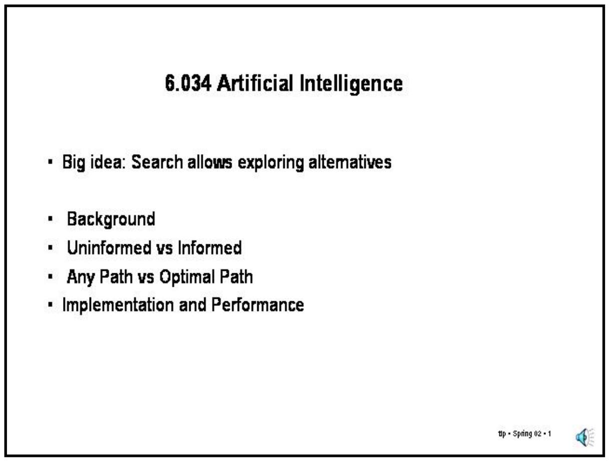
6.034 Notes: Section 2.1

Slide 2.1.1

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.

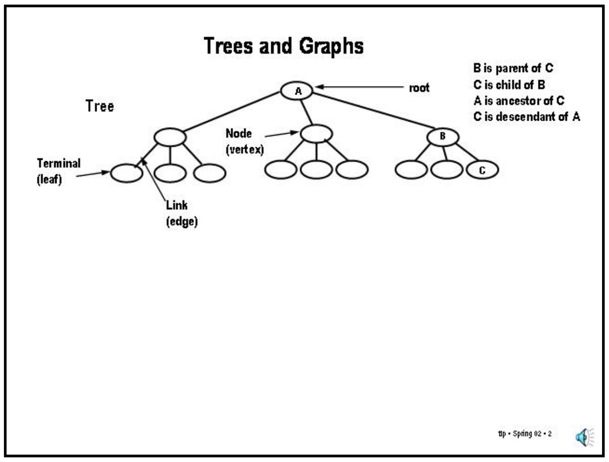
We will start with some background, terminology and basic implementation strategies and then cover four classes of search algorithms, which differ along two dimensions: First, is the difference between **uninformed** (also known as **blind**) search and then **informed** (also known as **heuristic**) 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

tlp • Spring 02 • 1



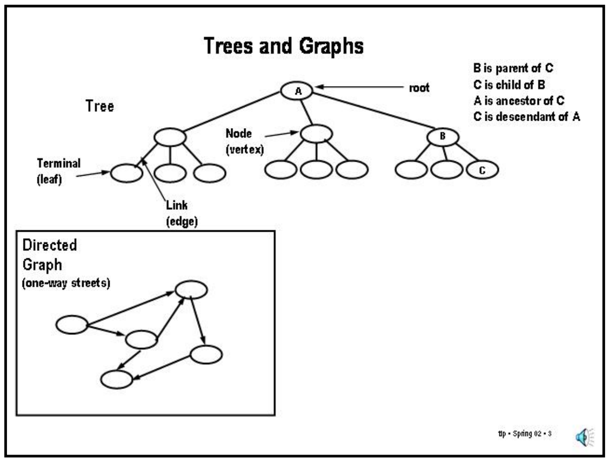
Slide 2.1.2

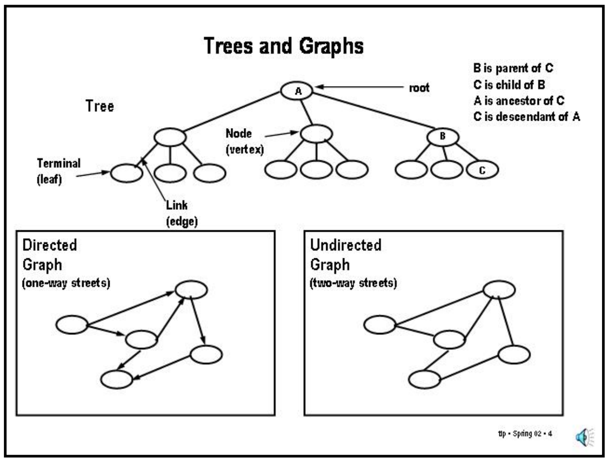
The search methods we will be dealing with are defined on trees and graphs, so we need to fix on some terminology for these structures:

- 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 2.1.3

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).



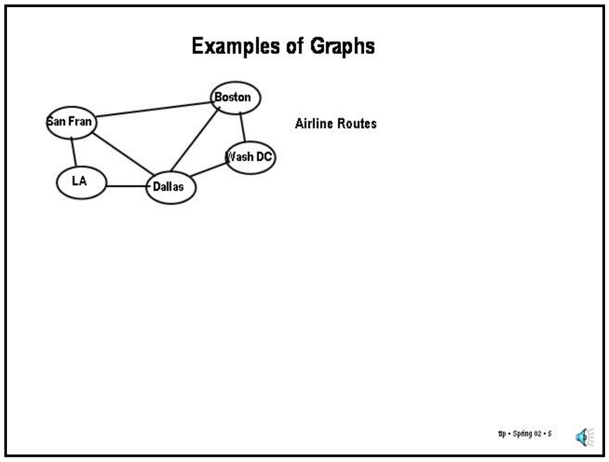


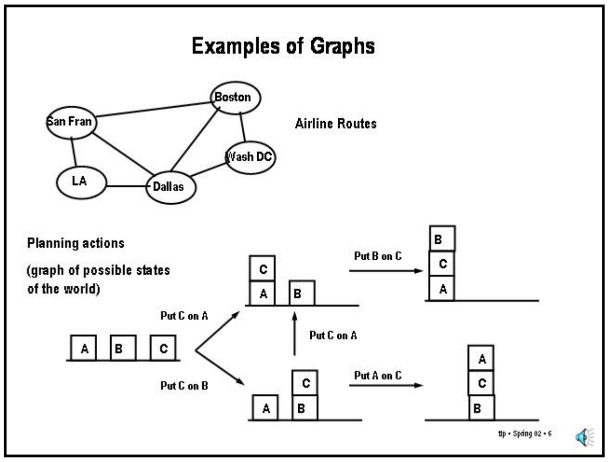
Slide 2.1.4

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 2.1.5

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 2.1.6

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.

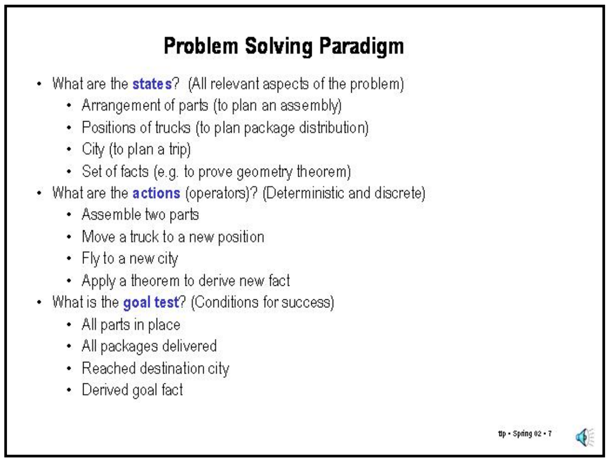
A path through such a graph (from a start node to a goal node) is a "plan of action" to achieve some desired goal state from some known starting state. It is this type of graph that is of more general interest in AI.

Slide 2.1.7

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).



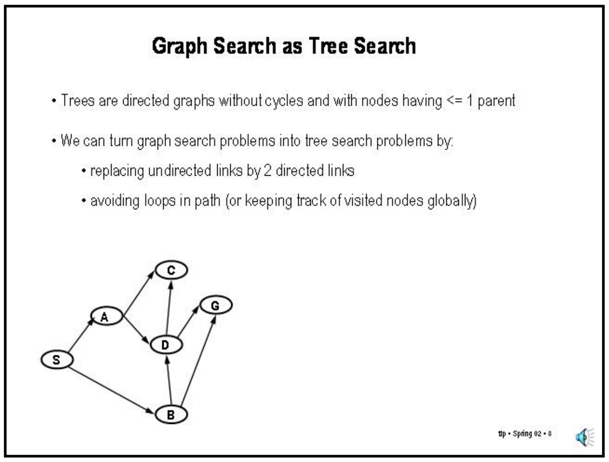
Problem Solving Paradigm

- 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

tlp • Spring 02 • 7

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 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 <= 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)

tlp • Spring 02 • 8

Slide 2.1.8

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.

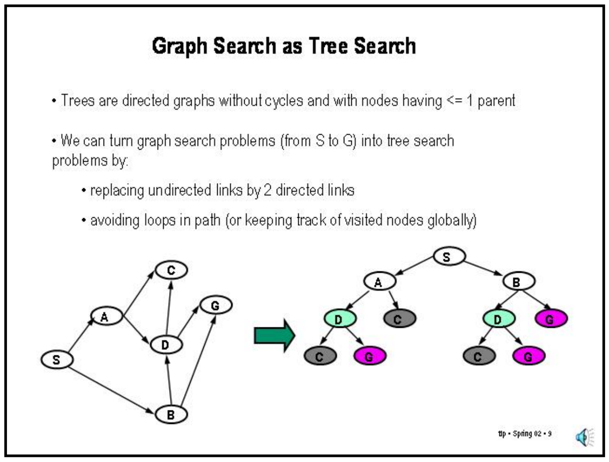
When asked to search a graph, we can construct an equivalent problem of searching a tree by doing two things: turning undirected links into two directed links; and, more importantly, making sure we never consider a path with a loop or, even better, by never visiting the same node twice.

Slide 2.1.9

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.

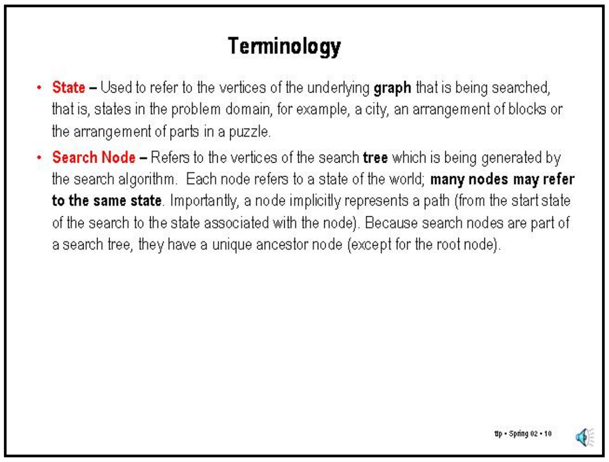
The issue of how much effort to place in avoiding loops and avoiding extra visits to nodes is an important one that we will revisit later when we discuss the various search algorithms.



Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having <= 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)

tlp • 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).

tlp • Spring 02 • 10

Slide 2.1.10

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

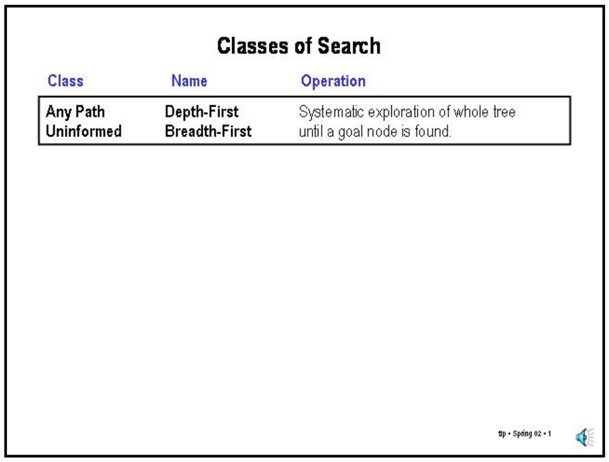
A state is an arrangement of the real world (or at least our model of it). We assume that there is an underlying "real" state graph that we are searching (although it might not be explicitly represented in the computer; it may be implicitly defined by the actions). We assume that you can arrive at the 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 2.2

Slide 2.2.1

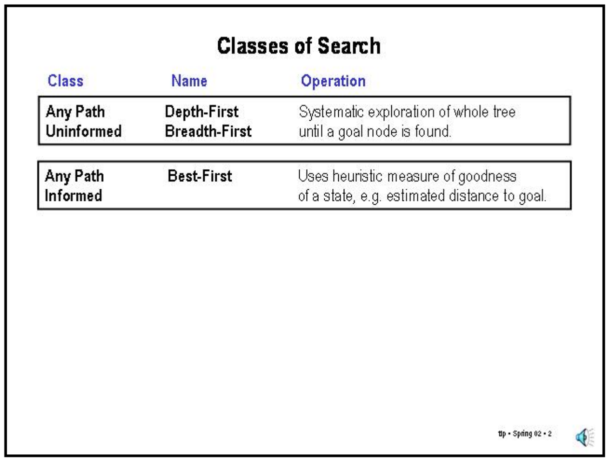
So, let's look at the different classes of search algorithms that we will be exploring. The simplest class is that of the **uninformed, any-path** algorithms. In particular, we will look at **depth-first** and **breadth-first** search. Both of these algorithms basically look at all the nodes in the search 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.

tlp • 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.

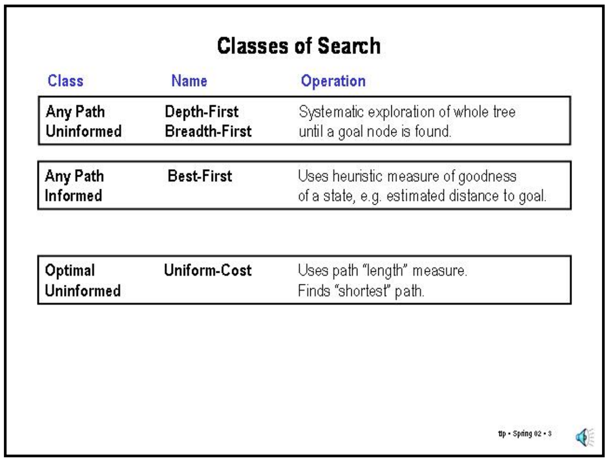
tlp • Spring 02 • 2

Slide 2.2.2

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 2.2.3

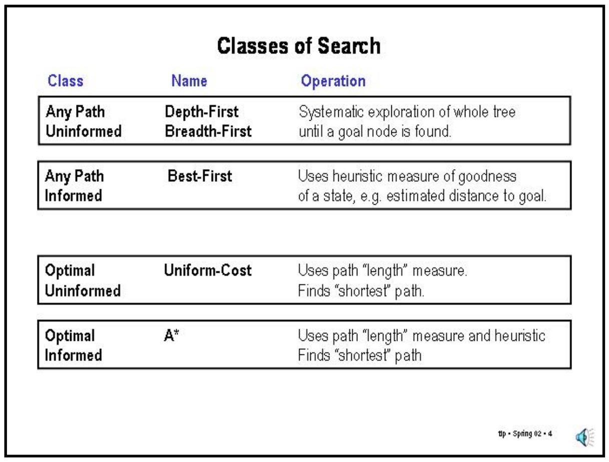
Next, we look at the class of **uninformed, optimal** algorithms. These methods guarantee finding the "best" path (as measured by the sum of weights on the graph edges) but do not use any information beyond what is in the graph definition.



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.

tlp • 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

tlp • Spring 02 • 4

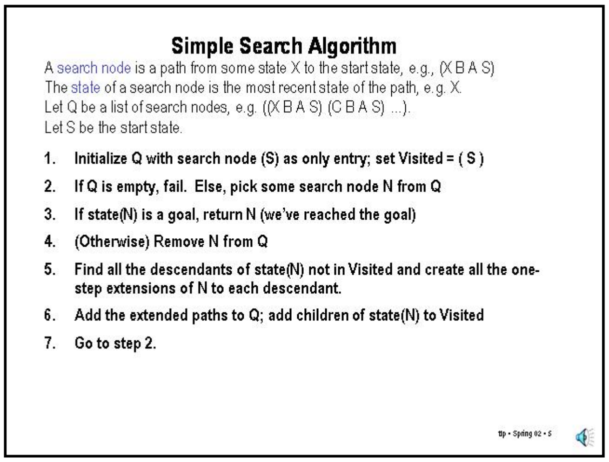
Slide 2.2.4

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 2.2.5

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.

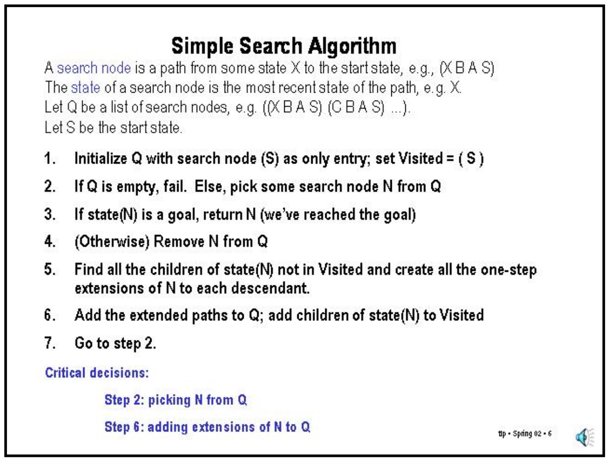


Simple Search Algorithm

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) ...).
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.

tlp • 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) ...).
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

tlp • Spring 02 • 6

Slide 2.2.6

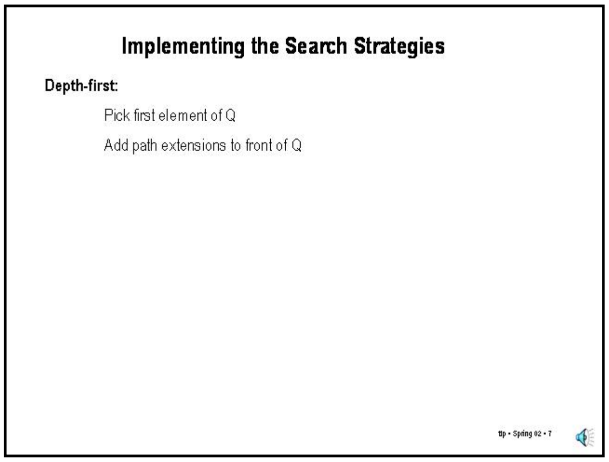
The key questions, of course, are *which* entry to pick off of Q and how precisely to add the new paths back onto Q. Different choices for these operations produce the various search strategies.

Slide 2.2.7

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.



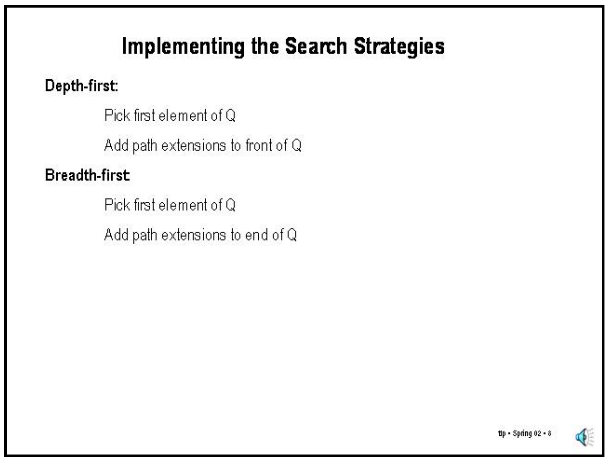
Implementing the Search Strategies

Depth-first:

Pick first element of Q

Add path extensions to front of Q

tlp • 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

tlp • Spring 02 • 8

Slide 2.2.8

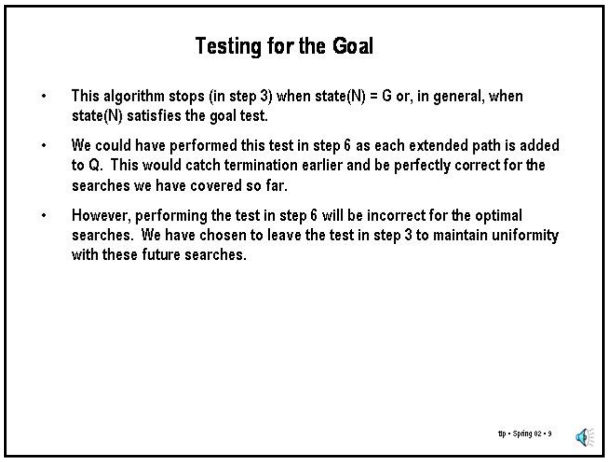
Breadth-first is the other major type of uninformed (or blind) search. The basic approach is to once again pick the first element of Q to examine BUT now we place the extended paths at the back of Q. This means that the next path pulled off of Q will typically not be a descendant of the current one,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

