly correct for the searches we have covered so far.
- However, performing the test in step 6 will be incorrect for the optimal searches. We have chosen to leave the test in step 3 to maintain uniformity with these future searches.

lp • Spring 02 • 9



## Terminology

- **Visited** – a state M is first visited when a path to M first gets added to Q. In general, a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly “visited” them to place them on Q, but we have not yet examined them carefully.

lp • Spring 02 • 10



## Slide 5.2.10

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.

## Slide 5.2.11

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.

## Terminology

- **Visited** – a state M is first visited when a path to M first gets added to Q. In general, a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly “visited” them to place them on Q, but we have not yet generated its descendants.
- **Expanded** – a state M is expanded when it is the state of a search node that is pulled off of Q. At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M (instead of M itself) as being expanded. However, once a node is expanded we are done with it; we will not need to expand it again. In fact, we discard it from Q.

lp • Spring 02 • 11



## Terminology

- **Visited** – a state M is first visited when a path to M first gets added to Q. In general, a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly “visited” them to place them on Q, but we have not yet examined them carefully.
- **Expanded** – a state M is expanded when it is the state of a search node that is pulled off of Q. At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M (instead of M itself) as being expanded. However, once a node is expanded we are done with it; we will not need to expand it again. In fact, we discard it from Q.
- This distinction plays a **key** role in our discussion of the various search algorithms; study it carefully.

lp • Spring 02 • 12



## Slide 5.2.12

Please try to get this distinction straight; it will save you no end of grief.

## Slide 5.2.13

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.

## Visited States

- Keeping track of visited states generally improves time efficiency when searching graphs, without affecting correctness. Note, however, that substantial additional space may be required to keep track of visited states.
- If all we want to do is find a path from the start to the goal, there is no advantage to adding a search node whose state is already the state of another search node.
- Any state reachable from the node the second time would have been reachable from that node the first time.
- Note that, when using Visited, each state will only ever have at most one path to it (search node) in Q.
- We'll have to revisit this issue when we look at optimal searching.

Spring 02 • 13



## Implementation Issues: The Visited list

- Although we speak of a Visited list, this is never the preferred implementation.
- If the graph states are known ahead of time as an explicit set, then space is allocated in the state itself to keep a mark; which makes both adding to Visited and checking if a state is Visited a constant time operation.
- Alternatively, as is more common in AI, if the states are generated on the fly, then a hash table may be used for efficient detection of previously visited states.
- Note that, in any case, the incremental space cost of a Visited list will be proportional to the number of states – which can be very high in some problems.

Spring 02 • 14



## Slide 5.2.14

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.

## Slide 5.2.15

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.

## Terminology

- **Heuristic** – The word generally refers to a "rule of thumb," something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal."

Spring 02 • 15



## Terminology

- **Heuristic** – The word generally refers to a "rule of thumb," something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal."
- **Heuristic function** – In search terms, a function that computes a value for a state (but does not depend on any path to that state) that may be helpful in guiding the search. There are two related forms of heuristic guidance that one sees:

Spring 02 • 16



## Slide 5.2.16

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 5.2.17

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).

## Terminology

- **Heuristic** – The word generally refers to a "rule of thumb," something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal."
- **Heuristic function** – In search terms, a function that computes a value for a state (but does not depend on any path to that state) that may be helpful in guiding the search.
- **Estimated distance to goal** – this type of heuristic function depends on the state and the goal. The classic example is straight-line distance used as an estimate for actual distance in a road network. This type of information can help increase the efficiency of a search.

Spring 02 • 17



## Implementing the Search Strategies

## Depth-first:

- Pick first element of Q
- Add path extensions to front of Q

## Breadth-first:

- Pick first element of Q
- Add path extensions to end of Q

## Best-first:

- Pick "best" (measured by heuristic value of state) element of Q
- Add path extensions anywhere in Q (it may be more efficient to keep the Q ordered in some way so as to make it easier to find the "best" element).

Spring 02 • 18



## Slide 5.2.18

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 5.2.19

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.

## Implementation Issues: Finding the best node

- There are many possible approaches to finding the best node in Q.
  - Scanning Q to find lowest value
  - Sorting Q and picking the first element
  - Keeping the Q sorted by doing "sorted" insertions
  - Keeping Q as a priority queue
- Which of these is best will depend among other things on how many children nodes have on average. We will look at this in more detail later.

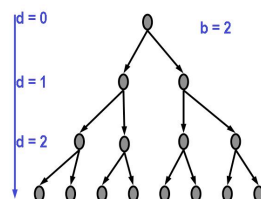
Spring 02 • 19



## Worst Case Running Time

Max Time  $\propto$  Max #Visited

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.



$d$  is depth  
 $b$  is branching factor  
 $b^d < (b^{d+1} - 1) / (b - 1) < b^{d+1}$   
 states in tree

Spring 02 • 20



## Slide 5.2.20

Let's think a bit about the worst case running time of the searches that we have been discussing. The actual running time, of course, will depend on details of the computer and of the software implementation. But, we can roughly compare the various algorithms by thinking of the number of nodes added to Q. The running time should be roughly proportional to this number.

In AI we usually think of a "typical" search space as being a tree with uniform branching factor  $b$  and depth  $d$ . The depth parameter may represent the number of steps in a plan of action or the number of moves in a game. The branching factor reflects the number of different choices that we have at each step. It is easy to see that the number of states in such a tree grows exponentially with the depth.



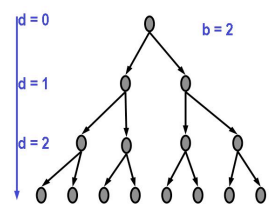
Slide 5.2.21

In a tree-structured search space, the nodes added to the search Q will simply correspond to the visited states. In the worst case, when the states 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 to this worst case.

### Worst Case Running Time

Max Time  $\propto$  Max #Visited

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited list, may have to visit each state at least once, in the worst case.
- So, all searches will have worst case running times that are at least proportional to the total number of states and therefore exponential in the "depth" parameter.

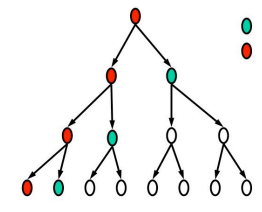


$b^d < (b^{d+1} - 1) / (b - 1) < b^{d+1}$   
states in tree

© Spring 02 • 21

### Worst Case Space

Max Q size = Max (#Visited - #Expanded)



visited (green)  
expanded (red)

Depth First max Q size  
 $(b - 1)d \approx bd$

© Spring 02 • 22

Slide 5.2.22

In addition to thinking about running time, we should also think about the memory space required for searches. The dominant factor in the space requirements for these searches is the maximum size of the search Q. The size of the search Q in a tree-structured search space is simply the number of visited states minus the number of expanded states.

For a depth-first search, we can see that Q holds the unexpanded "siblings" of the nodes along the path that we are currently considering. In a tree, the path length cannot be greater than d and the number of unexpanded siblings cannot be greater than b-1, so this tells us that the length of Q is always less than b\*d, that is, the space requirements are linear in d.

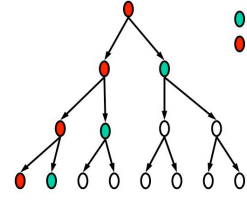
Slide 5.2.23

The situation for breadth-first search is much different than that for depth-first search. Here the worst case happens after we've visited all the nodes at depth d-1. At that point, all the nodes at depth d have been visited and none expanded. So, the Q has size  $b^d$ , that is, a size exponential in d.

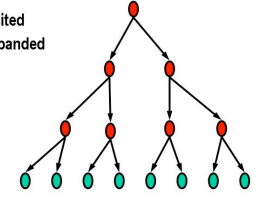
Note that, in the worst case, best-first behaves as breadth-first and has the same space requirements.

### Worst Case Space

Max Q size = Max (#Visited - #Expanded)



Depth First max Q size  
 $(b - 1)d \approx bd$



Breadth First max Q size  
 $b^d$

© Spring 02 • 23

### Cost and Performance of Any-Path Methods

Searching a tree with branching factor b and depth d  
(without using a Visited list)

Search Method	Worst Time	Worst Space	Fewest states?	Guaranteed to find path?
Depth-First	$b^{d+1}$	$bd$	No	Yes*
Breadth-first	$b^{d+1}$	$b^d$	Yes	Yes
Best-First	$b^{d+1}$ **	$b^d$	No	Yes*

\*If there are no infinitely long paths in the search space  
\*\* 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

© Spring 02 • 24

Slide 5.2.24

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.

