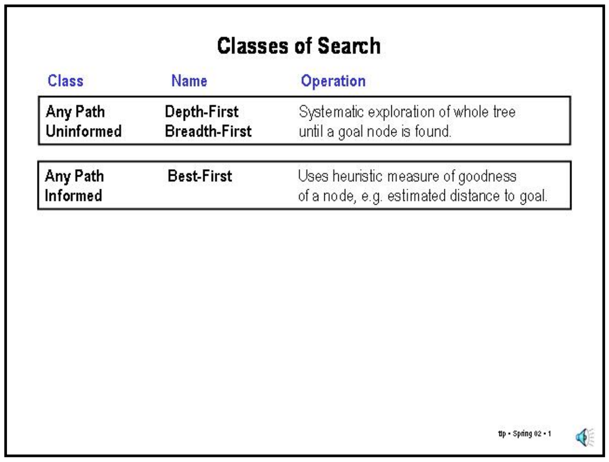
6.034 Notes: Section 2.4

Slide 2.4.1

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



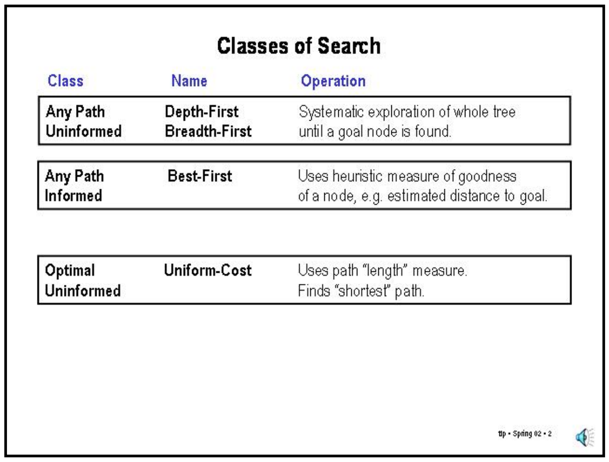
Classes of Search

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.

tlp • Spring 02 • 1

Slide 2.4.2

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



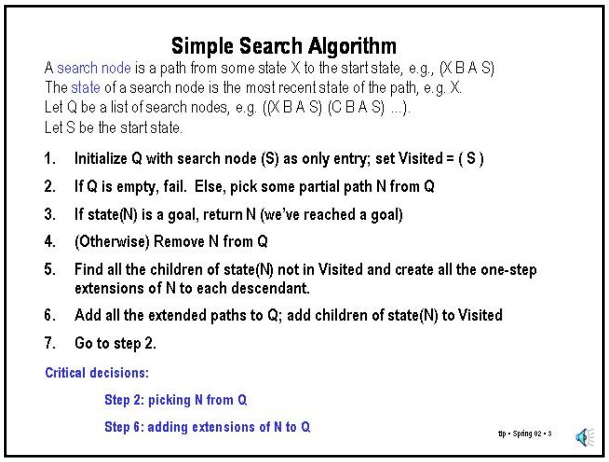
Classes of Search

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.

tlp • Spring 02 • 2

Slide 2.4.3

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



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.

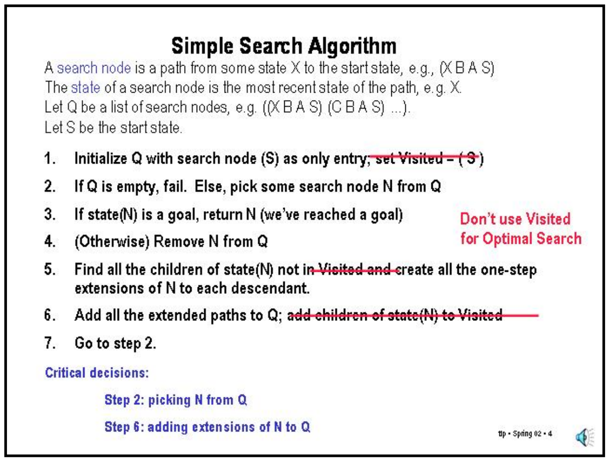
1. Initialize Q with search node (S) as only entry; set Visited = (S)
2. If Q is empty, fail. Else, pick some partial path N from Q
3. If state(N) is a goal, return N (we've reached a goal)
4. (Otherwise) Remove N from Q
5. Find all the children of state(N) not in Visited and create all the one-step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

tlp • 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)
4. (Otherwise) Remove N from Q
5. Find all the children of state(N) not i~~n 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.

Don't use Visited for Optimal Search

Critical decisions:

Step 2: picking N from Q

Step 6: adding extensions of N to Q

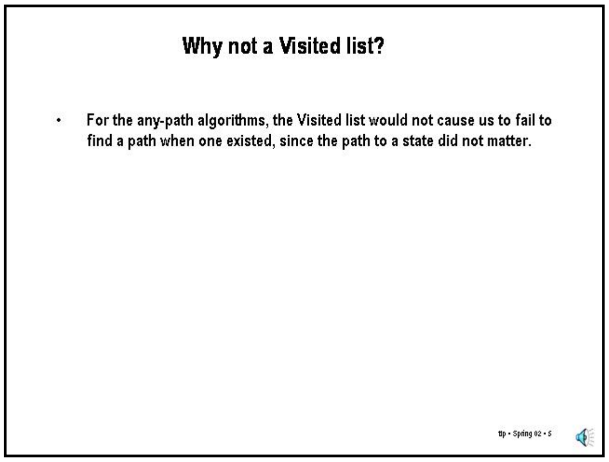
tlp • Spring 02 • 4

Slide 2.4.4

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

Slide 2.4.5

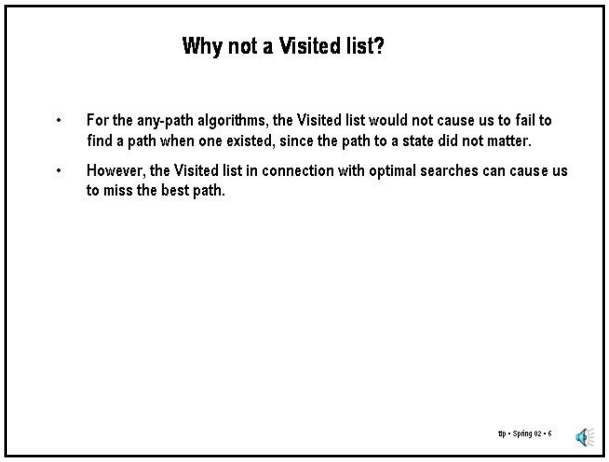
Why can't we use a Visited list in connection with optimal searching? In the earlier searches, the use of the Visited list guaranteed that we would not do extra work by re-visiting or re-expanding states. It did not cause any failures then (except possibly of intuition).



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.

tlp • 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.