Slide 2.2.9

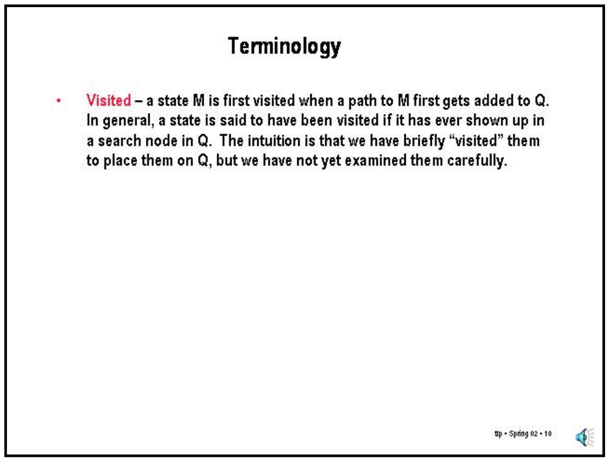
One subtle point is where in the algorithm one tests for success (that is, the goal test). There are two plausible points: one is when a path is extended and it reaches a goal, the other is when a path is pulled off of Q. We have chosen the latter (testing in step 3 of the algorithm) because it will 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 state(N) = G or, in general, when 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.

tlp • 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.

tlp • Spring 02 • 10

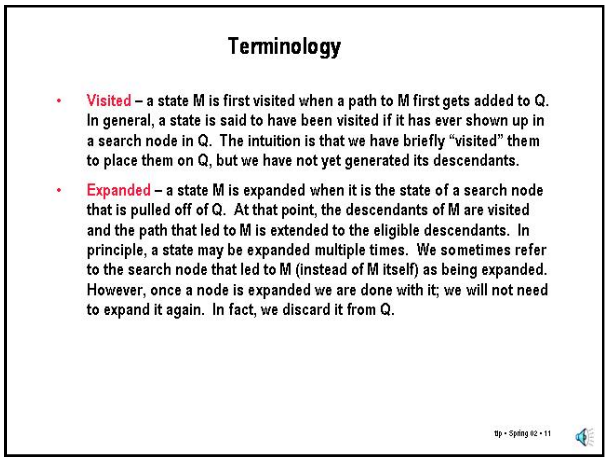
Slide 2.2.10

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 2.2.11

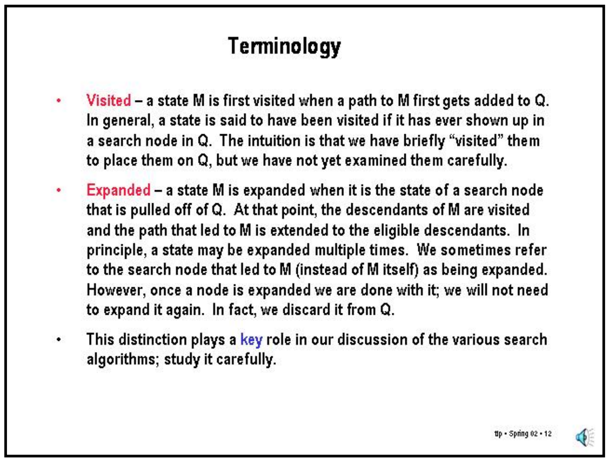
A state M is **Expanded** when a path to that state is pulled off of Q. At that point, the descendants of M are visited and the paths to those descendants added to the Q.



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.

tlp • 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.

tlp • Spring 02 • 12

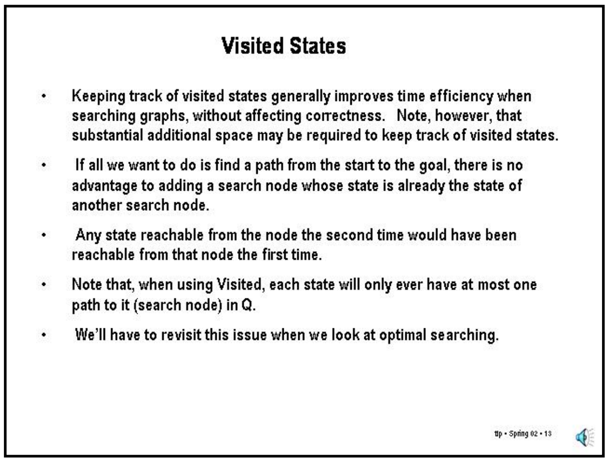
Slide 2.2.12

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

Slide 2.2.13

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.



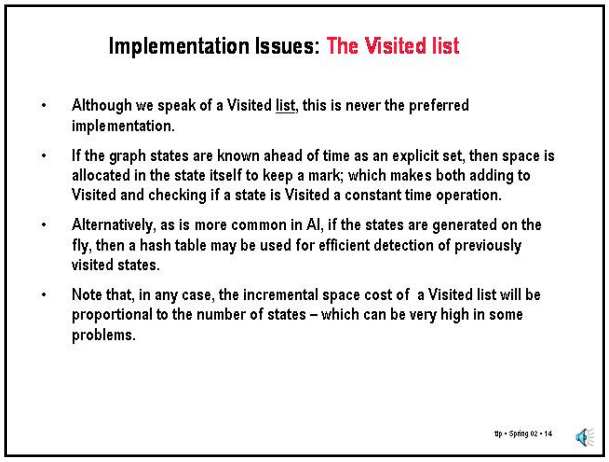
Visited States

- 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.

tlp • Spring 02 • 13

Slide 2.2.14

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.



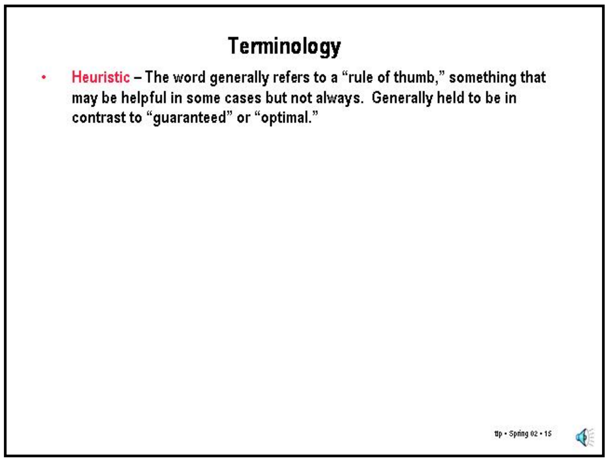
Implementation Issues: The Visited list

- 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.

tlp • Spring 02 • 14

Slide 2.2.15

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.



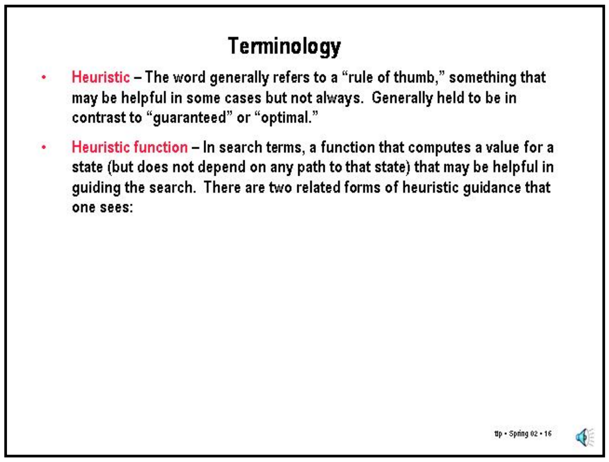
Terminology

- 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."

tlp • Spring 02 • 15

Slide 2.2.16

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.



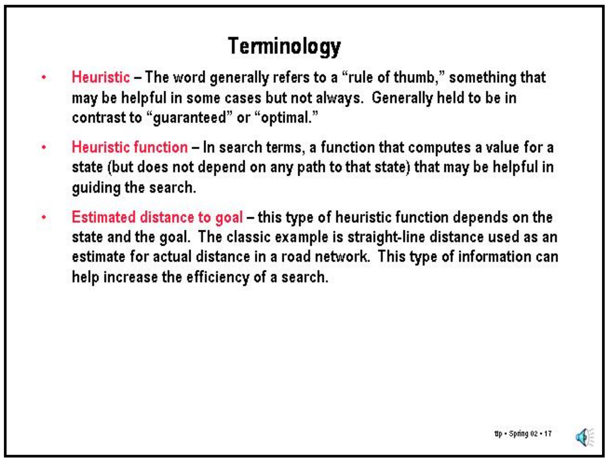
Terminology

- 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:

tlp • Spring 02 • 16

Slide 2.2.17

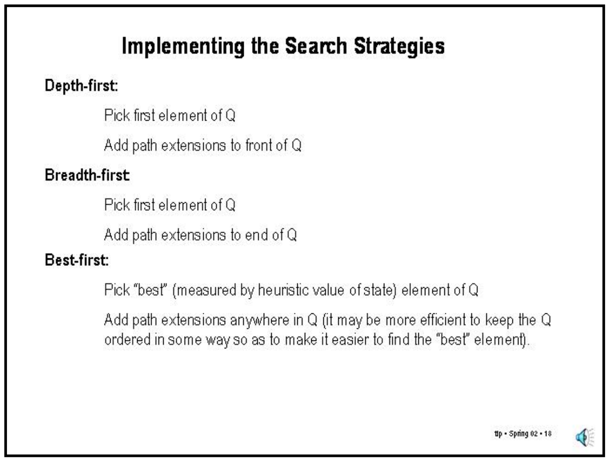
If we can get some estimate of the "distance" to a goal from the current node and we introduce a preference for nodes closer to the goal, then there is a good chance that the search will terminate more quickly. This intuition is clear when thinking about "airline" (as-the-crow-flies) distance to guide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



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.

tlp • 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).

tlp • Spring 02 • 18

Slide 2.2.18

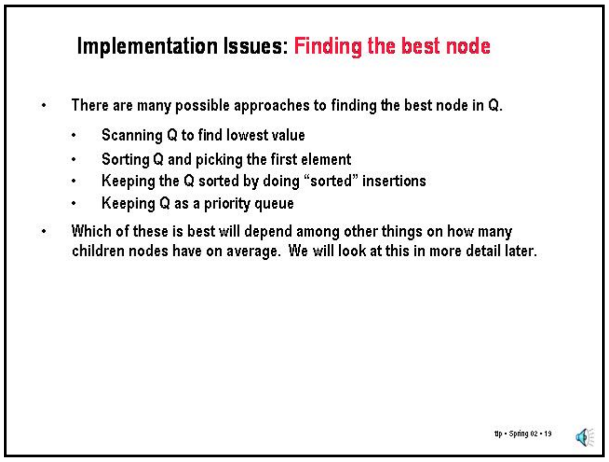
Best-first (also known as "greedy") search is a heuristic (informed) search that uses the value of a heuristic function defined on the states to guide the search. This will not guarantee finding a "best" path, for example, the shortest path to a goal. The heuristic is used in the hope that it will steer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