Recall that 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 5.2.25

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

Searching a tree with branching factor  $b$  and depth  $d$   
(using a Visited list)

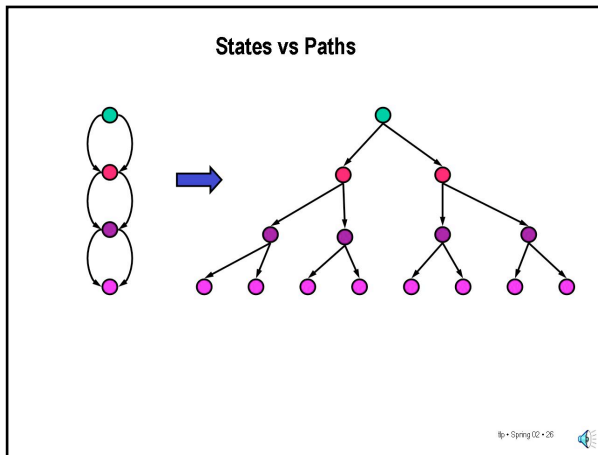
Search Method	Worst Time	Worst Space	Fewest states?	Guaranteed to find path?
Depth-First	$b^{d+1}$	<del><math>b^d</math></del> $b^{d+1}$	No	Yes*
Breadth-first	$b^{d+1}$	<del><math>b^d</math></del> $b^{d+1}$	Yes	Yes
Best-First	$b^{d+1}$ **	<del><math>b^d</math></del> $b^{d+1}$	No	Yes*

\*If there are no infinitely long paths in the search space

\*\* 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)

Spring 02 • 25



### Slide 5.2.26

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 5.2.27

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

$$(2^3)^{10} \times 2^3 = 2^{33} \text{ bytes} = 8,000 \text{ Mbytes} = 8\text{Gbytes}$$

Spring 02 • 27



### 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
 
$$(2^8)^{10} \times 2^3 = 2^{23} \text{ bytes} = 8,000 \text{ Mbytes} = 8\text{Gbytes}$$
- One strategy is to trade time for memory. For example, we can emulate breadth-first search by repeated applications of depth-first search, each up to a preset depth limit. This is called **progressive deepening search (PDS)**:
  1.  $C=1$
  2. Do DFS to max depth  $C$ . If path found, return it.
  3. Otherwise, increment  $C$  and go to 2.

Spring 02 • 28

#### Slide 5.2.28

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 5.2.29

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.

### 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 finding shallowest goal

Spring 02 • 29

### 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 is has one great problem:
  - BFS requires a great deal of space (exponential in depth)

Spring 02 • 30

#### Slide 5.2.30

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.

#### Slide 5.2.31

Progressive-deepening search, on the other 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 is 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

Spring 02 • 31

### Progressive Deepening Search

- Isn't Progressive Deepening (PDS) too expensive?

Spring 02 • 32

**Slide 5.2.32**

At first sight, most people find PDS horrifying. Isn't progressive deepening really wasteful? It looks at the same nodes over and over again...

**Slide 5.2.33**

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.

### Progressive Deepening Search

- Isn't Progressive Deepening (PDS) too expensive?
- In exponential trees, time is dominated by deepest search.

Spring 02 • 33

### Progressive Deepening Search

- Isn't Progressive Deepening (PDS) too expensive?
- In exponential trees, time is dominated by deepest search.
- For example, if branching factor is 2, then the number of nodes at depth  $d$  is  $2^d$  while the total number of nodes in all previous levels is  $2^d - 1$ , so the difference between looking at whole tree versus only the deepest level is at worst a factor of 2 in performance.

Spring 02 • 34

**Slide 5.2.34**

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.

**Slide 5.2.35**

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:  
 $(\text{Avg time for PDS})/(\text{Avg time for DFS}) \approx (b+1)/(b-1)$

b	ratio
2	3
3	2
5	1.5
25	1.08
100	1.02

Spring 02 • 35

### 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:  
 $(\text{Avg time for PDS})/(\text{Avg time for DFS}) \approx (b+1)/(b-1)$
- Progressive deepening is an effective strategy for difficult searches.

b	ratio
2	3
3	2
5	1.5
25	1.08
100	1.02

lp • Spring 02 • 36

### Slide 5.2.36

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.

## 6.034 Notes: Section 5.3

### Slide 5.3.1

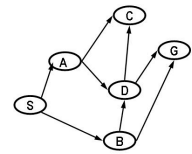
We will now step through the any-path search methods looking at their implementation in terms of the simple algorithm. We start with depth-first search using a Visited list.

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.

### Depth-First

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1		
2		
3		
4		
5		



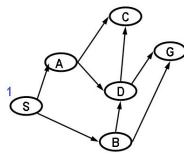
Added paths in blue  
 We show the paths in reversed order; the node's state is the first entry.

lp • Spring 02 • 1

### Depth-First

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2		
3		
4		
5		



Added paths in blue  
 We show the paths in reversed order; the node's state is the first entry.

lp • Spring 02 • 2

### Slide 5.3.2

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.

**Slide 5.3.3**

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 at the front of Q.

**Depth-First**

Pick first element of Q. Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3		
4		
5		

Added paths in **blue**  
We show the paths in **reversed** order; the node's state is the first entry.

lp • Spring 02 • 3

**Slide 5.3.4**

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.

**Depth-First**

Pick first element of Q. Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4		
5		

Added paths in **blue**  
We show the paths in **reversed** order; the node's state is the first entry.

lp • Spring 02 • 4

**Slide 5.3.5**

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 Q.

**Depth-First**

Pick first element of Q. Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5		

Added paths in **blue**  
We show the paths in **reversed** order; the node's state is the first entry.

lp • Spring 02 • 5

**Slide 5.3.6**

We pick the first node of Q, whose state is G, the intended goal state, so we stop and return the path.

**Depth-First**

Pick first element of Q. Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S

Added paths in **blue**  
We show the paths in **reversed** order; the node's state is the first entry.

lp • Spring 02 • 6

Slide 5.3.7

The final path returned goes from S to A, then to D and then to G.

### Depth-First

Pick first element of Q. Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S

Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

lp • Spring 02 • 7

### Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)  
Dark blue fill = Visited & Expanded  
Light gray fill = Visited

lp • Spring 02 • 8

Slide 5.3.8

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 5.3.9

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.

### Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)  
Dark blue fill = Visited & Expanded  
Light gray fill = Visited

lp • Spring 02 • 9

### Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)  
Dark blue fill = Visited & Expanded  
Light gray fill = Visited

lp • Spring 02 • 10

Slide 5.3.10

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 5.3.11

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.

### Depth-First

Another (easier?) way to see it

NB: C is not visited again

Numbers indicate order pulled off of Q (expanded)  
 Dark blue fill = Visited & Expanded  
 Light gray fill = Visited

Spring 02 • 11

Slide 5.3.12

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.

### Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)  
 Dark blue fill = Visited & Expanded  
 Light gray fill = Visited

Spring 02 • 12

Slide 5.3.13

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.

### Depth-First (without Visited list)

Pick first element of Q; Add path extensions to front of Q

	Q
1	(S)
2	(A S) (B S)
3	(C A S) (D A S) (B S)
4	(D A S) (B S)
5	(C D A S) (G D A S) (B S)
6	(G D A S) (B S)

Added paths in blue  
 We show the paths in **reversed** order; the node's state is the first entry.  
 Do not extend a path to a state if the resulting path would have a loop.

Spring 02 • 13

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2	
3	
4	
5	
6	

Added paths in blue  
 We show the paths in **reversed** order; the node's state is the first entry.

Spring 02 • 14

Slide 5.3.14

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.



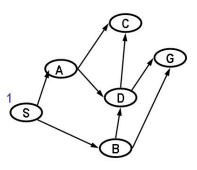
Slide 5.3.15

We pick it and add paths to A and B, as before.

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3	
4	
5	
6	



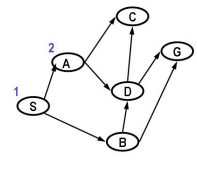
Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 15

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4	
5	
6	



Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 16

Slide 5.3.16

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.

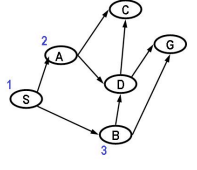
Slide 5.3.17

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 Q.

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4 (C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	
6	



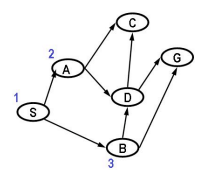
Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 17

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4 (C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	
6	



Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.  
\* We could have stopped here, when the first path to the goal was generated.

Spring 02 • 18

Slide 5.3.18

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 5.3.19

We now pull out the node corresponding to C from Q but it does not generate any extensions since C has no descendants.