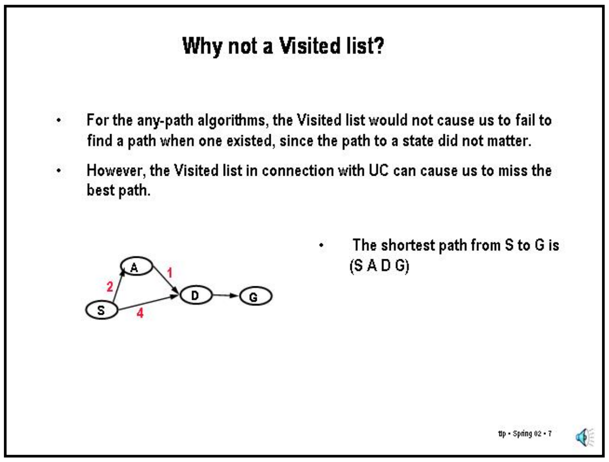
tlp • Spring 02 • 6

Slide 2.4.6

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

Slide 2.4.7

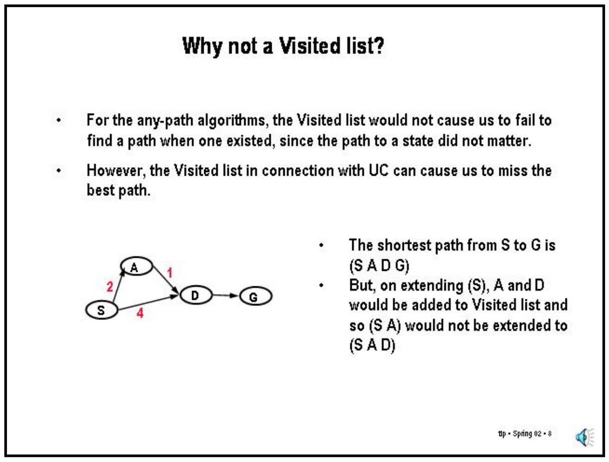
Clearly, the shortest path (as determined by sum of link costs) to G is (S A D G) and an optimal search had better find it.



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)

A
1
2
D
G
S
4
tlp • Spring 02 • 7



Why not a Visited list?

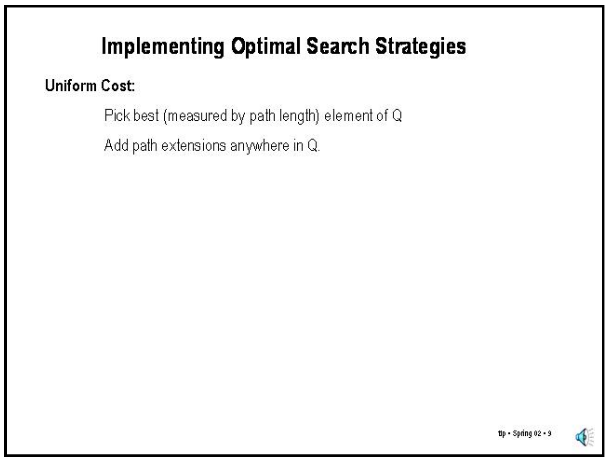
- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with UC can cause us to miss the best path.
- The shortest path from S to G is (S A D G)
- But, on extending (S), A and D would be added to Visited list and so (S A) would not be extended to (S A D)

Slide 2.4.8

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

Slide 2.4.9

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



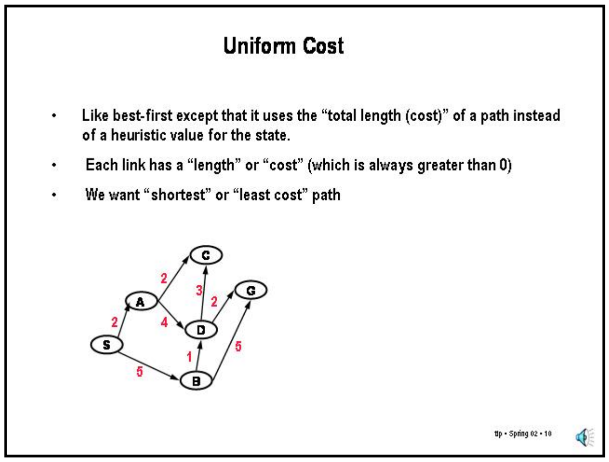
Implementing Optimal Search Strategies

Uniform Cost:

Pick best (measured by path length) element of Q

Add path extensions anywhere in Q.

tlp • 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

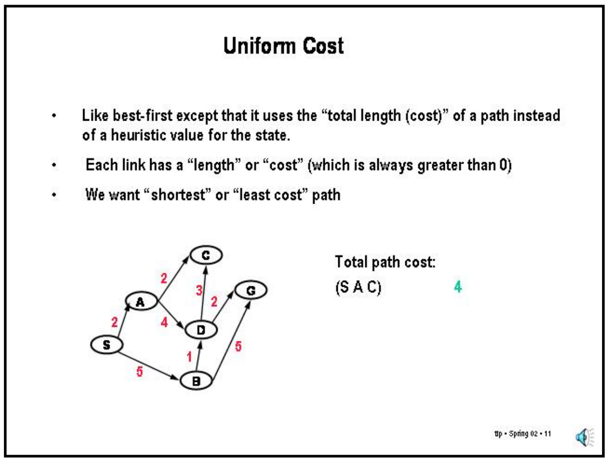
C
2
A
3
2
G
2
4
D
S
1
5
5
B
tlp • Spring 02 • 10

Slide 2.4.10

To reiterate, uniform-cost search uses the total length (or cost) of a path to decide which one to expand. Since we generally want the least-cost path, we will pick the node with the smallest path cost/length. By the way, we will often use the word "length" when talking about these types of searches, which makes intuitive sense when we talk about the pictures of graphs. However, we mean any cost measure (like length) that is positive and greater than 0 for the link between any two states.

Slide 2.4.11

The path length is the SUM of the length associated with the links in the path. For example, the path from S to A to C has total length 4, since it includes two links, each with edge 2.



Uniform Cost

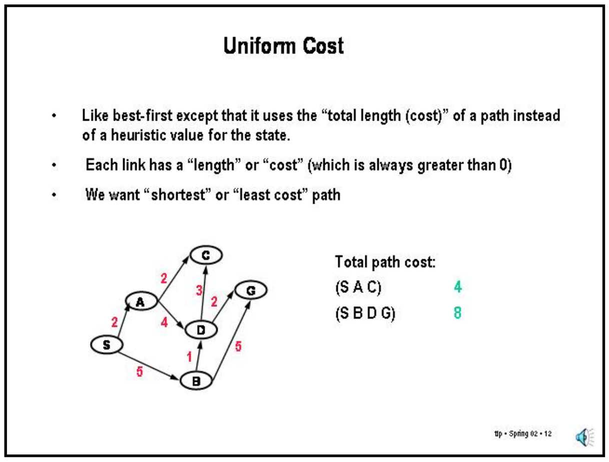
- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path

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

Total path cost:

(S A C)	4

tlp • 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

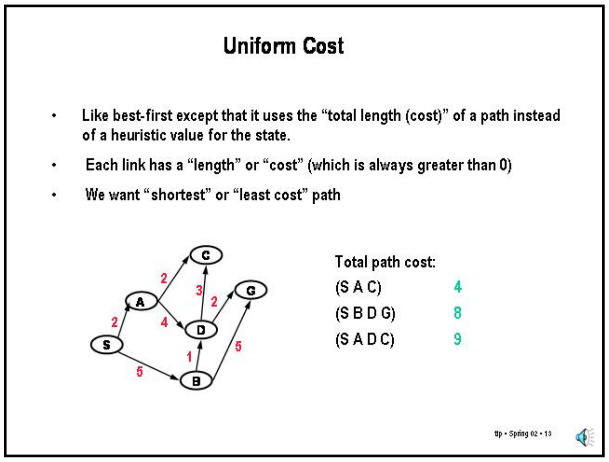
tlp • Spring 02 • 12

Slide 2.4.12

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

Slide 2.4.13

Similarly for S-A-D-C.