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

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

Slide 2.2.19

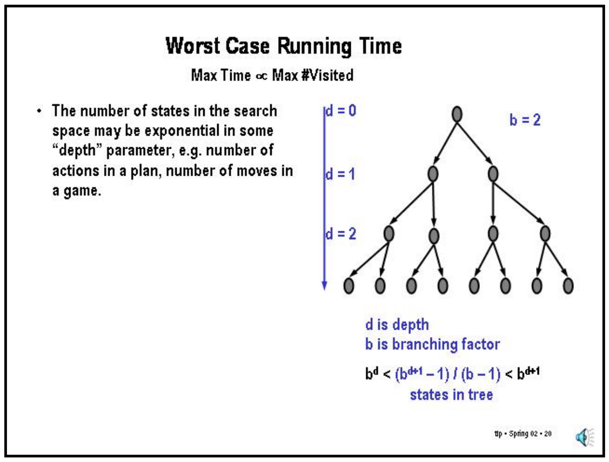
Note that best-first search requires finding the best node in Q. This is a classic problem in computer science and there are many different approaches that are appropriate in different circumstances. One simple method is simply to scan the Q completely, keeping track of the best element found. Surprisingly,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



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.

tlp • 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

tlp • Spring 02 • 20

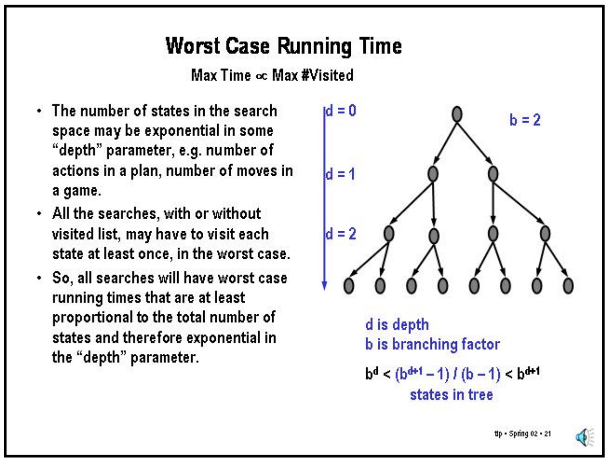
Slide 2.2.20

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 2.2.21

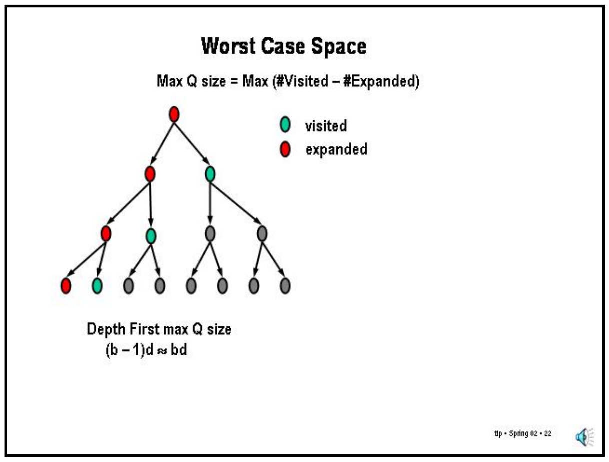
In a tree-structured search space, the nodes added to the search Q will simply correspond to the visited states. In the worst case, when the states are arranged in the worst possible way, all the search methods may end up having to visit or expand all of the states (up to some depth). In practice, we 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 ∝ 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.



Worst Case Space

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

Slide 2.2.22

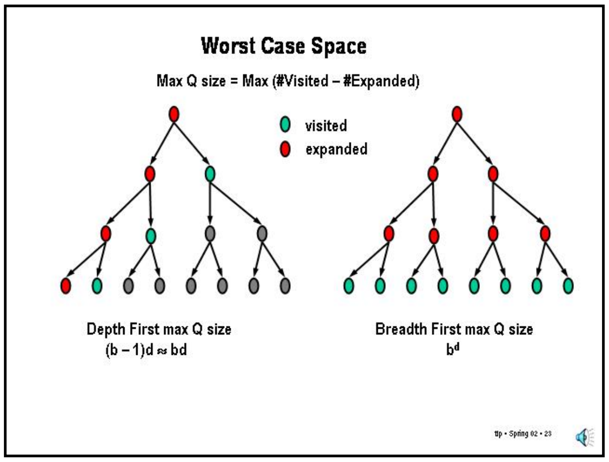
In addition to thinking about running time, we should also think about the memory space required for searches. The dominant factor in the space requirements for these searches is the maximum size of the search Q. The size of the search Q in a tree-structured search space is simply the number of visited 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 2.2.23

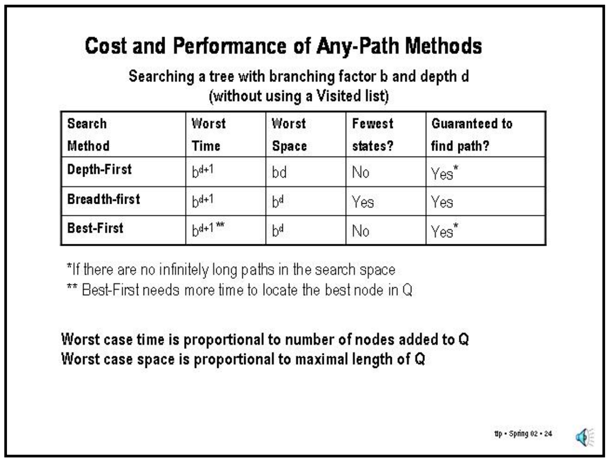
The situation for breadth-first search is much different than that for depth-first search. Here the worst case happens after we've visited all the nodes at depth d-1. At that point, all the nodes at depth d have been visited and none expanded. So, the Q has size b^d, that is, a size exponential in d.

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)



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

Slide 2.2.24

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.

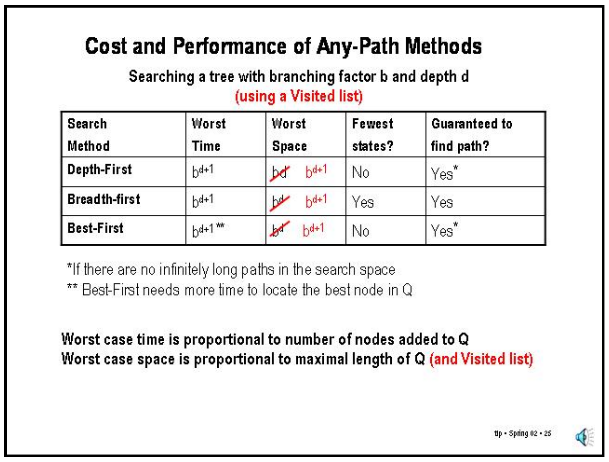
Recall that this analysis is done for searching a **tree** with uniform branching factor b and depth d. Therefore, the size of this search space grows exponentially with the depth. So, it should not be surprising that methods that guarantee finding a path will require exponential time in this situation. 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 2.2.25

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

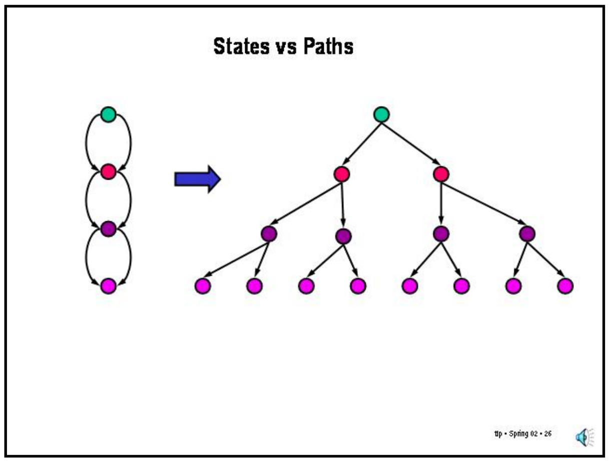
Searching a tree with branching factor b and depth d
(using a Visited list)

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

tlp • Spring 02 • 25



States vs Paths

tlp • Spring 02 • 26

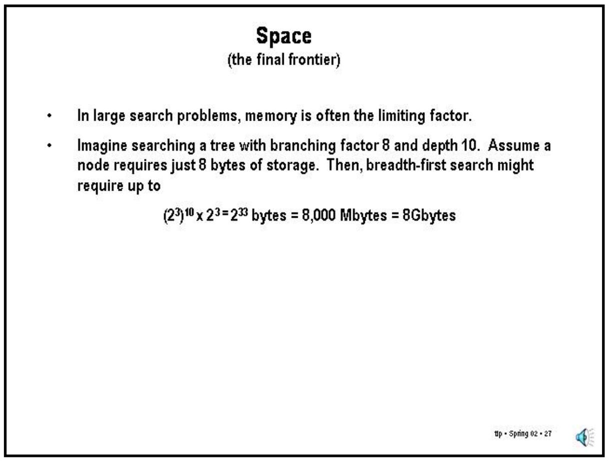
Slide 2.2.26

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 2.2.27

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.