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4 (C A S) (D A S) (G B S)*	G,C,D,B,A,S
5 (D A S) (G B S)	G,C,D,B,A,S
6	

Added paths in blue

We show the paths in reversed order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Spring 02 • 19

Slide 5.3.20

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

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4 (C A S) (D A S) (G B S)*	G,C,D,B,A,S
5 (D A S) (G B S)	G,C,D,B,A,S
6	

Added paths in blue

We show the paths in reversed order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Spring 02 • 20

Slide 5.3.21

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

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4 (C A S) (D A S) (G B S)*	G,C,D,B,A,S
5 (D A S) (G B S)	G,C,D,B,A,S
6 (G B S)	G,C,D,B,A,S

Added paths in blue

We show the paths in reversed order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Spring 02 • 21

Slide 5.3.22

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.

### Breadth-First

Pick first element of Q; Add path extensions to end of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A,B,S
3 (B S) (C A S) (D A S)	C,D,B,A,S
4 (C A S) (D A S) (G B S)*	G,C,D,B,A,S
5 (D A S) (G B S)	G,C,D,B,A,S
6 (G B S)	G,C,D,B,A,S

Added paths in blue

We show the paths in reversed order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Spring 02 • 22

Slide 5.3.23

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.

### Breadth-First

Another (easier?) way to see it

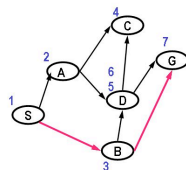
Numbers indicate order pulled off of Q (expanded)  
 Dark blue fill = Visited & Expanded  
 Light gray fill = Visited

Spring 02 • 23

### Breadth-First (without Visited list)

Pick first element of Q; Add path extensions to end of Q

	Q
1	(S)
2	(A S) (B S)
3	(B S) (C A S) (D A S)
4	(C A S) (D A S) (D B S) (G B S)*
5	(D A S) (D B S) (G B S)
6	(D B S) (G B S) (C D A S) (G D A S)
7	(G B S) (C D A S) (G D A S) (C D B S) (G D B S)



Added paths in blue

We show the paths in **reversed** order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Slide 5.3.24

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.

Slide 5.3.25

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.

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2	
3	
4	
5	

Heuristic Values  
 A=2 C=1 S=10  
 B=3 D=4 G=0

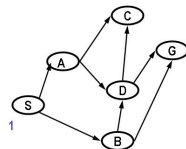
Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in **reversed** order; the node's state is the first entry.

Spring 02 • 25

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2 (2 A S) (3 B S)	A,B,S
3	
4	
5	



Heuristic Values  
 A=2 C=1 S=10  
 B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in **reversed** order; the node's state is the first entry.

Slide 5.3.26

We pick the first node and extend to A and B.

Slide 5.3.27

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

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2 (2 A S) (3 B S)	A,B,S
3 (1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	
5	

Heuristic Values  
A=2 C=1 S=10  
B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 27

Slide 5.3.28

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

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2 (2 A S) (3 B S)	A,B,S
3 (1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4 (3 B S) (4 D A S)	C,D,B,A,S
5	

Heuristic Values  
A=2 C=1 S=10  
B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 28

Slide 5.3.29

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).

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2 (2 A S) (3 B S)	A,B,S
3 (1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4 (3 B S) (4 D A S)	C,D,B,A,S
5 (0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values  
A=2 C=1 S=10  
B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 29

Slide 5.3.30

We pick the node corresponding to G and rejoice.

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2 (2 A S) (3 B S)	A,B,S
3 (1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4 (3 B S) (4 D A S)	C,D,B,A,S
5 (0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values  
A=2 C=1 S=10  
B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 30

Slide 5.3.31

We found the path to the goal from S to B to G.

### Best-First

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

Q	Visited
1 (10 S)	S
2 (2 A S) (3 B S)	A,B,S
3 (1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4 (3 B S) (4 D A S)	C,D,B,A,S
5 (0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values

A=2    C=1    S=10  
B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
We show the paths in reversed order; the node's state is the first entry.

lp • Spring 02 • 31

## 6.034 Notes: Section 5.4

Slide 5.4.1

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.

### Classes of Search

Class	Name	Operation
Any Path Uninformed	Depth-First	Systematic exploration of whole tree until a goal node is found.
	Breadth-First	
Any Path Informed	Best-First	Uses heuristic measure of goodness of a node, e.g. estimated distance to goal.

lp • Spring 02 • 1

### Classes of Search

Class	Name	Operation
Any Path Uninformed	Depth-First	Systematic exploration of whole tree until a goal node is found.
	Breadth-First	
Any Path Informed	Best-First	Uses heuristic measure of goodness of a node, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.

lp • Spring 02 • 2

Slide 5.4.2

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 5.4.3

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.

### Simple Search Algorithm

A **search node** is a path from some state X to the start state, e.g., (X B A S)  
 The **state** of a search node is the most recent state of the path, e.g. X.  
 Let Q be a list of search nodes, e.g. ((X B A S) (C B A S) ...).  
 Let S be the start state.

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

**Critical decisions:**  
 Step 2: picking N from Q  
 Step 6: adding extensions of N to Q

lp • Spring 02 • 3

### Simple Search Algorithm

A **search node** is a path from some state X to the start state, e.g., (X B A S)  
 The **state** of a search node is the most recent state of the path, e.g. X.  
 Let Q be a list of search nodes, e.g. ((X B A S) (C B A S) ...).  
 Let S be the start state.

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

**Critical decisions:**  
 Step 2: picking N from Q  
 Step 6: adding extensions of N to Q

lp • Spring 02 • 4

## Slide 5.4.4

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 5.4.5

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).

### Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.

lp • Spring 02 • 5

### Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with optimal searches can cause us to miss the best path.

lp • Spring 02 • 6

## Slide 5.4.6

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

Slide 5.4.7

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.

### Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with UC can cause us to miss the best path.

- The shortest path from S to G is (S A D G)

lp • Spring 02 • 7

### Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with UC can cause us to miss the best path.

- The shortest path from S to G is (S A D G)
- But, on extending (S), A and D would be added to Visited list and so (S A) would not be extended to (S A D)

lp • Spring 02 • 8

Slide 5.4.8

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 5.4.9

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.

### Implementing Optimal Search Strategies

**Uniform Cost:**

- Pick best (measured by path length) element of Q
- Add path extensions anywhere in Q.

lp • Spring 02 • 9

### Uniform Cost

- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path

lp • Spring 02 • 10

Slide 5.4.10

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 5.4.11

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.

### Uniform Cost

- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path

Total path cost:  
(S A C) 4

Spring 02 • 11

### Uniform Cost

- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path

Total path cost:  
(S A C) 4  
(S B D G) 8

Spring 02 • 12

Slide 5.4.12

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 5.4.13

Similarly for S-A-D-C.

### Uniform Cost

- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path

Total path cost:  
(S A C) 4  
(S B D G) 8  
(S A D C) 9

Spring 02 • 13

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	State
1	<u>(0 S)</u>

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 14

Slide 5.4.14

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 5.4.15

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.

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	<u>(2 A S)</u> (5 B S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 15

Slide 5.4.16

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

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	<u>(2 A S)</u> (5 B S)
3	<u>(4 C A S)</u> (6 D A S) (5 B S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 16

Slide 5.4.17

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.

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	<u>(2 A S)</u> (5 B S)
3	<u>(4 C A S)</u> (6 D A S) (5 B S)
4	<u>(6 D A S)</u> (5 B S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 17

Slide 5.4.18

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.

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	<u>(2 A S)</u> (5 B S)
3	<u>(4 C A S)</u> (6 D A S) (5 B S)
4	<u>(6 D A S)</u> <u>(5 B S)</u>
5	<u>(6 D B S)</u> (10 G B S) (6 D A S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 18

Slide 5.4.19

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.

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 19

Slide 5.4.20

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.

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 20

Slide 5.4.21

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

### Uniform Cost

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 21

Slide 5.4.22

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).

### 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.

Spring 02 • 22

Slide 5.4.23

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.

Spring 02 • 23

### 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.

Spring 02 • 24

Slide 5.4.24

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).

Slide 5.4.25

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.

Spring 02 • 25

### Uniform Cost

Another (easier?) way to see it

Total path cost

UC enumerates paths in order of total path cost!