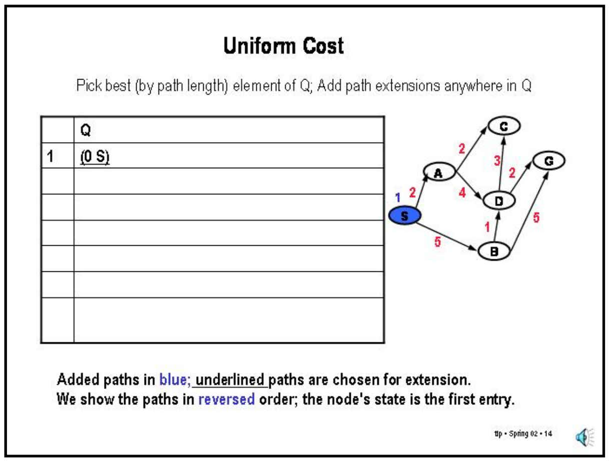
Uniform Cost

- 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

tlp • Spring 02 • 13



Uniform Cost

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

	Q
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.

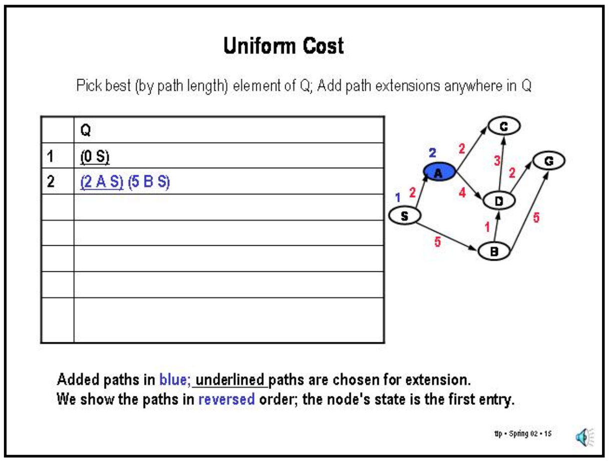
tlp • Spring 02 • 14

Slide 2.4.14

Given this, let's simulate the behavior of uniform-cost search on this simple directed graph. As usual we start with a single node containing just the start state S. This path has zero length. Of course, we choose this path for expansion.

Slide 2.4.15

This generates two new entries on Q; the path to A has length 2 and the one to B has length 5. So, we pick the path to A to expand.



Uniform Cost

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

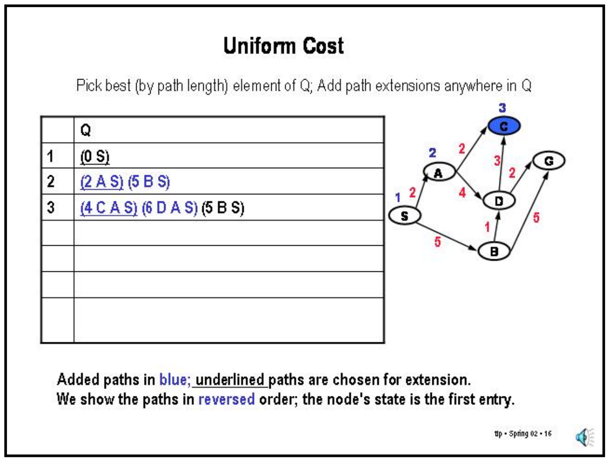
	Q
1	(0 S)
2	(2 A S) (5 B S)

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

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.

Slide 2.4.16

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

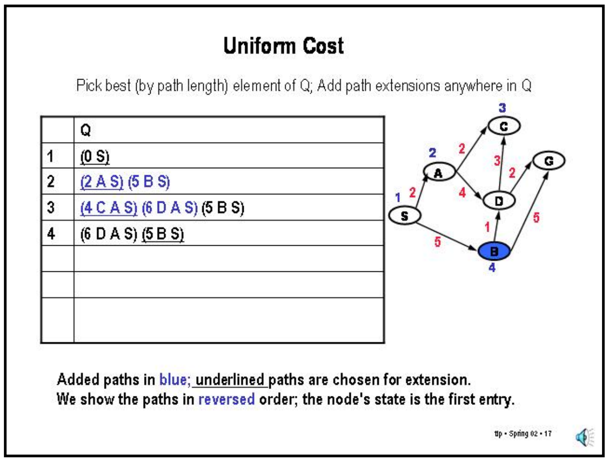
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)

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

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.

Slide 2.4.17

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



Uniform Cost

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

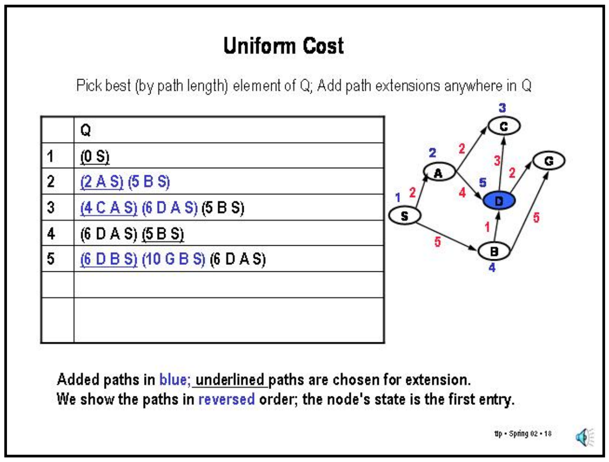
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)

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

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.

Slide 2.4.18

The path to B is extended to D and G and the path to D from B is tied with the path to D from A. We are using order in Q to settle ties and so we pick the path from B to expand. Note that at this point G has been visited but not expanded.



Uniform Cost

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

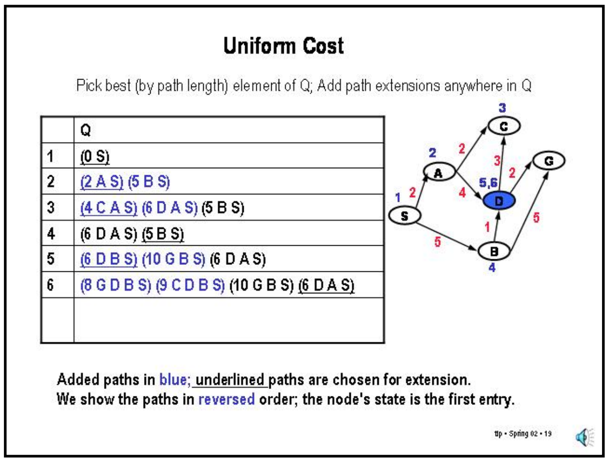
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)

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

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.

Slide 2.4.19

Expanding D adds paths to C and G. Now the earlier path to D from A is the best pending path and we choose it to expand.



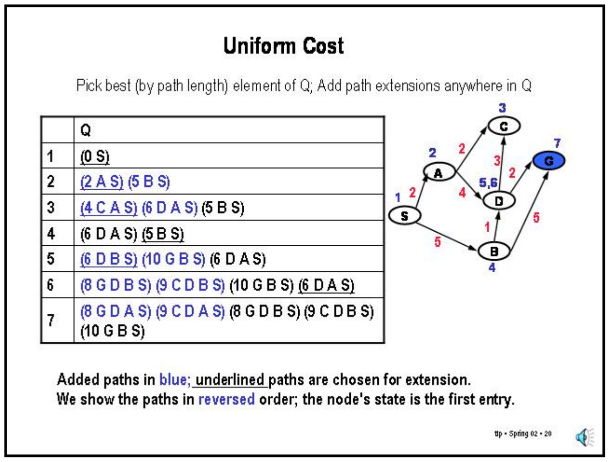
Uniform Cost

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.

tlp • Spring 02 • 19



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.