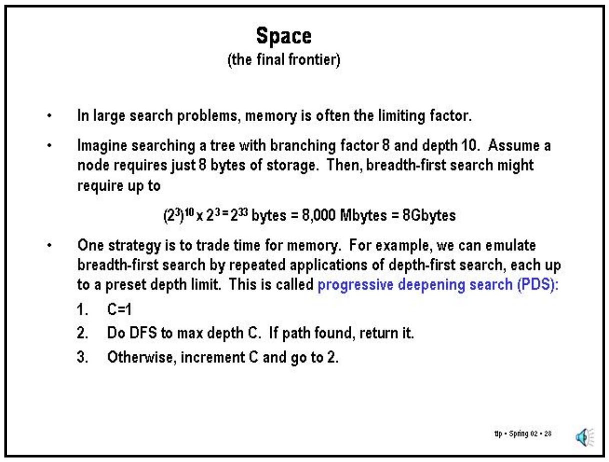


Space
(the final frontier)

- 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

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

tlp • 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

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

tlp • Spring 02 • 28

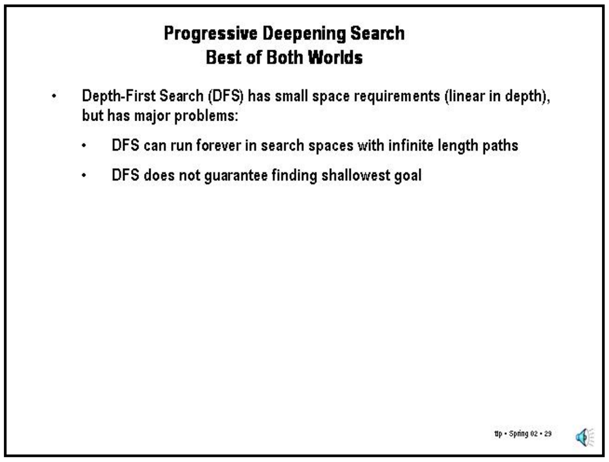
Slide 2.2.28

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 2.2.29

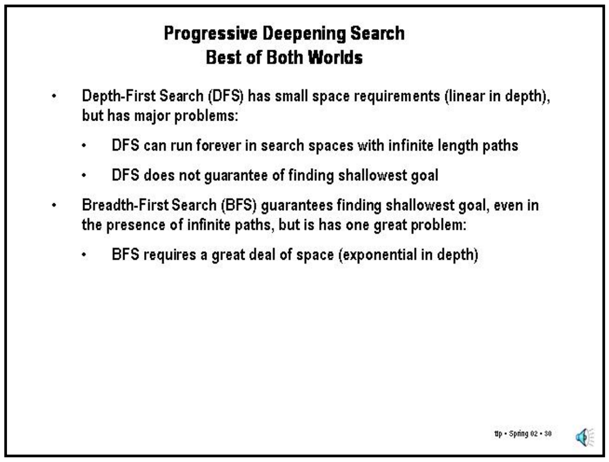
Depth-first search has one strong point - its limited space requirements, which are linear in the depth of the search tree. Aside from that there's not much that can be said for it. In particular, it is susceptible to "going off the deep-end", that is, chasing very deep (possibly infinitely deep) paths. 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

tlp • 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)

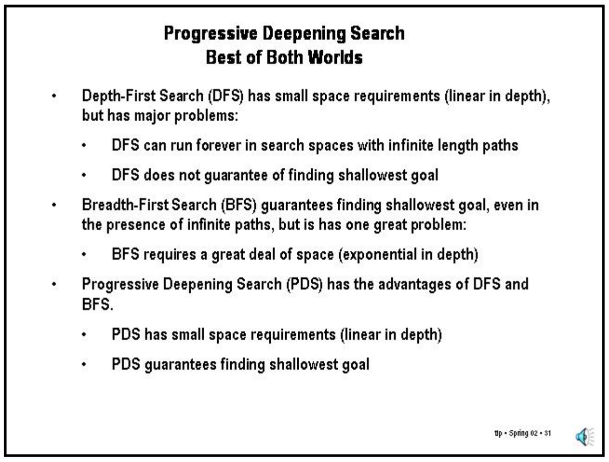
tlp • Spring 02 • 30

Slide 2.2.30

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 2.2.31

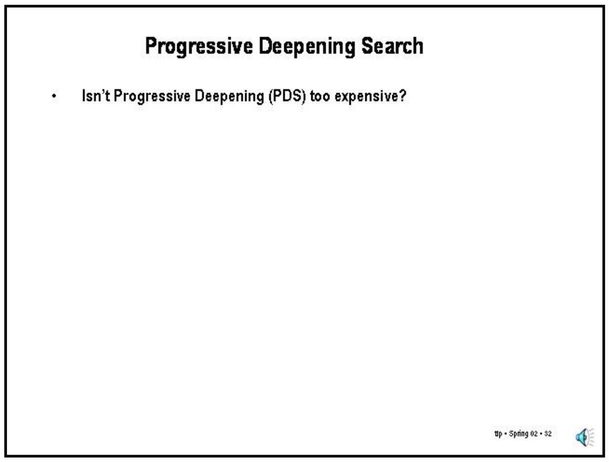
Progressive-deepening search, on the other other hand, has both limited space requirements of DFS and the strong optimality guarantee of BFS. Great! No?



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

tlp • Spring 02 • 31



Progressive Deepening Search

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

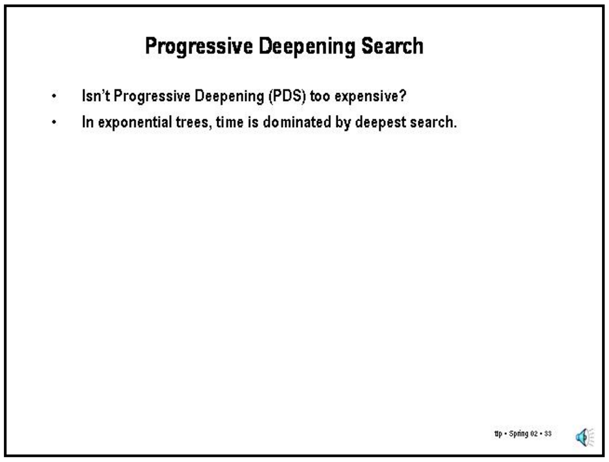
tlp • Spring 02 • 32

Slide 2.2.32

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 2.2.33

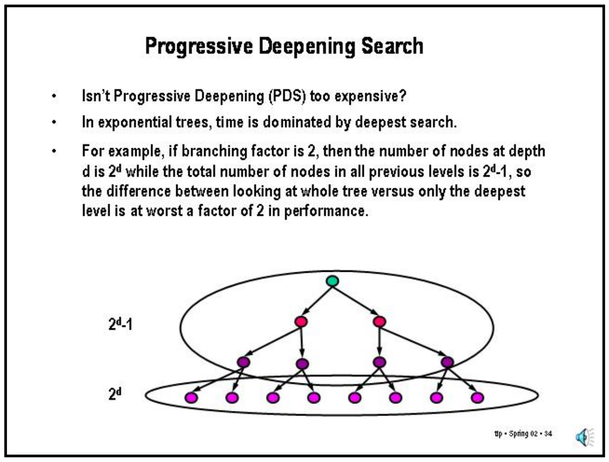
In small graphs, yes it is wasteful. But, if we really are faced with an exponentially growing space (in the depth), then it turns out that the work at the deepest level dominates the total cost.



Progressive Deepening Search

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

tlp • 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.

tlp • Spring 02 • 34

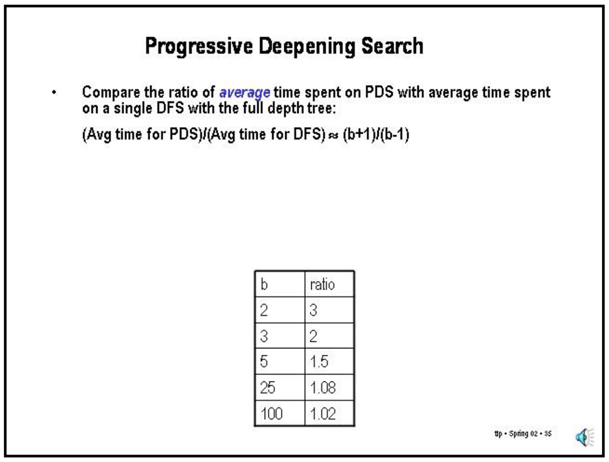
Slide 2.2.34

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 2.2.35

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).



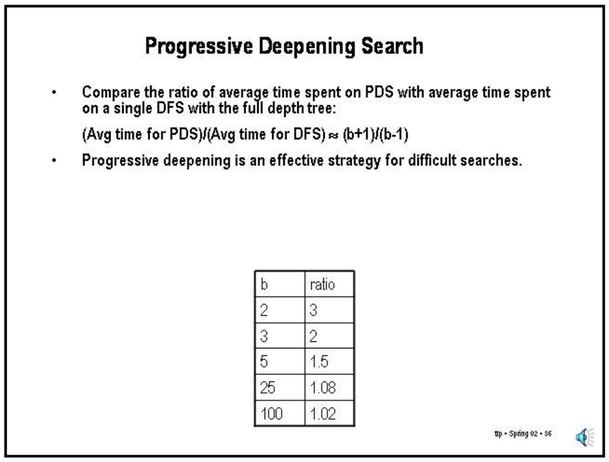
Progressive Deepening Search

- Compare the ratio of *average* time spent on PDS with average time spent on a single DFS with the full depth tree:

  (Avg time for PDS)/(Avg time for DFS) ≈ (b+1)/(b-1)

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

tlp • 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:

  (Avg time for PDS)/(Avg time for DFS) ≈ (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

tlp • Spring 02 • 36

Slide 2.2.36

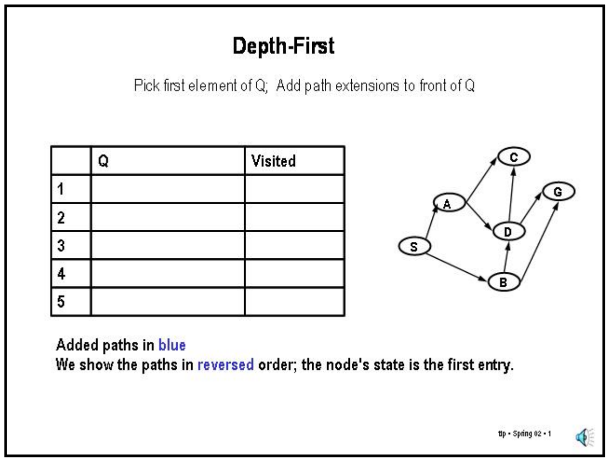
For many difficult searches, progressive deepening is in fact the only way to go. There are also progressive deepening versions of the optimal searches that we will see later, but that's beyond our scope.