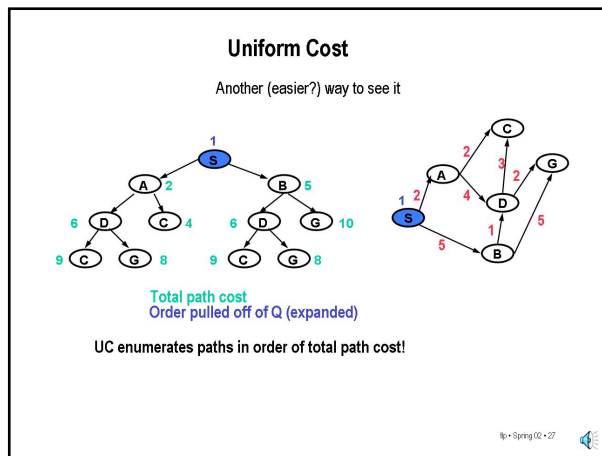
Spring 02 • 26

Slide 5.4.26

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.

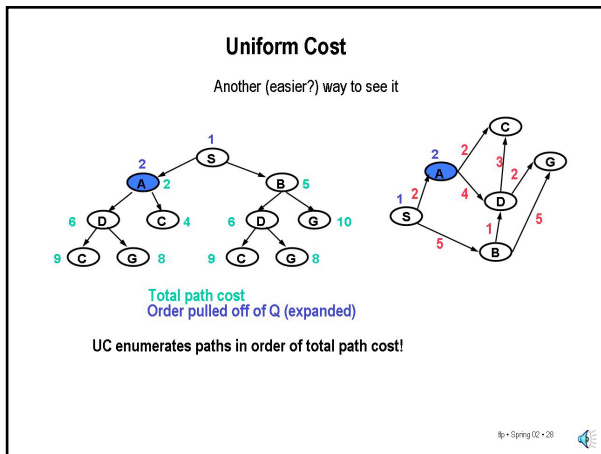
Slide 5.4.27

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



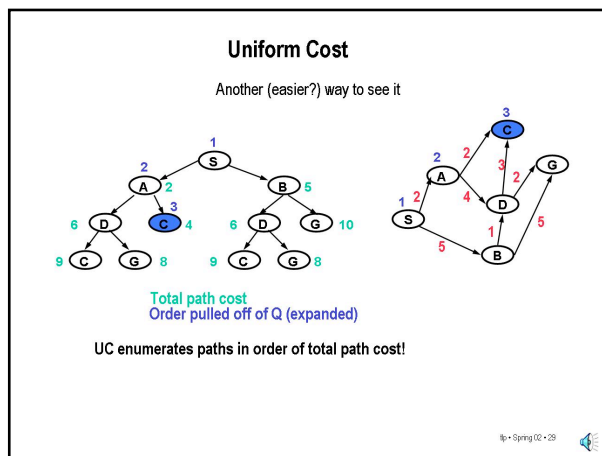
Slide 5.4.28

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



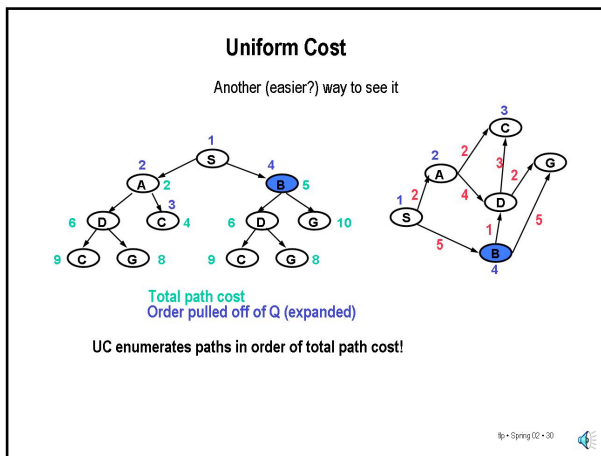
Slide 5.4.29

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



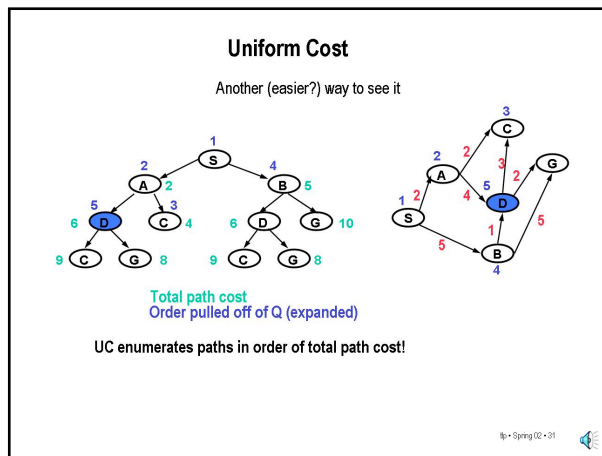
Slide 5.4.30

Then comes the path from S to B, with length 5.



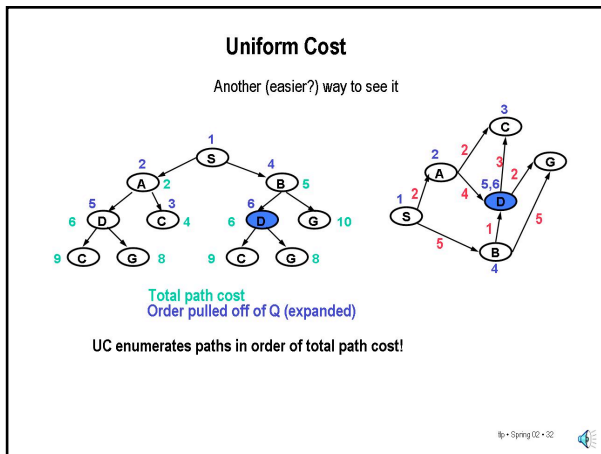
Slide 5.4.31

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



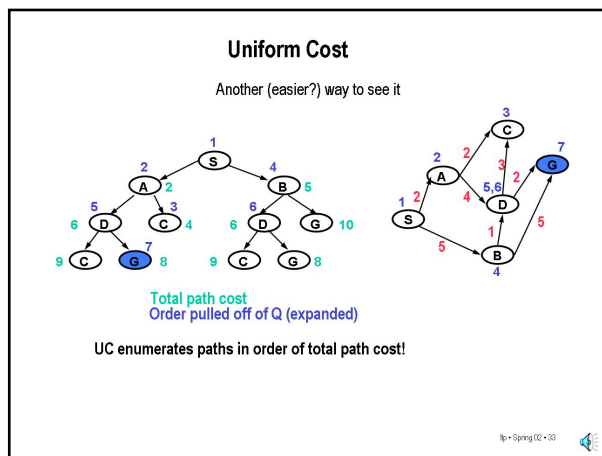
Slide 5.4.32

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



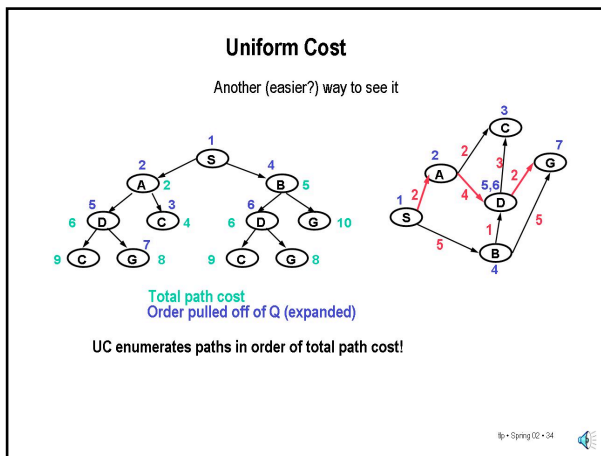
Slide 5.4.33

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 5.4.34

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.




## 6.034 Notes: Section 5.5

### Slide 5.5.1


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.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a node, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	A*	Uses path "length" measure and heuristic. Finds "shortest" path.

fp • Spring 02 • 1 

### Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.

fp • Spring 02 • 2 

### Slide 5.5.2


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 5.5.3

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.

### Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function  $h(N)$ , which is an **estimate (h)** of the distance from a state to the goal.

fp • Spring 02 • 3 

### Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function  $h(N)$ , which is an **estimate (h)** of the distance from a state to the goal.
- Instead of enumerating paths in order of just **length (g)**, enumerate paths in terms of  **$f = \text{estimated total path length} = g + h$** .

lp • Spring 02 • 4

### Slide 5.5.4

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 5.5.5

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

### Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function  $h(N)$ , which is an **estimate (h)** of the distance from a state to the goal.
- Instead of enumerating paths in order of just **length (g)**, enumerate paths in terms of  **$f = \text{estimated total path length} = g + h$** .
- An estimate that always underestimates the real path length to the goal is called **admissible**. For example, an estimate of 0 is admissible (but useless). Straight line distance is admissible estimate for path length in Euclidean space.

lp • Spring 02 • 5

### Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function  $h(N)$ , which is an **estimate (h)** of the distance from a state to the goal.
- Instead of enumerating paths in order of just **length (g)**, enumerate paths in terms of  **$f = \text{estimated total path length} = g + h$** .
- An estimate that always underestimates the real path length to the goal is called **admissible**. For example, an estimate of 0 is admissible (but useless). Straight line distance is admissible estimate for path length in Euclidean space.
- Use of an admissible estimate guarantees that UC will still find the shortest path.

lp • Spring 02 • 6

### Slide 5.5.6

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 ensure that our heuristic estimate is admissible. Note that straight-line distance is always an underestimate of path-length in Euclidean space. Of course, by our constraint on distances, the constant function 0 is always admissible (but useless).