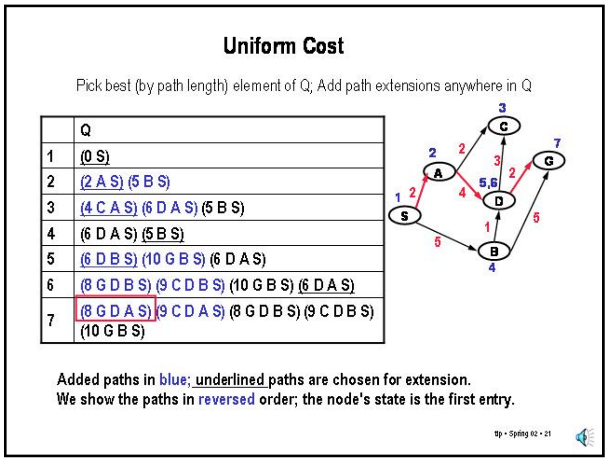
tlp • Spring 02 • 20

Slide 2.4.20

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

Slide 2.4.21

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



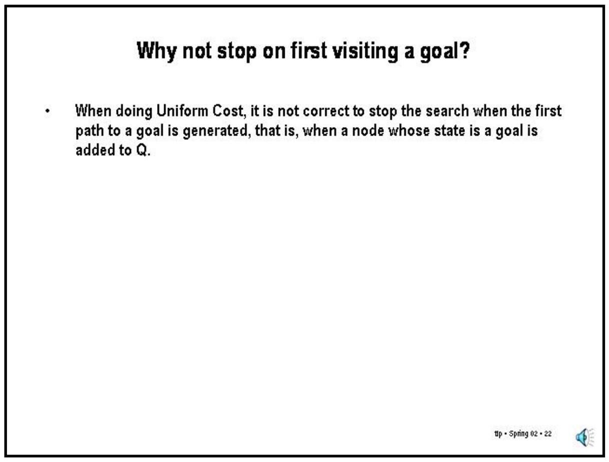
Uniform Cost

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	$\boxed{\text{(8 G D A S)}}$ (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.

tlp • Spring 02 • 21



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.

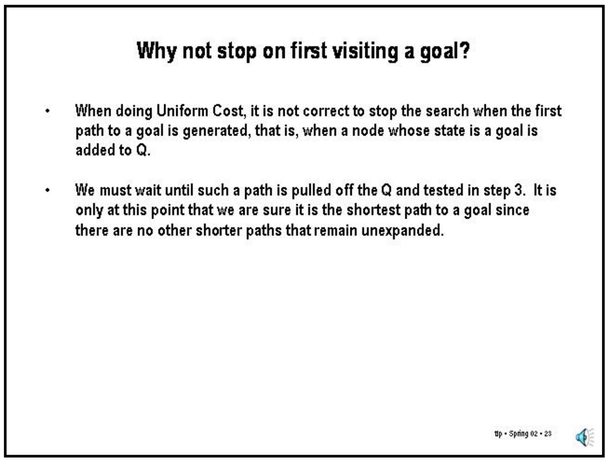
tlp • Spring 02 • 22

Slide 2.4.22

Note that once again we are not stopping on first visiting (placing on Q) the goal. We stop when the goal gets expanded (pulled off Q).

Slide 2.4.23

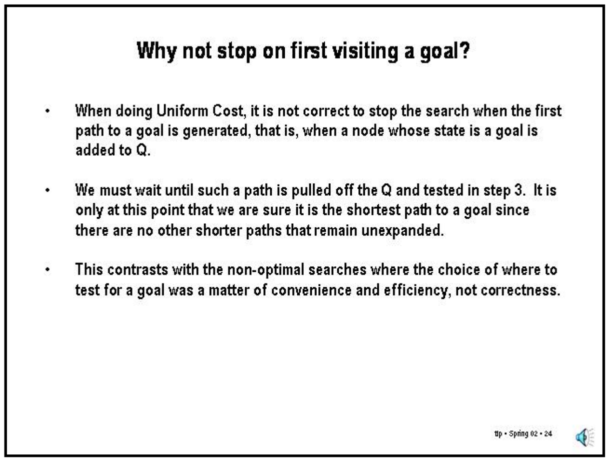
In uniform-cost search, it is imperative that we only stop when G is expanded and not just when it is visited. Until a path is first expanded, we do not know for a fact that we have found the shortest path to the state.



Why not stop on first visiting a goal?

- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.

tlp • 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.

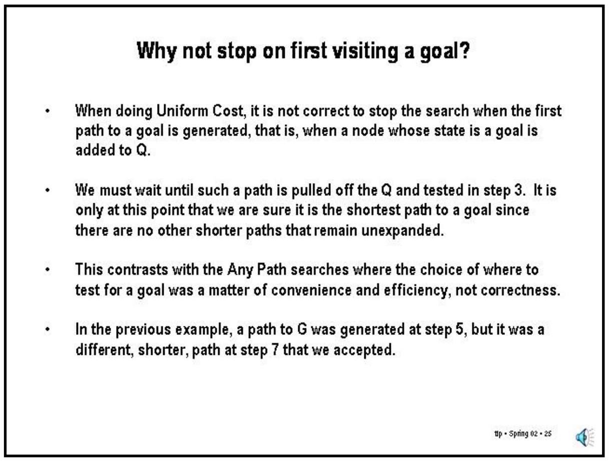
tlp • Spring 02 • 24

Slide 2.4.24

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

Slide 2.4.25

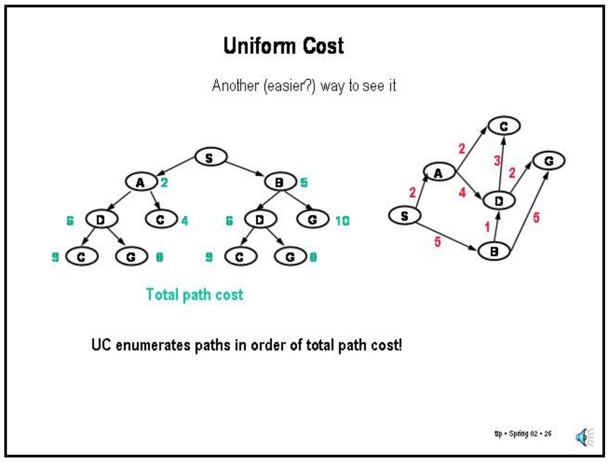
Note that the first path that visited G was not the eventually chosen optimal path to G. This explains our unwillingness to stop on first visiting G in the example we just did.



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.

tlp • Spring 02 • 25



Uniform Cost

Another (easier?) way to see it

UC enumerates paths in order of total path cost!