6.034 Notes: Section 2.3

Slide 2.3.1

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

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



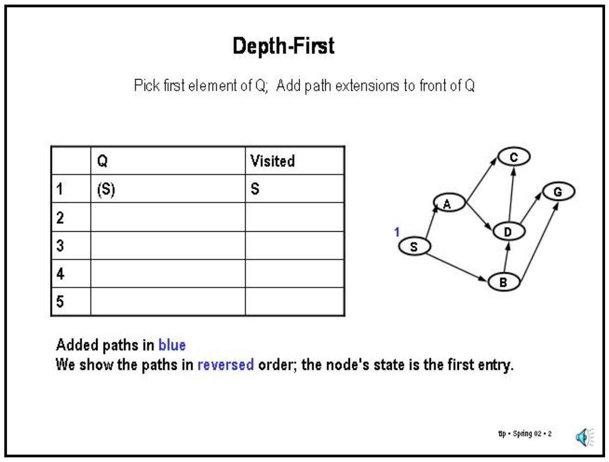
Depth-First

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.

tlp • 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.

tlp • Spring 02 • 2

Slide 2.3.2

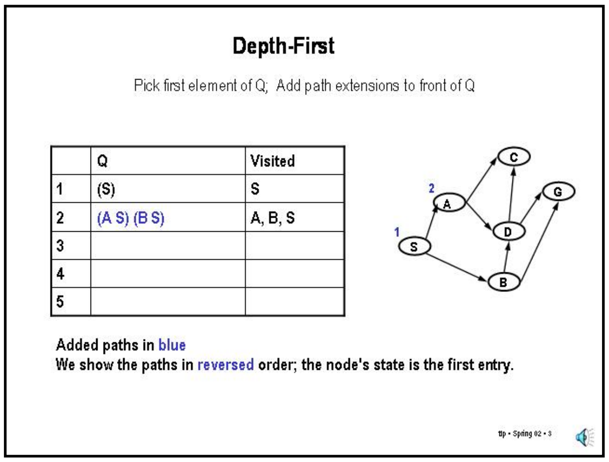
The first step is to initialize Q with a single node corresponding to the start state (S in this case) and the Visited list with the start state.

Slide 2.3.3

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.



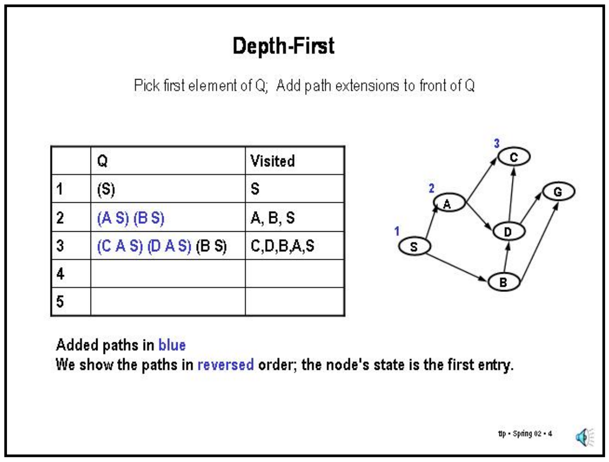
Depth-First

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.

tlp • Spring 02 • 3



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.

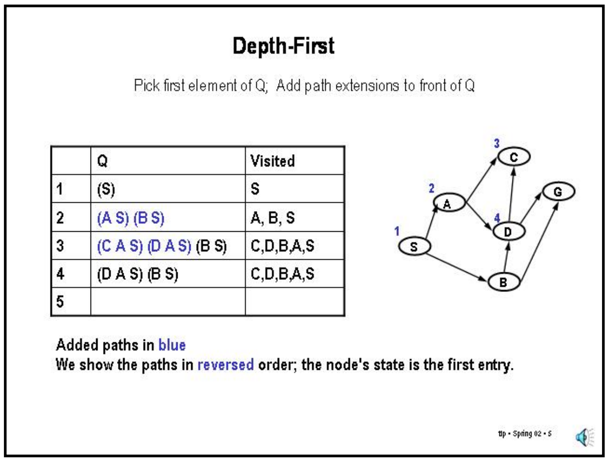
tlp • Spring 02 • 4

Slide 2.3.4

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.

Slide 2.3.5

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

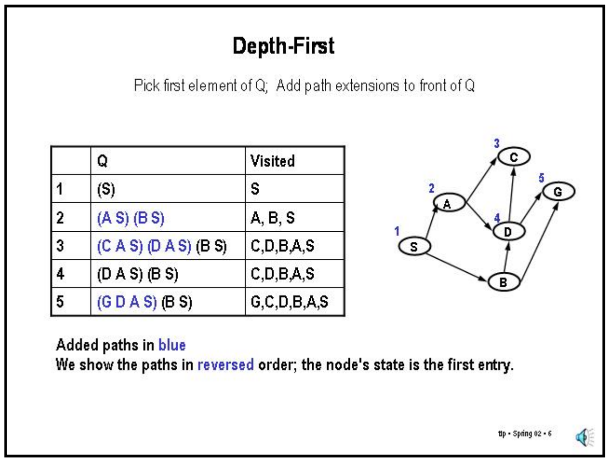
Pick first element of Q; Add path extensions to front of Q

	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		

1
S
2
A
3
C
4
D
G
B

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

tlp • Spring 02 • 5



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

1
S
2
A
3
C
4
D
5
G
B

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

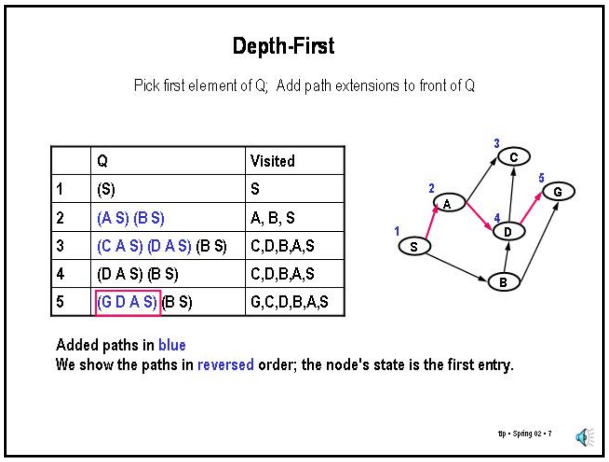
tlp • Spring 02 • 6

Slide 2.3.6

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

Slide 2.3.7

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



Depth-First

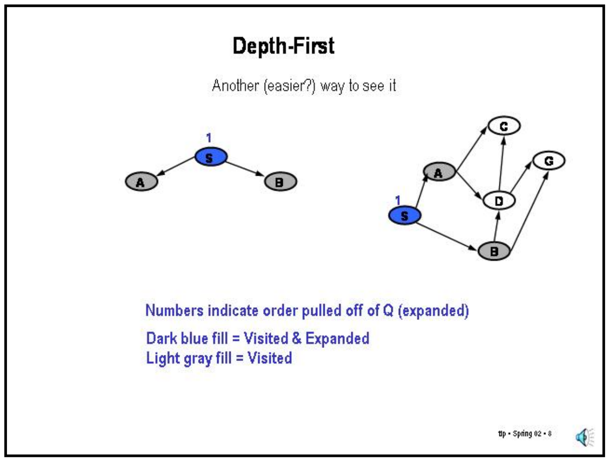
Pick first element of Q; Add path extensions to front of Q

	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.

tlp • Spring 02 • 7



Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

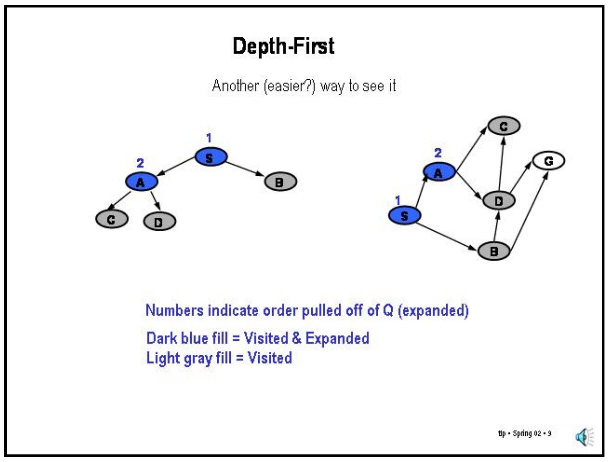
Light gray fill = Visited

Slide 2.3.8

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

Slide 2.3.9

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



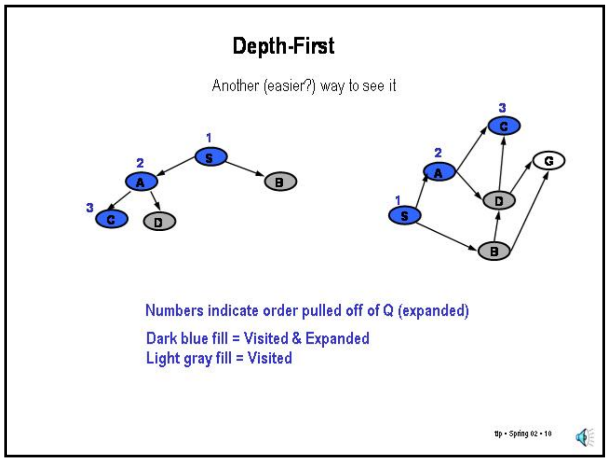
Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited



Depth-First

Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

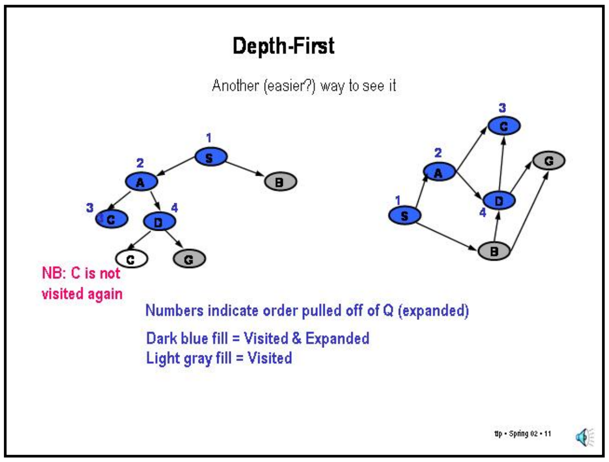
Light gray fill = Visited

Slide 2.3.10

We now expand the node corresponding to C, which has no descendants, so we cannot continue to go deeper. At this point, one talks about having to **back up** or **backtrack** to the parent node and expanding any unexpanded descendant nodes of the parent. If there were none at that level, we would continue to keep backing up to its parent and so on until an unexpanded node is found. We declare failure if we cannot find any remaining unexpanded nodes. In this case, we find an unexpanded descendant of A, namely D.

Slide 2.3.11

So, we expand D. Note that states C and G are both reachable from D. However, we have already visited C, so we do not add a node corresponding to that path. We add only the new node corresponding to the path to G.