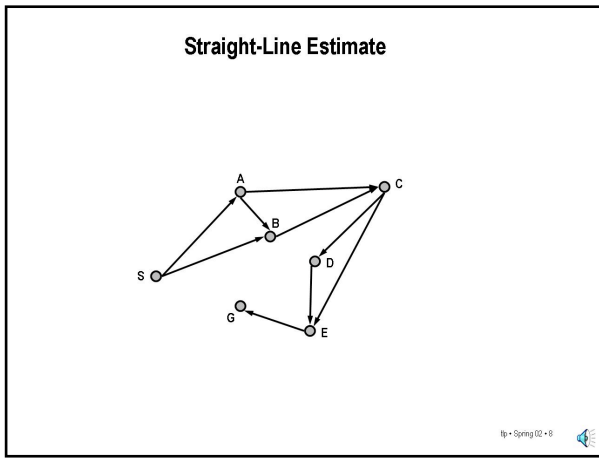
### Slide 5.5.7

UC using an admissible heuristic is known as  $A^*$  (A star). It is a very popular search method in AI.

### Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function  $h(N)$ , which is an **estimate (h)** of the distance from a state  $n$  to a goal.
- Instead of enumerating paths in order of just **length (g)**, enumerate paths in terms of  **$f = \text{estimated total path length} = g + h$** .
- An estimate that always underestimates the real path length to the goal is called **admissible**. For example, an estimate of 0 is admissible (but useless). Straight line distance is admissible estimate for path length in Euclidean space.
- Use of an admissible estimate guarantees that UC will still find the shortest path.
- UC with an admissible estimate is known as  $A^*$  (pronounced "A star") search.

lp • Spring 02 • 7

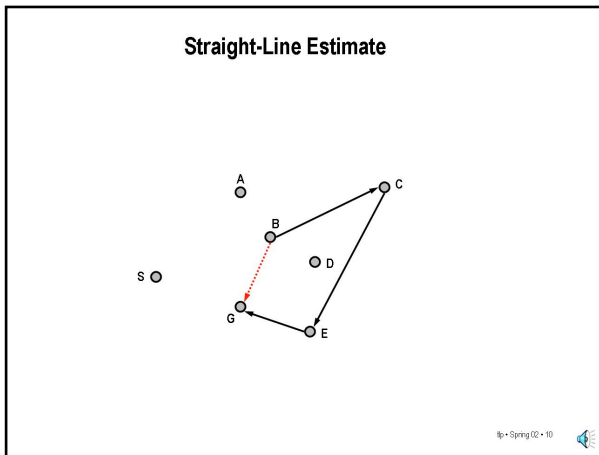
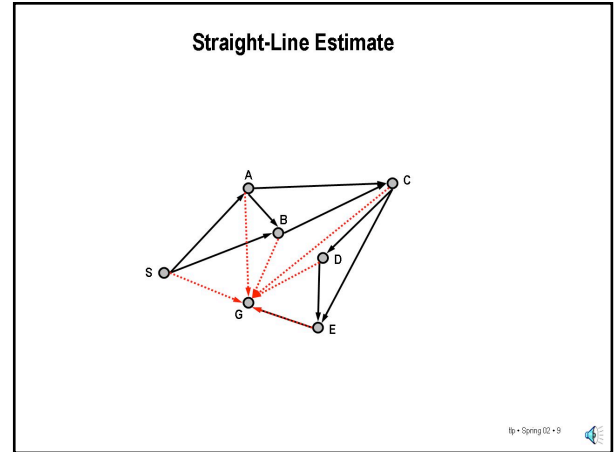


Slide 5.5.8

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.

Slide 5.5.9

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.



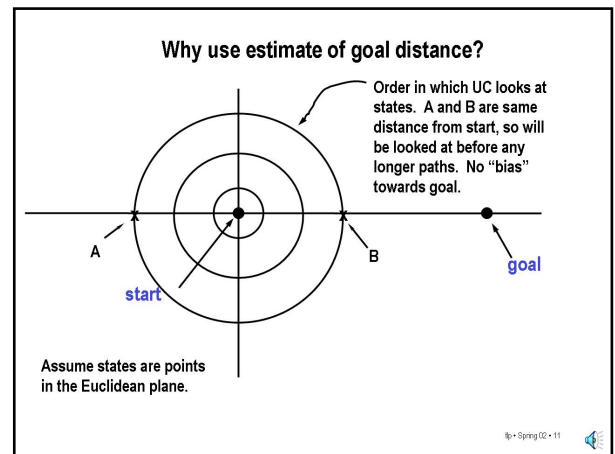
Slide 5.5.10

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.

Slide 5.5.11

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.





### Why use estimate of goal distance?

Order in which UC looks at states. A and B are same distance from start, so will be looked at before any longer paths. No "bias" towards goal.

Order of examination using dist. from start + estimate of dist. to goal. Note "bias" toward the goal; points away from goal look worse.

Assume states are points in the Euclidean plane.

Spring 02 • 12

Slide 5.5.12

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 5.5.13

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.

### A\*

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)

Heuristic Values  
A=2 C=1 S=0  
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 13

### A\*

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)
2 (4 A S) (8 B S)

Heuristic Values  
A=2 C=1 S=0  
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 14

Slide 5.5.14

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 5.5.15

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

### A\*

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)
2 (4 A S) (8 B S)
3 (5 C A S) (7 D A S) (8 B S)

Heuristic Values  
A=2 C=1 S=0  
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 15

**A\***

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)
2 (4 A S) (8 B S)
3 (5 C A S) (7 D A S) (8 B S)
4 (7 D A S) (8 B S)

Heuristic Values  
A=2 C=1 S=0  
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 16

**Slide 5.5.16**

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

**Slide 5.5.17**

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.

**A\***

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)
2 (4 A S) (8 B S)
3 (5 C A S) (7 D A S) (8 B S)
4 (7 D A S) (8 B S)
5 (8 G D A S) (10 C D A S) (8 B S)

Heuristic Values  
A=2 C=1 S=0  
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 17

**A\***

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)
2 (4 A S) (8 B S)
3 (5 C A S) (7 D A S) (8 B S)
4 (7 D A S) (8 B S)
5 (8 G D A S) (10 C D A S) (8 B S)

Heuristic Values  
A=2 C=1 S=0  
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 18

**Slide 5.5.18**

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

**Slide 5.5.19**

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.

**Not all heuristics are admissible**

Given the link lengths in the figure, is the table of heuristic values that we used in our earlier best-first example an admissible heuristic?

No!

- A is ok
- B is ok
- C is ok
- D is too big, needs to be <= 2
- S is too big, can always use 0 for start

Heuristic Values  
A=2 C=1 S=10  
B=3 D=4 G=0

Spring 02 • 19

### Admissible Heuristics

8 Puzzle: Move tiles to reach goal. Think of a move as moving "empty" tile.



Alternative underestimates of "distance" (number of moves) to goal:

1. Number of misplaced tiles (7 in example above)

Spring 02 • 20

### Slide 5.5.20

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.

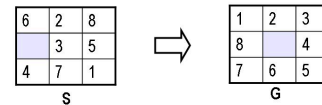
### Slide 5.5.21

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).

### Admissible Heuristics

8 Puzzle: Move tiles to reach goal. Think of a move as moving "empty" tile.



Alternative underestimates of "distance" (number of moves) to goal:

1. Number of misplaced tiles (7 in example above)
2. Sum of Manhattan distance of tile to its goal location (17 in example above). Manhattan distance between  $(x_1, y_1)$  and  $(x_2, y_2)$  is  $|x_1 - x_2| + |y_1 - y_2|$ . Each move can only decrease the distance of exactly one tile.

The second of these is much better at predicting actual number of moves.