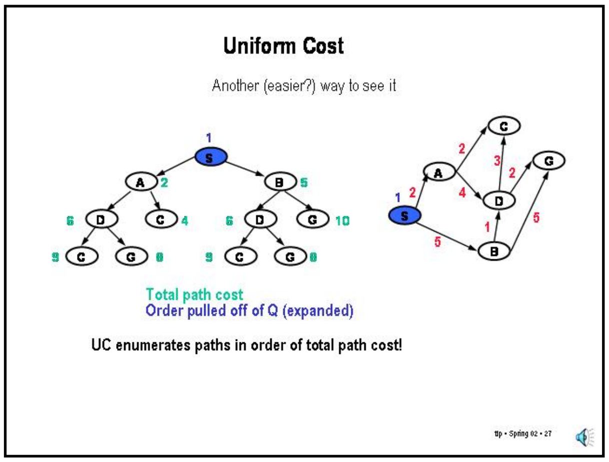
tlp • Spring 02 • 26

Slide 2.4.26

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

Slide 2.4.27

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



Uniform Cost

Another (easier?) way to see it

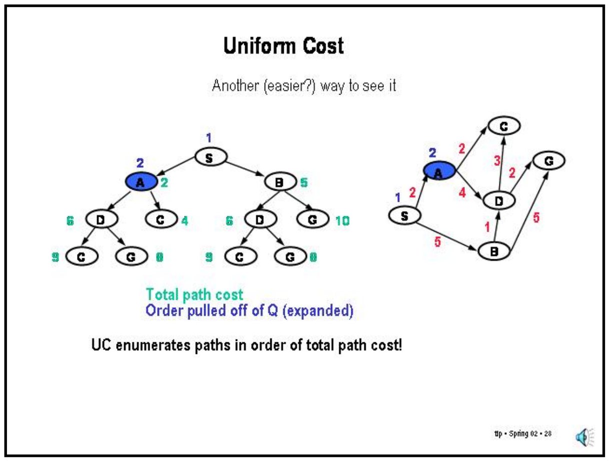
Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 27

Slide 2.4.28

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



Uniform Cost

Another (easier?) way to see it

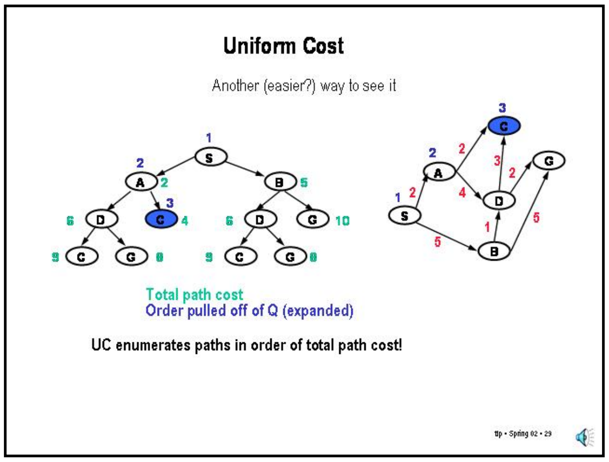
Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 28

Slide 2.4.29

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



Uniform Cost

Another (easier?) way to see it

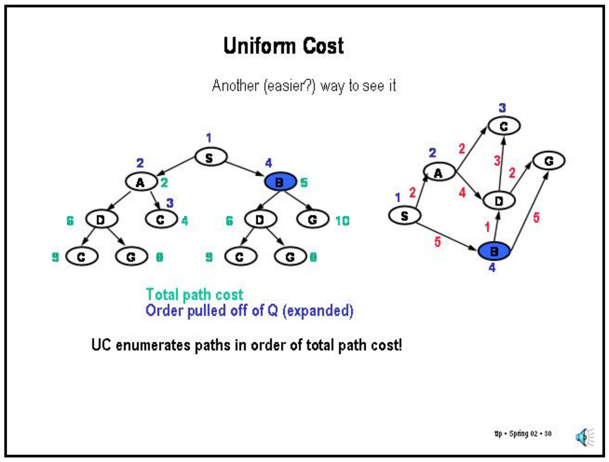
Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 29

Slide 2.4.30

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



Uniform Cost

Another (easier?) way to see it

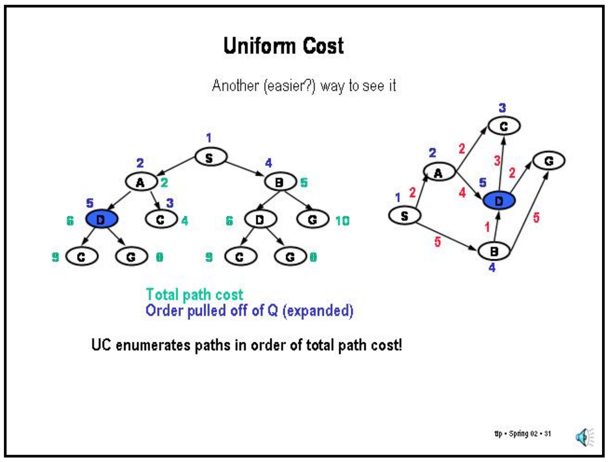
Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 30

Slide 2.4.31

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



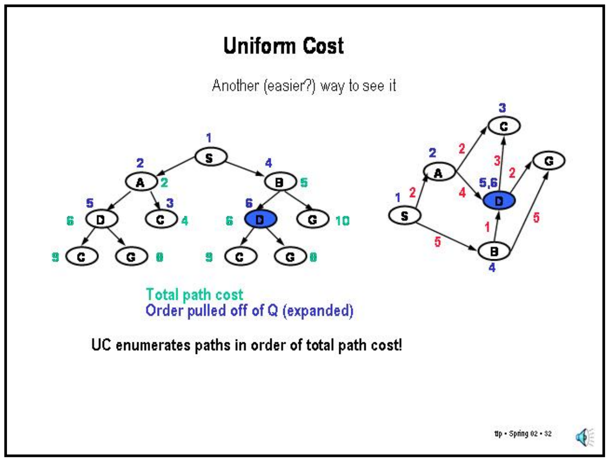
Uniform Cost

Another (easier?) way to see it

Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 31



Uniform Cost

Another (easier?) way to see it

Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

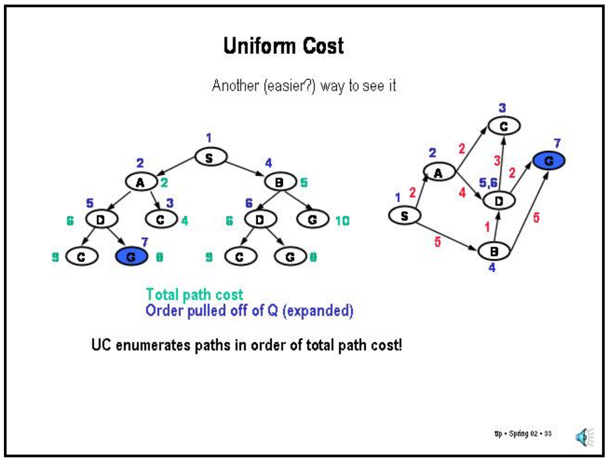
tlp • Spring 02 • 32

Slide 2.4.32

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

Slide 2.4.33

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



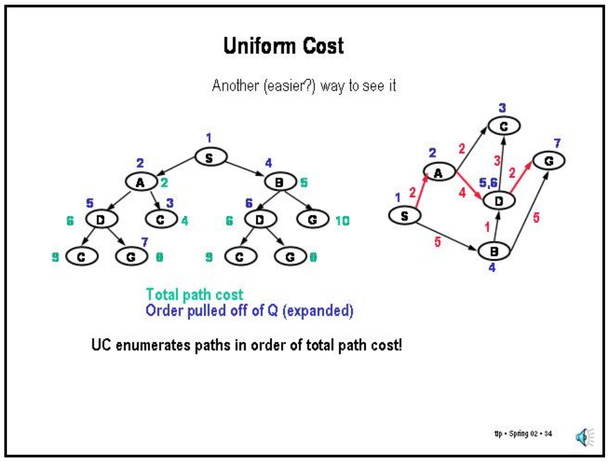
Uniform Cost

Another (easier?) way to see it

Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 33



Uniform Cost

Another (easier?) way to see it

Total path cost
Order pulled off of Q (expanded)

UC enumerates paths in order of total path cost!

tlp • Spring 02 • 34

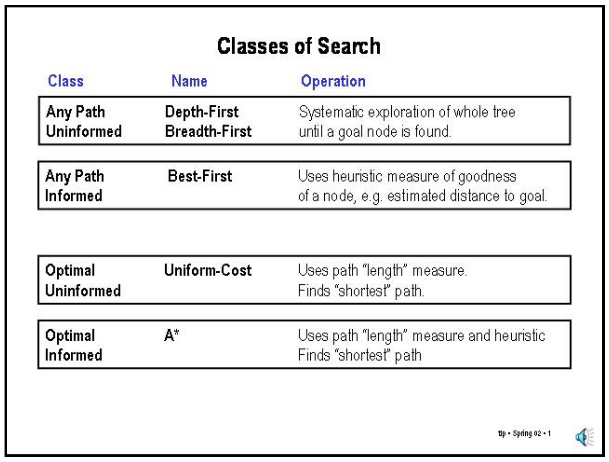
Slide 2.4.34

This gives us the path we found. Note that the sequence of expansion corresponds precisely to path-length order, so it is not surprising we find the shortest path.