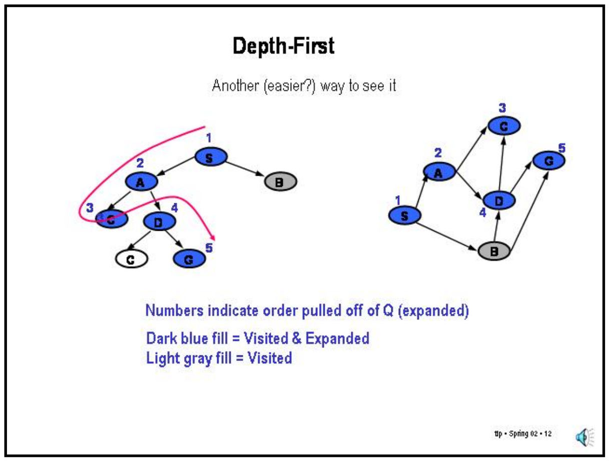
Depth-First

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



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

Slide 2.3.12

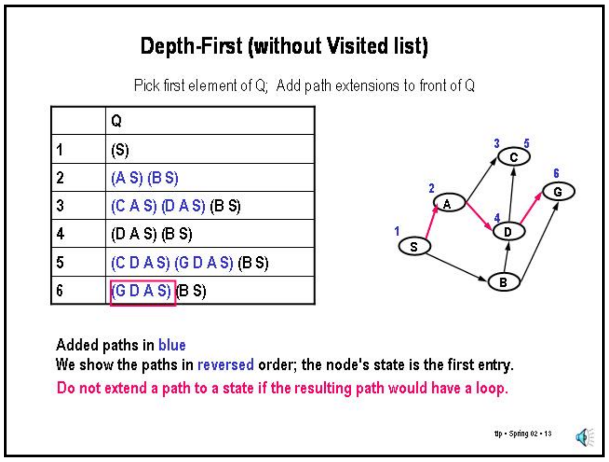
We now expand G and stop.

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

Slide 2.3.13

We can repeat the depth-first process without the Visited list and, as expected, one sees the second path to C added to Q, which was blocked by the use of the Visited list. I'll leave it as an exercise to go through the steps in detail.

Note that in the absence of a Visited list, we still require that we do not form any paths with loops, so if we have visited a state along a particular path, we do not re-visit that state again in any extensions of the path.



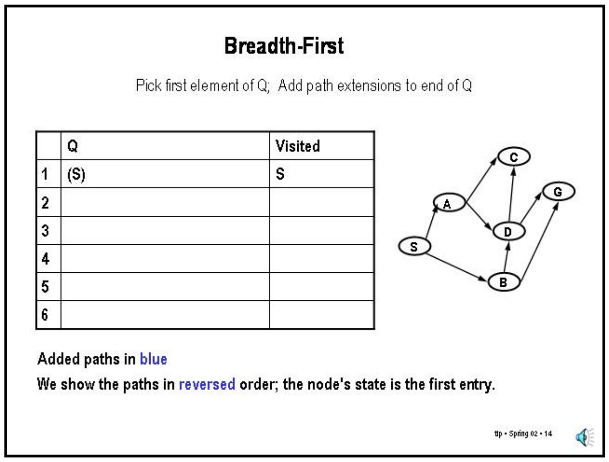
Depth-First (without Visited list)

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.

1
S
2
A
3 5
C
4
D
6
G
B
tlp • 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.

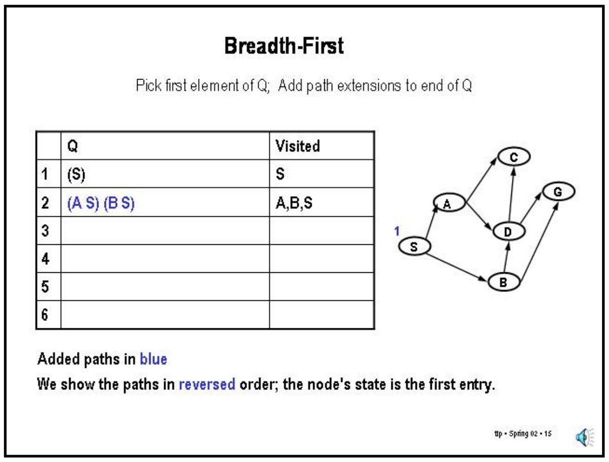
S
A
C
D
G
B
tlp • Spring 02 • 14

Slide 2.3.14

Let's look now at breadth-first search. The difference from depth-first search is that new paths are added to the back of Q. We start as with depth-first with the initial node corresponding to S.

Slide 2.3.15

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



Breadth-First

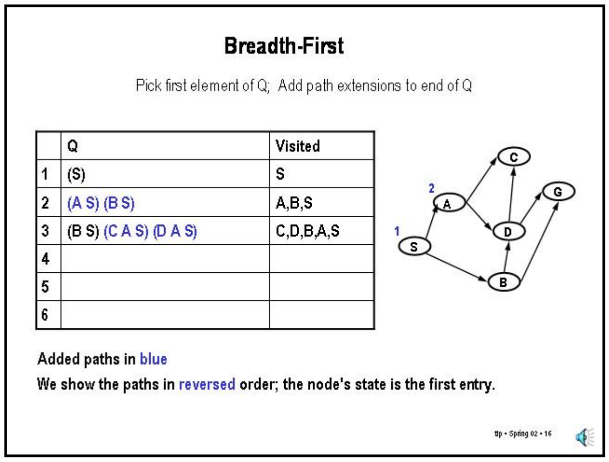
Pick first element of Q; Add path extensions to end of Q

	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.

tlp • 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.

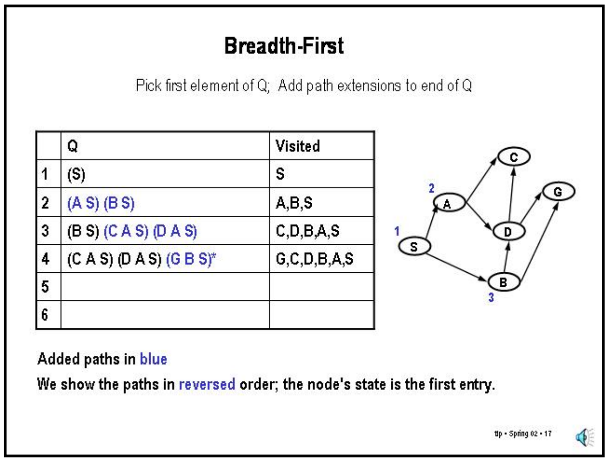
tlp • Spring 02 • 16

Slide 2.3.16

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

Slide 2.3.17

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



Breadth-First

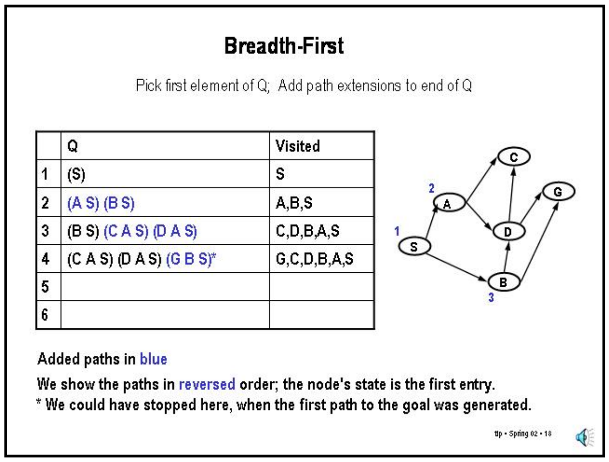
Pick first element of Q; Add path extensions to end of Q

	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.

tlp • 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.

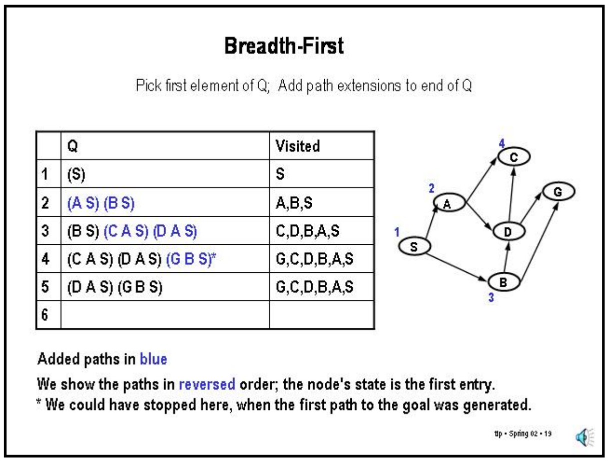
tlp • Spring 02 • 18

Slide 2.3.18

At this point, having generated a path to G, we would be justified in stopping. But, as we mentioned earlier, we proceed until the path to the goal becomes the first path in Q.

Slide 2.3.19

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



Breadth-First

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

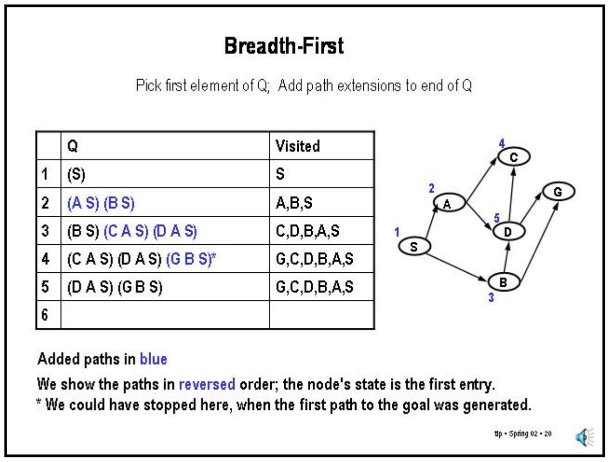
	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		

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.

tlp • Spring 02 • 19



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.

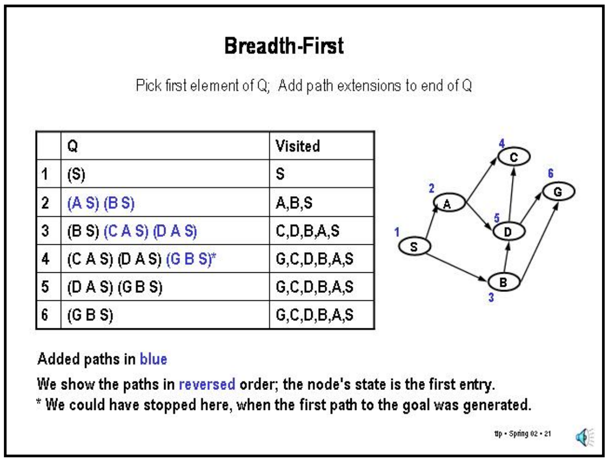
tlp • Spring 02 • 20

Slide 2.3.20

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

Slide 2.3.21

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



Breadth-First

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

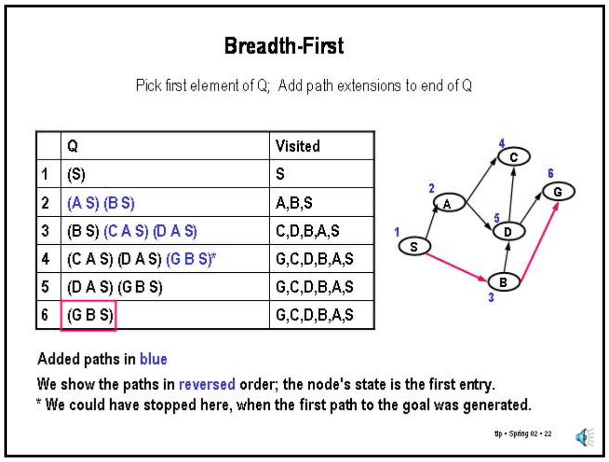
	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

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.

tlp • Spring 02 • 21



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.