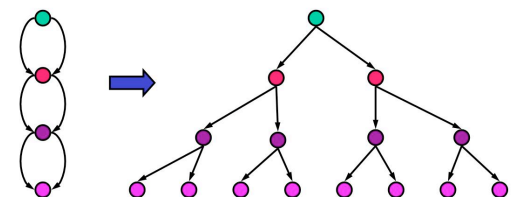
Spring 02 • 21

## 6.034 Notes: Section 5.6

### Slide 5.6.1

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.

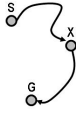
### States vs Paths



Spring 02 • 1

### Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".



lp • Spring 02 • 2

#### Slide 5.6.2

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 5.6.3

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.

### Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.

lp • Spring 02 • 3

### Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.
- Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.

lp • Spring 02 • 4

#### Slide 5.6.4

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 5.6.5

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.

### Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.
- Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.
- So, once expand one path to state X, we don't need to consider (extend) any other paths to X. We can keep a list of these states, call it Expanded. If the state of the search node we pull off of Q is in the Expanded list, we discard the node. When we use the Expanded list this way, we call it "strict".

lp • Spring 02 • 5

### Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.
- Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.
- So, once we expand one path to state X, we don't need to consider (extend) any other paths to X. We can keep a list of these states, call it Expanded. If the state of the search node we pull off of Q is in the Expanded list, we discard the node. When we use the Expanded list this way, we call it "strict".
- Note that UC without this is still correct, but inefficient for searching graphs.

lp • Spring 02 • 6

### Slide 5.6.6

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 5.6.7

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...

### Simple Optimal Search Algorithm Uniform Cost

A search node is a path from some state X to the start state, e.g., (X B A S)  
The state of a search node is the most recent state of the path, e.g. X.  
Let Q be a list of search nodes, e.g. ((X B A S) (C B A S) ...).  
Let S be the start state.

- Initialize Q with search node (S) as only entry;
- If Q is empty, fail. Else, pick least cost search node N from Q
- If state(N) is a goal, return N (we've reached the goal)
- (Otherwise) Remove N from Q.
- 
- Find all the children of state(N) and create all the one-step extensions of N to each descendant.
- Add all the extended paths to Q;
- Go to step 2.

lp • Spring 02 • 7

### Simple Optimal Search Algorithm Uniform Cost + Strict Expanded List

A search node is a path from some state X to the start state, e.g., (X B A S)  
The state of a search node is the most recent state of the path, e.g. X.  
Let Q be a list of search nodes, e.g. ((X B A S) (C B A S) ...).  
Let S be the start state.

- Initialize Q with search node (S) as only entry; set Expanded = ( )
- If Q is empty, fail. Else, pick least cost search node N from Q
- If state(N) is a goal, return N (we've reached the goal)
- (Otherwise) Remove N from Q.
- if state(N) in Expanded, go to step 2, otherwise add state(N) to Expanded.
- Find all the children of state(N) (Not in Expanded) and create all the one-step extensions of N to each descendant.
- Add all the extended paths to Q; if descendant state already in Q, keep only shorter path to the state in Q.
- Go to step 2.

lp • Spring 02 • 8

### Slide 5.6.8

... 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.

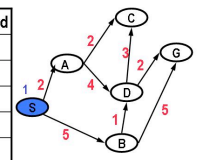
### Slide 5.6.9

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.

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	



Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

lp • Spring 02 • 8

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 10

**Slide 5.6.10**

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

**Slide 5.6.11**

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.

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S
3	<u>(4 C A S)</u> (6 D A S) (5 B S)	S,A

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 11

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S
3	<u>(4 C A S)</u> (6 D A S) (5 B S)	S,A
4	(6 D A S) ( <u>5 B S</u> )	S,A,C

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 12

**Slide 5.6.12**

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 5.6.13**

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.

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S
3	<u>(4 C A S)</u> (6 D A S) (5 B S)	S,A
4	(6 D A S) ( <u>5 B S</u> )	S,A,C
5	<del>(6 D B S)</del> (10 G B S) (6 D A S)	S,A,C,B

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 13

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	Expanded
1 (0 S)	
2 (2 A S) (5 B S)	S
3 (4 C A S) (6 D A S) (5 B S)	S, A
4 (6 D A S) (5 B S)	S, A, C
5 (6 D B S) (10 G B S) (6 D A S)	S, A, C, B
6 (8 G D A S) (9 C D A S) (10 G B S)	S, A, C, B, D

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 14

**Slide 5.6.14**

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 5.6.15**

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?

### Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	Expanded
1 (0 S)	
2 (2 A S) (5 B S)	S
3 (4 C A S) (6 D A S) (5 B S)	S, A
4 (6 D A S) (5 B S)	S, A, C
5 (6 D B S) (10 G B S) (6 D A S)	S, A, C, B
6 (8 G D A S) (9 C D A S) (10 G B S)	S, A, C, B, D

Added paths in blue; underlined paths are chosen for extension.  
We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 15

### A\* (without expanded list)

- Let  $g(N)$  be the path cost of  $n$ , where  $n$  is a search tree node, i.e. a partial path.
- Let  $h(N)$  be  $h(\text{state}(N))$ , the heuristic estimate of the remaining path length to the goal from state  $(N)$ .
- Let  $f(N) = g(N) + h(\text{state}(N))$  be the total estimated path cost of a node, i.e. the estimate of a path to a goal that starts with the path given by  $N$ .
- A\* picks the node with lowest  $f$  value to expand

Spring 02 • 16

**Slide 5.6.16**

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 5.6.17**

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.

### A\* (without expanded list)

- Let  $g(N)$  be the path cost of  $n$ , where  $n$  is a search tree node, i.e. a partial path.
- Let  $h(N)$  be  $h(\text{state}(N))$ , the heuristic estimate of the remaining path length to the goal from state  $(N)$ .
- Let  $f(N) = g(N) + h(\text{state}(N))$  be the total estimated path cost of a node, i.e. the estimate of a path to a goal that starts with the path given by  $N$ .
- A\* picks the node with lowest  $f$  value to expand
- A\* (without expanded list) and with admissible heuristic is guaranteed to find optimal paths -- those with smallest path cost.

Spring 02 • 17

### A\* and the strict Expanded List

- 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.

Spring 02 • 18

### Slide 5.6.18

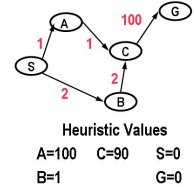
If we use the search algorithm we used for uniform-cost search with a strict Expanded list for A\*, adding in an admissible heuristic to the path length, then we can no longer guarantee that it will always find the optimal path. We need a stronger condition on the heuristics used than being an underestimate.

### Slide 5.6.19

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

### A\* and the strict Expanded List

- 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.
- 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.

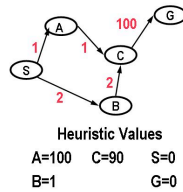


Spring 02 • 19

### A\* and the strict Expanded List

- 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.
- 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.

Q	Expanded
1 (0 S)	
2 (3 B S) (101 A S)	S
3 (94 C B S) (101 A S)	B, S
4 (101 A S) (104 G C B S)	C, B, S
5 (104 G C B S)	A, C, B, S



Added paths in blue; underlined paths are chosen for extension. We show the paths in reversed order; the node's state is the first entry.

Spring 02 • 20

### Slide 5.6.20

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 5.6.21

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.

### Consistency

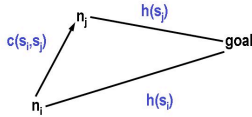
- To enable implementing A\* using the strict Expanded list, H needs to satisfy the following **consistency** (also known as **monotonicity**) conditions.
  - $h(s_g) = 0$ , if  $s_g$  is a goal
  - $h(s_i) - h(s_j) \leq c(s_i, s_j)$ , for  $n_j$  a child of  $n_i$

Spring 02 • 21



### Consistency

- To enable implementing A\* using the strict Expanded list, H needs to satisfy the following **consistency** (also known as **monotonicity**) conditions.
  - $h(s_i) = 0$ , if  $n_i$  is a goal
  - $h(s_i) - h(s_j) \leq c(s_i, s_j)$ , for  $n_j$  a child of  $n_i$
- That is, the heuristic cost in moving from one entry to the next cannot decrease by more than the arc cost between the states. This is a kind of *triangle inequality*. This condition is a highly desirable property of a heuristic function and often simply assumed (more on this later).



Spring 02 • 22

### Slide 5.6.22

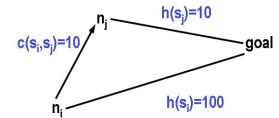
The best way of visualizing the consistency condition is as a "triangle inequality", that is, one side of the triangle is less than or equal the sum of the other two sides, as seen on the diagram here.

### Slide 5.6.23

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.