6.034 Notes: Section 2.5

Slide 2.5.1

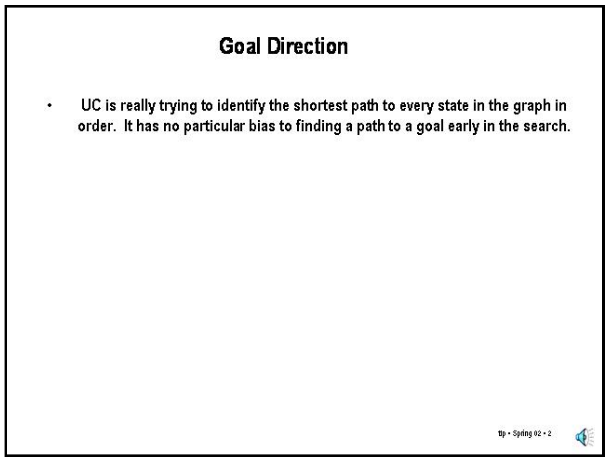
Now, we will turn our attention to what is probably the most popular search algorithm in AI, the A* algorithm. A* is an informed, optimal search algorithm. We will spend quite a bit of time going over A*; we will start by contrasting it with uniform-cost search.



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

tlp • 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.

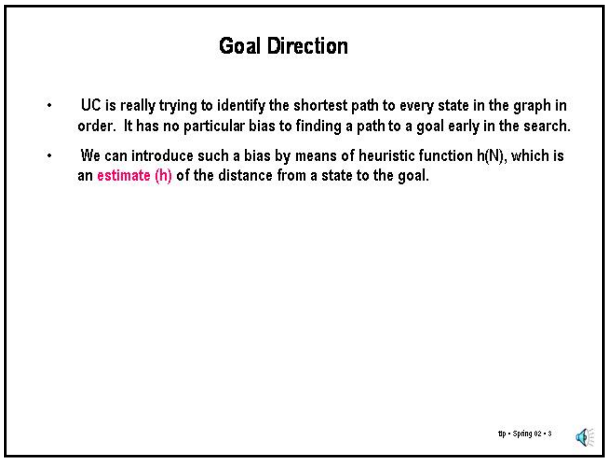
tlp • Spring 02 • 2

Slide 2.5.2

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

Slide 2.5.3

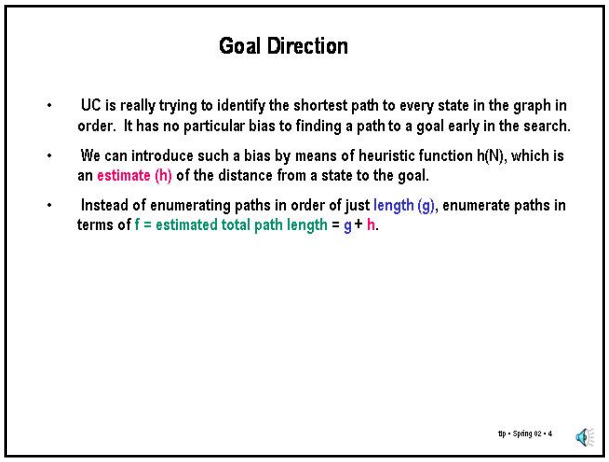
We can bias UC to find the shortest path to the goal that we are interested in by using a heuristic estimate of remaining distance to the goal. This, of course, cannot be the exact path distance (if we knew that we would not need much of a search); instead, it is a stand-in for the actual distance that can give us some guidance.



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.

tlp • 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 = estimated total path length = g + h.

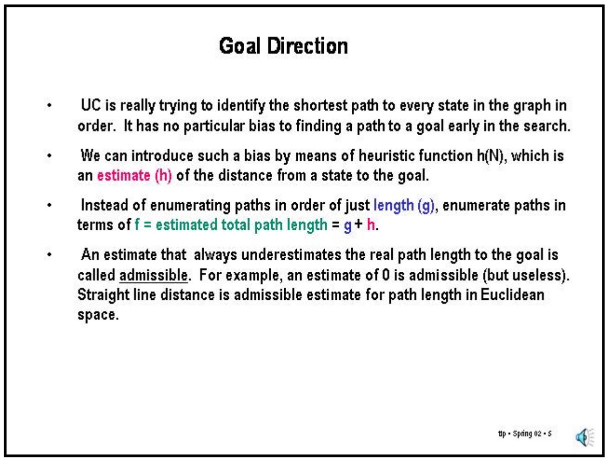
tlp • Spring 02 • 4

Slide 2.5.4

What we can do is to enumerate the paths by order of the SUM of the actual path length and the estimate of the remaining distance. Think of this as our best estimate of the TOTAL distance to the goal. This makes more sense if we want to generate a path to the goal preferentially to short paths away from the goal.

Slide 2.5.5

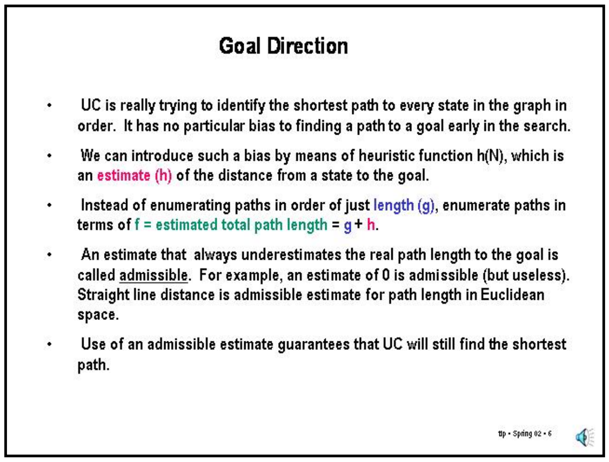
We call an estimate that always **under**estimates the remaining distance from any node an **admissible** (heuristic) estimate.



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 = 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.

tlp • 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 = 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.