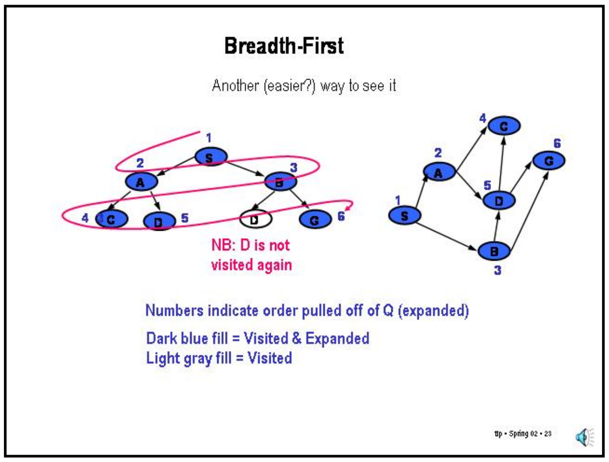
tlp • Spring 02 • 22

Slide 2.3.22

Note that we found a path with fewer states than we did with depth-first search, from S to B to G. In general, breadth-first search guarantees finding a path to the goal with the minimum number of states.

Slide 2.3.23

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

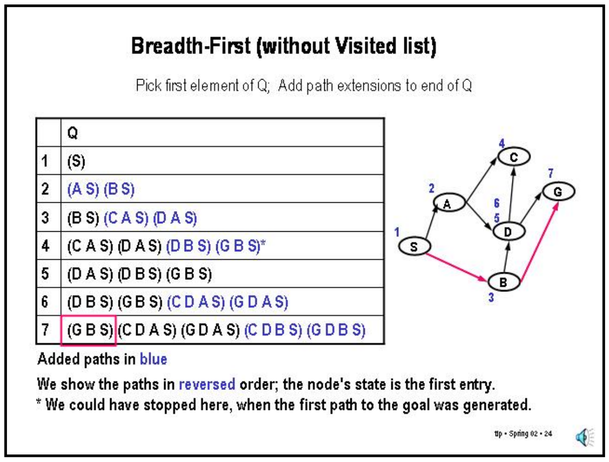
Another (easier?) way to see it

Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

tlp • 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)

1 S
2 A
3 B
4 C
5 6 D
7 G

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.

tlp • Spring 02 • 24

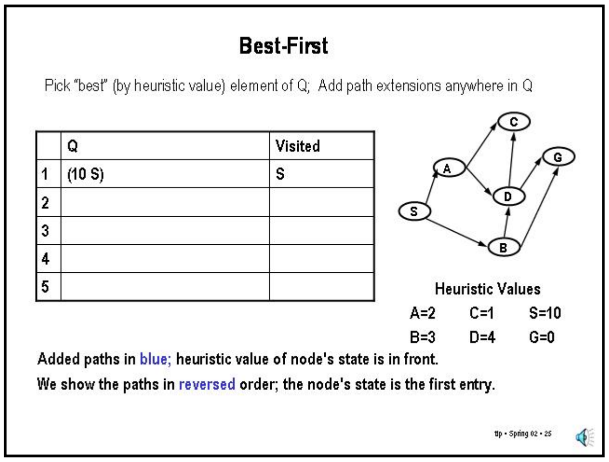
Slide 2.3.24

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 2.3.25

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		

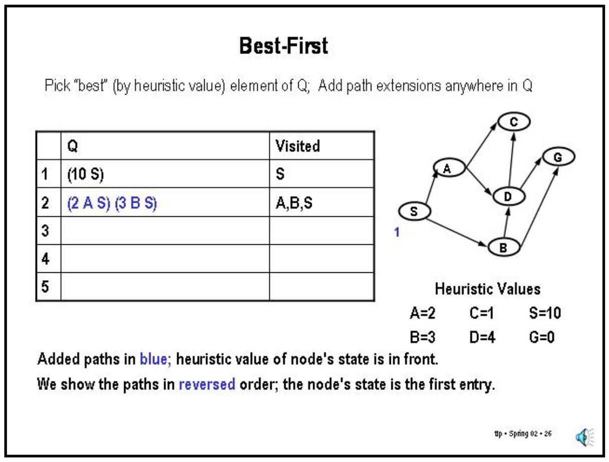
Heuristic Values

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.

tlp • 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.

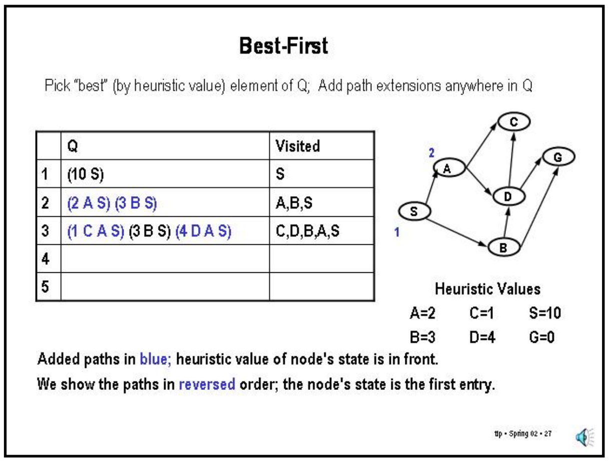
tlp • Spring 02 • 26

Slide 2.3.26

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

Slide 2.3.27

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



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		

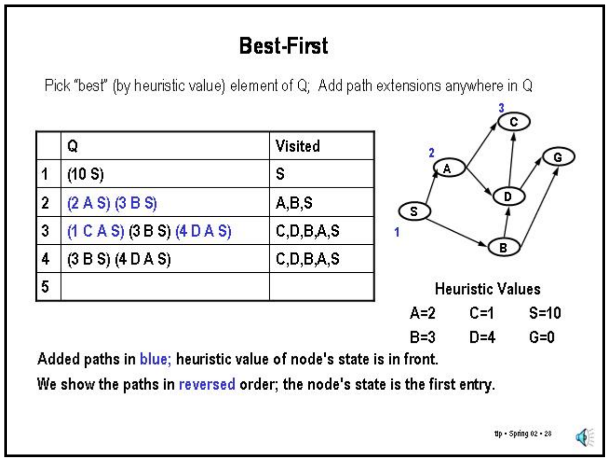
Heuristic Values

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.

tlp • Spring 02 • 27



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.

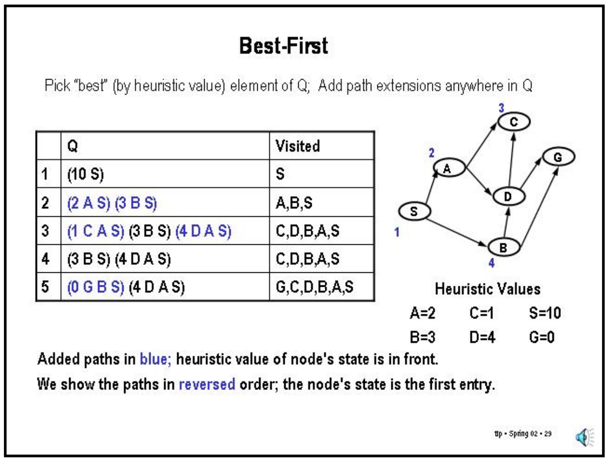
tlp • Spring 02 • 28

Slide 2.3.28

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

Slide 2.3.29

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



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

S
A
B
C
D
G
1
2
3
4

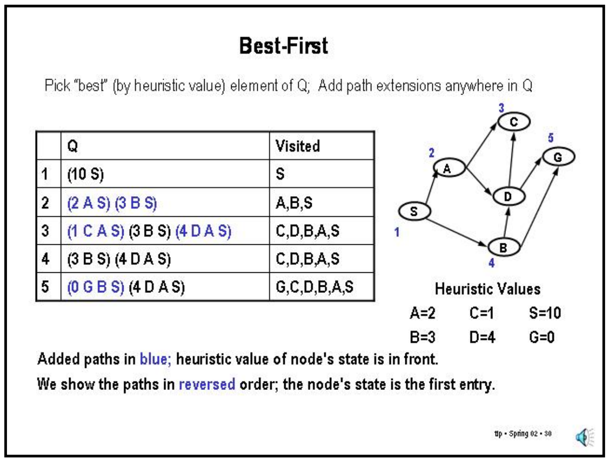
Heuristic Values

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.

tlp • Spring 02 • 29



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

S
A
B
C
D
G
1
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.

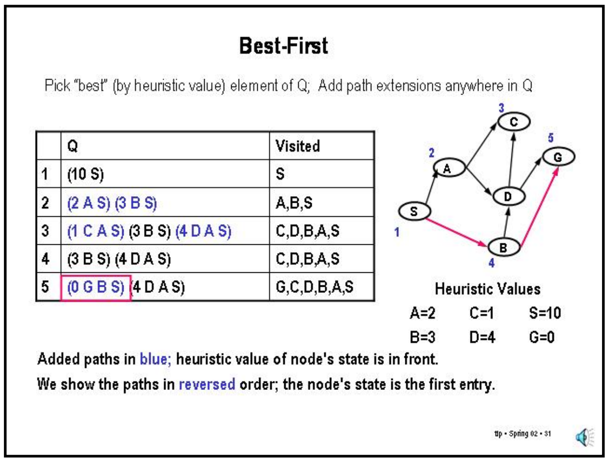
tlp • Spring 02 • 30

Slide 2.3.30

We pick the node corresponding to G and rejoice.

Slide 2.3.31

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



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

S
A
B
C
D
G
1
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.