### Consistency Violation

- A simple example of a violation of consistency.
- $h(s_i) - h(s_j) > c(s_i, s_j)$
- In example,  $100 - 10 > 10$
- 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.



Spring 02 • 23

### A\* (without expanded list)

- Let  $g(N)$  be the path cost of  $n$ , where  $n$  is a search tree node, i.e. a partial path.
- Let  $h(N)$  be  $h(\text{state}(N))$ , the heuristic estimate of the remaining path length to the goal from state  $(N)$ .
- Let  $f(N) = g(N) + h(\text{state}(N))$  be the total estimated path cost of a node, i.e. the estimate of a path to a goal that starts with the path given by  $n$ .
- A\* picks the node with lowest  $f$  value to expand
- A\* (without expanded list) and with admissible heuristic is guaranteed to find optimal paths – those with the smallest path cost.
- This is true even if heuristic is NOT consistent.

Spring 02 • 24

### Slide 5.6.24

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.

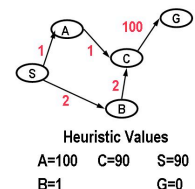
### Slide 5.6.25

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.

### A\* (without expanded list)

Note that heuristic is admissible but not consistent

	Q
1	(90 S)
2	(3 B S) (101 A S)
3	(94 C B S) (101 A S)
4	(101 A S) (104 G C B S)
5	(92 C A S) (104 G C B S)
6	(102 G C A S) (104 G C B S)



Added paths in blue; underlined paths are chosen for extension.

Spring 02 • 25

**A\* (with strict expanded list)**

- Just like Uniform Cost search.
- When a node N is expanded, if state(N) is in expanded list, discard N, else add state(N) to expanded list.
- If some node in Q has the same state as some descendant of N, keep only node with smaller f, which will also correspond to smaller g.
- For A\* (with strict expanded list) to be guaranteed to find the optimal path, the heuristic must be consistent.

Spring 02 • 26

**Slide 5.6.26**

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.

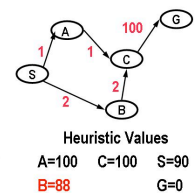
**Slide 5.6.27**

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.

**A\* (with strict expanded list)**

Note that this heuristic is admissible and consistent

Q	Expanded
1 (90 S)	
2 (90 B S) (101 A S)	S
3 (101 A S) (104 C B S)	A, S
4 (102 C A S) (104 C B S)	C, A, S
5 (102 G C A S)	G, C, A, S



Added paths in blue; underlined paths are chosen for extension.

Spring 02 • 27

**Dealing with inconsistent heuristic**

- What can we do if we have an inconsistent heuristic but we still want optimal paths?

Spring 02 • 28

**Slide 5.6.28**

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 5.6.29**

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

**Dealing with inconsistent heuristic**

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify A\* so that it detects and corrects when inconsistency has led us astray:

Spring 02 • 29

### Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify  $A^*$  so that it detects and corrects when inconsistency has led us astray:
- Assume we are adding  $node_1$  to Q and  $node_2$  is present in Expanded list with  $node_1.state = node_2.state$ .

Spring 02 • 30

### Slide 5.6.30

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 5.6.31

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

### Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify  $A^*$  so that it detects and corrects when inconsistency has led us astray:
- Assume we are adding  $node_1$  to Q and  $node_2$  is present in Expanded list with  $node_1.state = node_2.state$ .
- **Strict –**
  - do not add  $node_1$  to Q

Spring 02 • 31

### Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify  $A^*$  so that it detects and corrects when inconsistency has led us astray:
- Assume we are adding  $node_1$  to Q and  $node_2$  is present in Expanded list with  $node_1.state = node_2.state$ .
- **Strict –**
  - do not add  $node_1$  to Q
- **Non-Strict Expanded list –**
  - If  $node_1.path\_length < node_2.path\_length$ , then
    - Delete  $node_2$  from Expanded list
    - Add  $node_1$  to Q

Spring 02 • 32

### Slide 5.6.32

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 5.6.33

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.

### Worst Case Complexity

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.

Spring 02 • 33

### Worst Case Complexity

- The number of states in the search space may be exponential in some “depth” parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited or expanded lists, may have to visit (or expand) each state in the worst case.
- So, all searches will have worst case complexities that are at least proportional to the number of states and therefore exponential in the “depth” parameter.
- This is the bottom-line irreducible worst case cost of systematic searches.

Spring 02 • 34

### Slide 5.6.34

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 5.6.35

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.

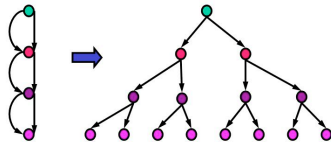
### Worst Case Complexity

- The number of states in the search space may be exponential in some “depth” parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited or expanded lists, may have to visit (or expand) each state in the worst case.
- So, all searches will have worst case complexities that are at least proportional to the number of states and therefore exponential in the “depth” parameter.
- This is the bottom-line irreducible worst case cost of systematic searches.
- Without memory of what states have been visited (expanded), searches can do (much) worse than visit every state.

Spring 02 • 35

### 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.



Spring 02 • 36

### Slide 5.6.36

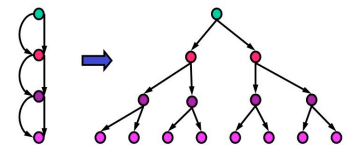
We've seen this example before. It shows that a state space with N states can generate a search tree with  $2^N$  nodes.

### Slide 5.6.37

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

### 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.

Spring 02 • 37

### Optimality & Worst Case Complexity

Algorithm	Heuristic	Expanded List	Optimality Guaranteed?	Worst Case # Expansions
Uniform Cost	None	Strict	Yes	N
A*	Admissible	None	Yes	>N
A*	Consistent	Strict	Yes	N
A*	Admissible	Strict	No	N
A*	Admissible	Non Strict	Yes	>N

N is number of states in graph

lp • Spring 02 • 38

#### Slide 5.6.38

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

## 6.034 Notes: Section 5.7

#### Slide 5.7.1

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
  - Algorithmic issues for A\*
- These are not required and are provided for those interested in pursuing these topics.

lp • Spring 02 • 1

#### Optimality of A\*

- Assume A\* has expanded a path to goal node G

lp • Spring 02 • 2

#### Slide 5.7.2

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 5.7.3

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.

### Optimality of A\*

- Assume A\* has expanded a path to goal node G
- Then, A\* has expanded all nodes N where  $f(N) < f(G)$ . Since h is admissible,  $f(G) = g(G)$ . So, every unexpanded node has  $f(N) \geq g(G)$ .

lp • Spring 02 • 3



### Optimality of A\*

- Assume A\* has expanded a path to goal node G
- Then, A\* has expanded all nodes N where  $f(N) < f(G)$ . Since h is admissible,  $f(G) = g(G)$ . So, every unexpanded node has  $f(N) \geq g(G)$ .
- Since h is admissible, we know that any path through N that reaches a goal node  $G^0$  has value  $g(G^0) \geq f(N)$

lp • Spring 02 • 4



### Slide 5.7.4

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,  $g(G')$ .

### Optimality of A\*

- Assume A\* has expanded a path to goal node G
- Then, A\* has expanded all nodes N where  $f(N) < f(G)$ . Since h is admissible,  $f(G) = g(G)$ . So, every unexpanded node has  $f(N) \geq g(G)$ .
- Since h is admissible, we know that any path through N that reaches a goal node  $G^0$  has value  $g(G^0) \geq f(N)$
- So, for every unexpanded node N, we have  $g(G^0) \geq f(N) \geq g(G)$ . That is, any goal reachable from those nodes has a path that is at least as long as the one we found.

lp • Spring 02 • 5



### Slide 5.7.5

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.

### Impact of better heuristic

- Let  $h^*$  be the "perfect" heuristic – returns actual path cost to goal.

lp • Spring 02 • 6



### Slide 5.7.6

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 5.7.7

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  $h^*$ .

## Impact of better heuristic

- Let  $h^*$  be the "perfect" heuristic – returns actual path cost to goal.
- If  $h_1(N) < h_2(N) \leq h^*(N)$  for all non-goal nodes, then  $h_2$  is a better heuristic than  $h_1$

lp • Spring 02 • 7



## Impact of better heuristic

- Let  $h^*$  be the "perfect" heuristic – returns actual path cost to goal.
- If  $h_1(N) < h_2(N) \leq h^*(N)$  for all non-goal nodes, then  $h_2$  is a better heuristic than  $h_1$
- If  $A_1^*$  uses  $h_1$ , and  $A_2^*$  uses  $h_2$ , then every node expanded by  $A_2^*$  is also expanded by  $A_1^*$ 
  - $f_1(G) = f_2(G) = g(G)$ , so both  $A_1^*$  and  $A_2^*$  expand all nodes with  $f < g(G)$
  - $f_1(N) = g(N) + h_1(N) < f_2(N) = g(N) + h_2(N) \leq g(G)$

lp • Spring 02 • 8



## Slide 5.7.8

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 expand 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 5.7.9

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.