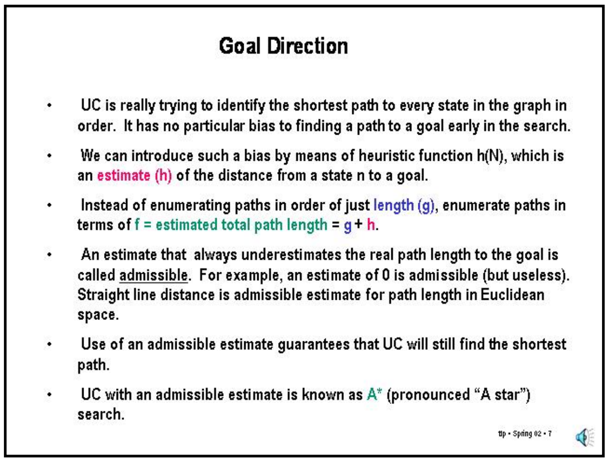
tlp • Spring 02 • 6

Slide 2.5.6

In order to preserve the guarantee that we will find the shortest path by expanding the partial paths based on the estimated **total** path length to the goal (like in UC without an expanded list), we must 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 2.5.7

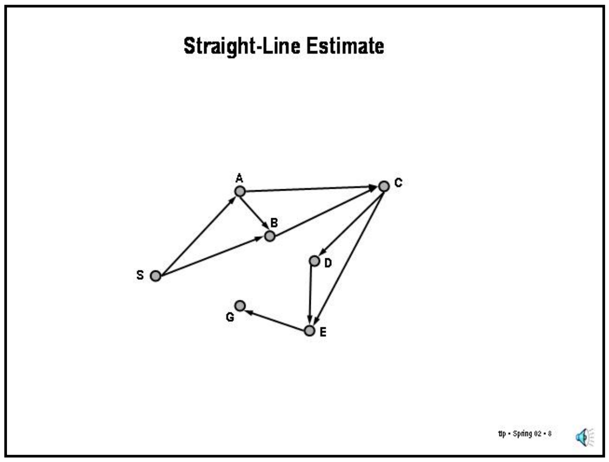
UC using an admissible heuristic is known as A* (A star). It is a very popular search method in AI.



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 = 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.

tlp • Spring 02 • 7

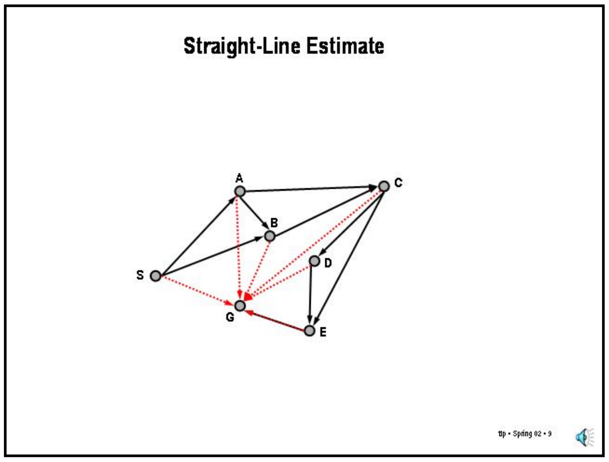


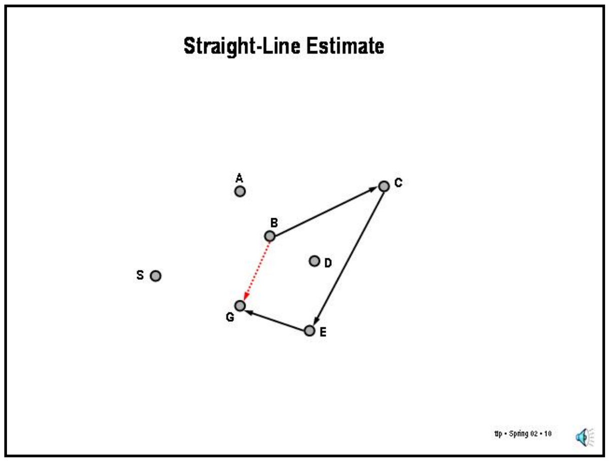
Slide 2.5.8

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 2.5.9

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.





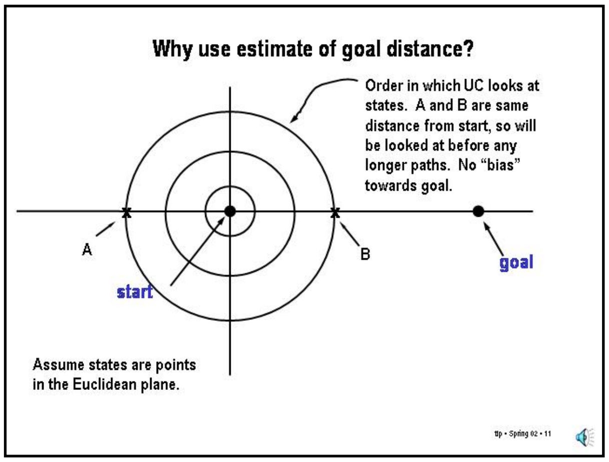
Slide 2.5.10

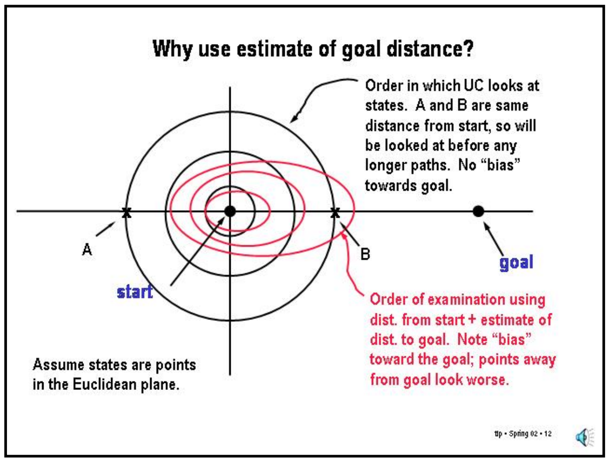
Here we see that the straight-line estimate between B and G is very bad. The actual driving distance is much longer than the straight-line underestimate. Imagine that B and G are on different sides of the Grand Canyon, for example.

Slide 2.5.11

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.
A
B
start
goal
Assume states are points in the Euclidean plane.
tlp • Spring 02 • 11



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.

tlp • Spring 02 • 12

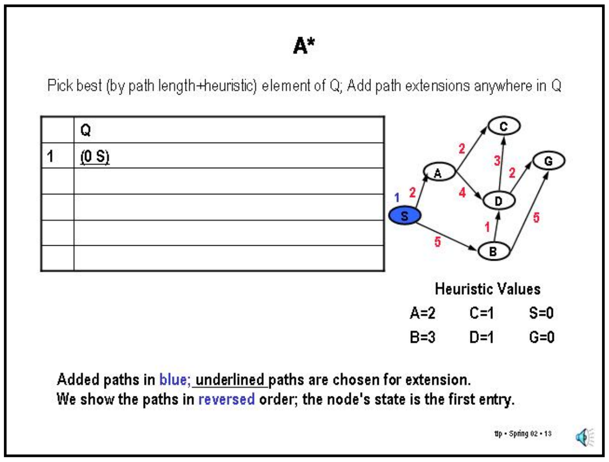
Slide 2.5.12

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

Slide 2.5.13

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

We start at S as usual.



A*

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

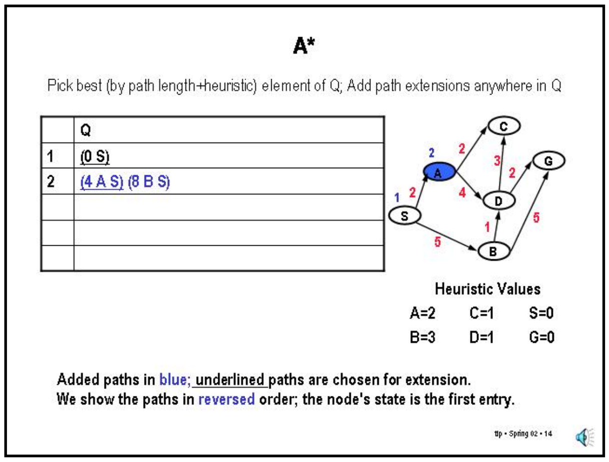
	Q
1	(0 S)

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.

tlp • 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.

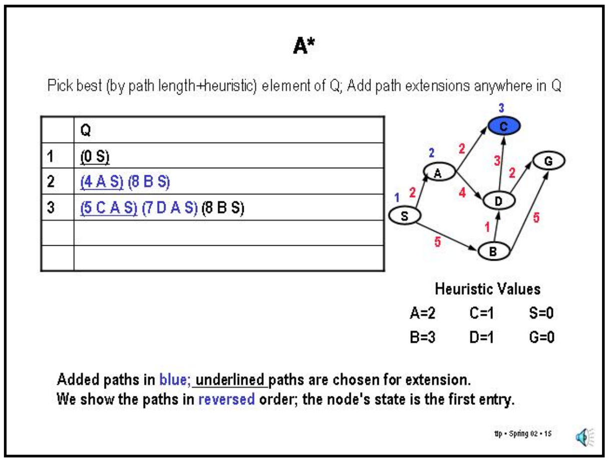
tlp • Spring 02 • 14

Slide 2.5.14

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

Slide 2.5.15

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



A*

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

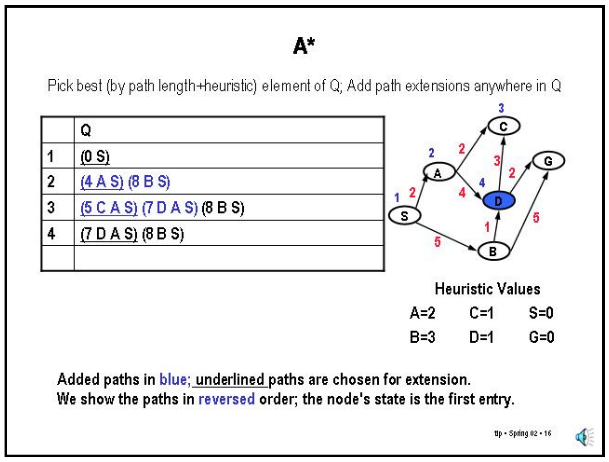
	Q
1	(0 S)
2	(4 A S) (8 B S)
3	(5 C A S) (7 D A S) (8 B S)

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.

tlp • 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

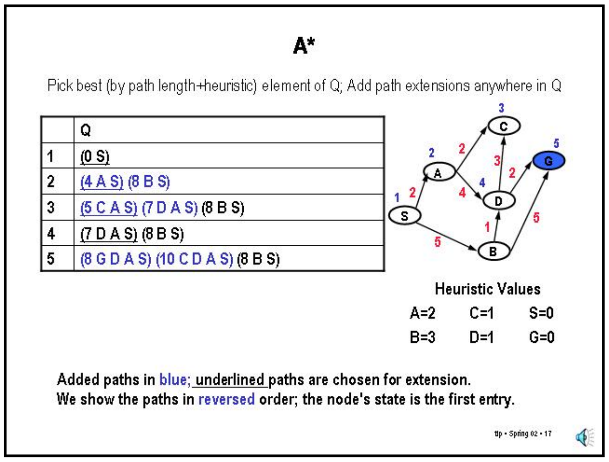
Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

Slide 2.5.16

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

Slide 2.5.17

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



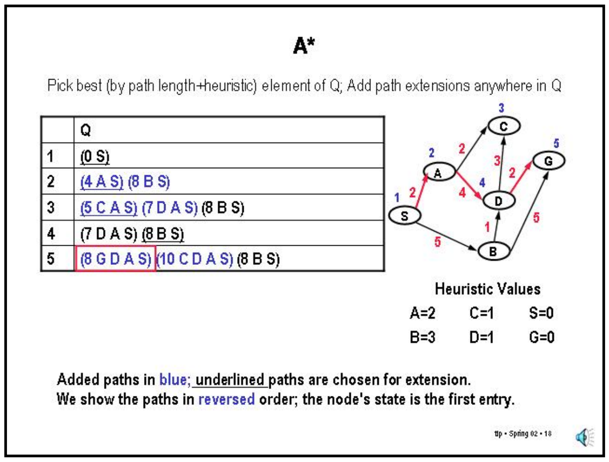
A*

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.



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

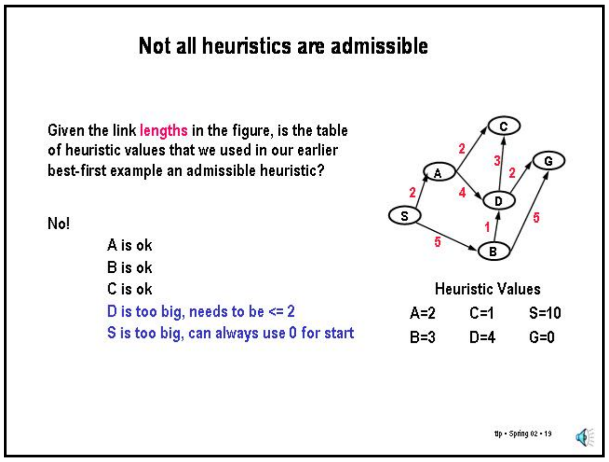
Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

Slide 2.5.18

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

Slide 2.5.19

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

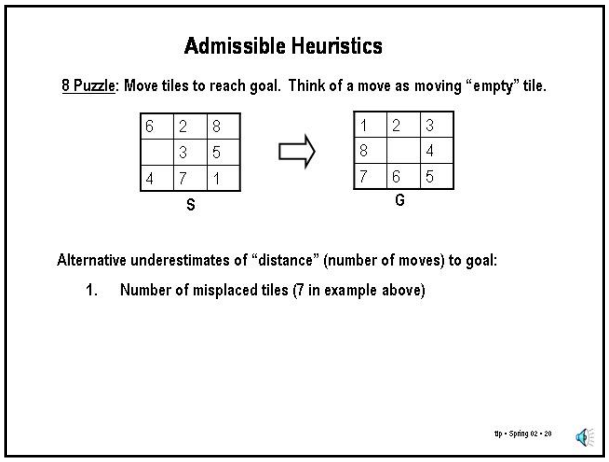


Not all heuristics are admissible

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



Admissible Heuristics

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

6	2	8
	3	5
4	7	1

S

1	2	3
8		4
7	6	5

G

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

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

tlp • Spring 02 • 20

Slide 2.5.20

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.

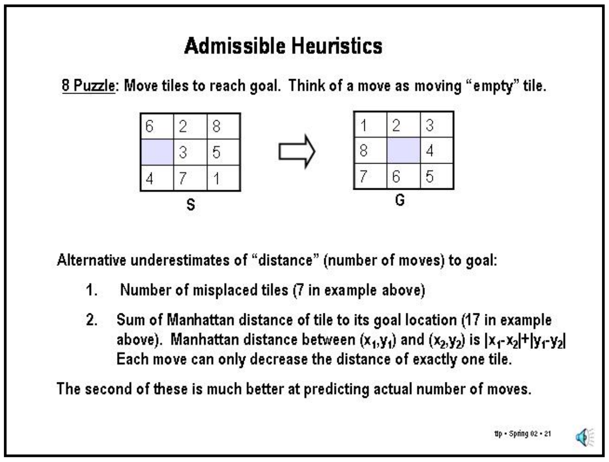
Take solving the so-called 8-puzzle, in which the goal is to arrange the pieces as in the goal state on the right. We can think of a move in this game as sliding the "empty" space to one of its nearest vertical or horizontal neighbors. We can help steer a search to find a short sequence of moves by using 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 2.5.21

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.

6	2	8
	3	5
4	7	1

S

1	2	3
8		4
7	6	5

G

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.

tlp • Spring 02 • 21