## Impact of better heuristic

- Let  $h^*$  be the "perfect" heuristic – returns actual path cost to goal.
- If  $h_1(N) < h_2(N) \leq h^*(N)$  for all non-goal nodes, then  $h_2$  is a better heuristic than  $h_1$
- If  $A_1^*$  uses  $h_1$ , and  $A_2^*$  uses  $h_2$ , then every node expanded by  $A_2^*$  is also expanded by  $A_1^*$ 
  - $f_1(G) = f_2(G) = g(G)$ , so both  $A_1^*$  and  $A_2^*$  expand all nodes with  $f < g(G)$
  - $f_1(N) = g(N) + h_1(N) < f_2(N) = g(N) + h_2(N) \leq g(G)$
- That is,  $A_1^*$  expands at least as many nodes as  $A_2^*$  and we say that  $A_2^*$  is better informed than  $A_1^*$ .

lp • Spring 02 • 9



## Impact of better heuristic

- Let  $h^*$  be the "perfect" heuristic – returns actual path cost to goal.
- If  $h_1(N) < h_2(N) \leq h^*(N)$  for all non-goal nodes, then  $h_2$  is a better heuristic than  $h_1$
- If  $A_1^*$  uses  $h_1$ , and  $A_2^*$  uses  $h_2$ , then every node expanded by  $A_2^*$  is also expanded by  $A_1^*$ 
  - $f_1(G) = f_2(G) = g(G)$ , so both  $A_1^*$  and  $A_2^*$  expand all nodes with  $f < g(G)$
  - $f_1(N) = g(N) + h_1(N) < f_2(N) = g(N) + h_2(N) \leq g(G)$
- That is,  $A_1^*$  expands at least as many nodes as  $A_2^*$  and we say that  $A_2^*$  is better informed than  $A_1^*$ .
- Note that  $A^*$  with any non-zero admissible heuristic is better informed (and therefore typically expands fewer nodes) than Uniform Cost search.

lp • Spring 02 • 10



## Slide 5.7.10

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 5.7.11

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.

**Consistency  $\rightarrow$  Non-decreasing  $f$**

$N_i \longrightarrow N_j$

- $N_j$  is a descendant of  $N_i$  in the search tree
- $f(N_i) = g(N_i) + h(\text{state}(N_i))$
- $f(N_j) = g(N_i) + h(\text{state}(N_j)) = g(N_i) + c(N_i, N_j) + h(\text{state}(N_j))$

Spring 02 • 11

**Consistency  $\rightarrow$  Non-decreasing  $f$**

$N_i \longrightarrow N_j$

- $N_j$  is a descendant of  $N_i$  in the search tree
- $f(N_i) = g(N_i) + h(\text{state}(N_i))$
- $f(N_j) = g(N_i) + h(\text{state}(N_j)) = g(N_i) + c(N_i, N_j) + h(\text{state}(N_j))$
- By consistency,  $h(\text{state}(N_i)) \leq h(\text{state}(N_j)) + c(N_i, N_j)$
- Then,  $f(N_i) \leq f(N_j)$
- Thus, when A\*, with a consistent heuristic, expands a node, all of its descendants have  $f$  values greater or equal to the expanded node (as do all the nodes left on Q). So, the  $f$  values of expanded nodes can never decrease.

Spring 02 • 12

## Slide 5.7.12

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 5.7.13

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.

**Non-decreasing  $f \rightarrow$  first path is optimal**

- A\* with consistent heuristic expands nodes N in non-decreasing order of  $f(N)$
- Then, when a node N is expanded, we have found the shortest path to the corresponding  $s = \text{state}(N)$

Spring 02 • 13

**Non-decreasing  $f \rightarrow$  first path is optimal**

- A\* with consistent heuristic expands nodes N in non-decreasing order of  $f(N)$
- Then, when a node N is expanded, we have found the shortest path to the corresponding  $s = \text{state}(N)$
- Imagine that we later found another node  $N^0$  with the same corresponding state  $s$  then we know that
  - $f(N^0) \geq f(N)$
  - $f(N) = g(N) + h(s)$
  - $f(N^0) = g(N^0) + h(s)$

Spring 02 • 14

## Slide 5.7.14

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 5.7.15

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!

### Non-decreasing $f \rightarrow$ first path is optimal

- $A^*$  with consistent heuristic expands nodes  $N$  in non-decreasing order of  $f(N)$
- Then, when a node  $N$  is expanded, we have found the shortest path to the corresponding  $s = \text{state}(N)$
- Imagine that we later found another node  $N^0$  with the same corresponding state  $s$  then we know that
  - $f(N^0) \geq f(N)$
  - $f(N) = g(N) + h(s)$
  - $f(N^0) = g(N^0) + h(s)$
- So, we can conclude that
  - $g(N^0) \geq g(N)$
- And we can safely ignore the second path to  $s$  as we would with the strict Expanded list.

Spring 02 • 15



### 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.

Spring 02 • 16



## Slide 5.7.16

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 5.7.17

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.

### 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  $O(N^2)$ .

Spring 02 • 17



**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  $O(N^2)$ .Nodes taken from Q ?  $O(N)$ 

Spring 02 • 18

**Slide 5.7.18**

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 5.7.19**

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.

**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  $O(N^2)$ .Nodes taken from Q ?  $O(N)$ Cost of picking a node from Q using linear scan?  $O(N)$ 

Spring 02 • 19

**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  $O(N^2)$ .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)?  $O(L)$ 

Spring 02 • 20

**Slide 5.7.20**

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 5.7.21**

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

**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  $O(N^2)$ .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 ?  $O(1)$ 

Spring 02 • 21

**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 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<sup>2</sup>).

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)
<b>Total cost ?</b>	<b>O(N<sup>2</sup> + L)</b>

Spring 02 • 22

**Slide 5.7.22**

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

**Slide 5.7.23**

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.

**Should we use a Priority Queue?**

- A priority queue is a data structure that makes it efficient to identify the "best" element of a set. A PQ is typically implemented as a balanced tree.
- The time to find best element in a PQ grows as O(log N) for a set of size N. This is very much better than N for large N. Also, note that even if we don't discard paths to Expanded nodes, the access is still O(log N), since O(log N<sup>2</sup>)=O(log N).

Spring 02 • 23**Should we use a Priority Queue?**

- A priority queue is a data structure that makes it efficient to identify the "best" element of a set. A PQ is typically implemented as a balanced tree.
- The time to find best element in a PQ grows as O(log N) for a set of size N. This is very much better than N for large N. Also, note that even if we don't discard paths to Expanded nodes, the access is still O(log N), since O(log N<sup>2</sup>)=O(log N).
- However, adding elements to a PQ also has time that grows as O(log N).
- Our algorithm does up to N "find best" operations and it does up to L "add" operations. If Q is a PQ, then cost is O(N\*log N + L\*log N)

Spring 02 • 24**Slide 5.7.24**

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 5.7.25**

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

**Should we use a Priority Queue?**

- A priority queue is a data structure that makes it efficient to identify the "best" element of a set. A PQ is typically implemented as a balanced tree.
- The time to find best element in a PQ grows as O(log N) for a set of size N. This is very much better than N for large N. Also, note that even if we don't discard paths to Expanded nodes, the access is still O(log N), since O(log N<sup>2</sup>)=O(log N).
- However, adding elements to a PQ also has time that grows as O(log N).
- Our algorithm does up to N "find best" operations and it does up to L "add" operations. If Q is a PQ, then cost is O(N\*log N + L\*log N)
- If graph is dense, and L is O(N<sup>2</sup>), then a PQ is not advisable.
- If graph is sparse (the more common case), and L is O(N), then a PQ is highly desirable.

Spring 02 • 25

## Cost and Performance

Searching a tree with  $N$  nodes and  $L$  links

Search Method	Worst Time (Dense)	Worst Time (Sparse)	Worst Space	Guaranteed to find shortest path
Uniform Cost $A^*$	$O(N^2)$	$O(N \log N)$	$O(N)$	Yes

Searching a tree with branching factor  $b$  and depth  $d$   
 $L = N = b^{d+1}$

Worst case time is proportional to number of nodes created  
 Worst case space is proportional to maximal length of  $Q$  (and Expanded)

Spring 02 '06

### Slide 5.7.26

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.