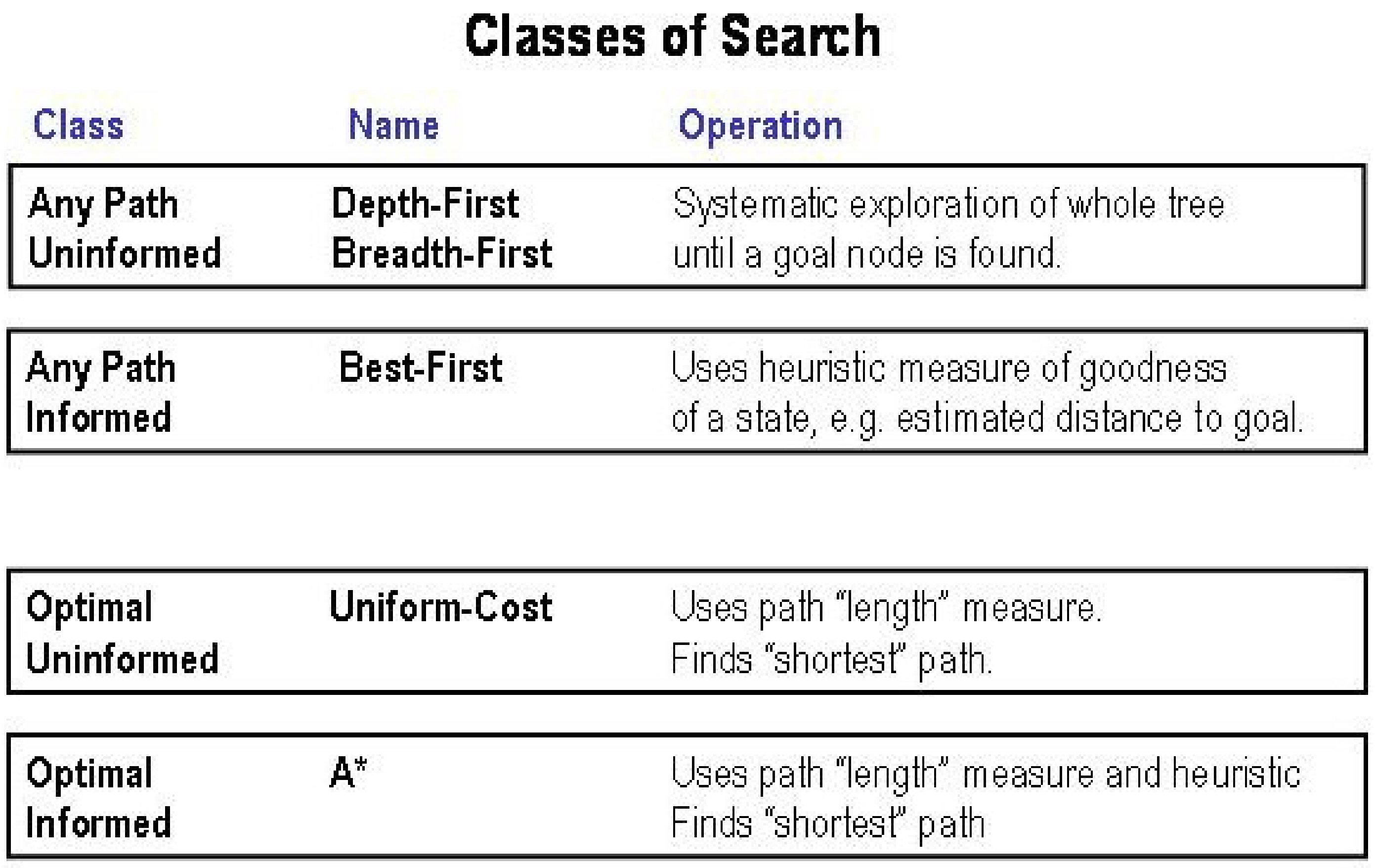


Classes of Search

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

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.**

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

Implementing the Search Strategies

Depth-first:

Pick first element of Q

Add path extensions to front of Q

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

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.

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.

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.

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.

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.

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).

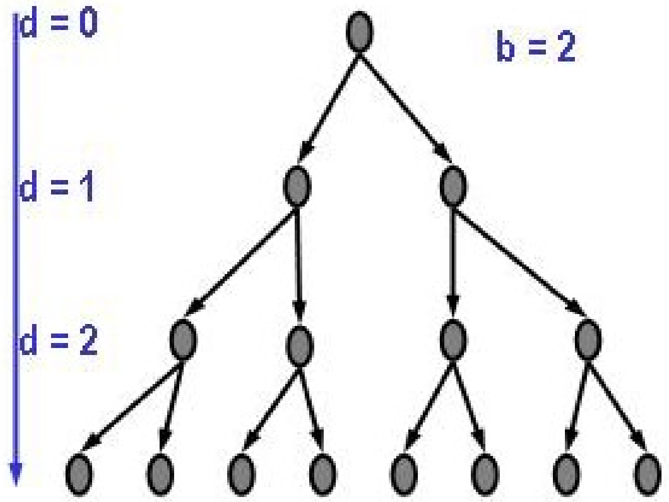
Implementation Issues: Finding the best node

- 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.

Worst Case Running Time

Max Time $\propto$ Max #Visited

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



d is depth
b is branching factor

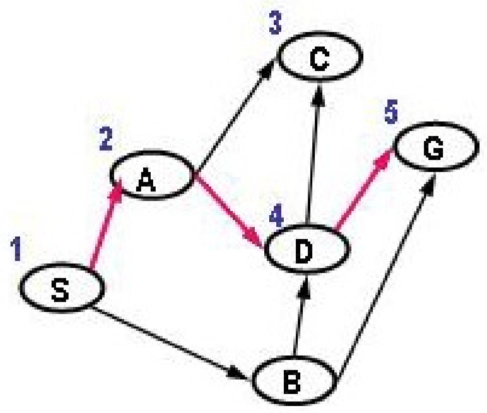
$b^d < (b^{d+1} - 1) / (b - 1) < b^{d+1}$

states in tree

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	$\boxed{\text{(G D A S)}}$ (B S)	G,C,D,B,A,S

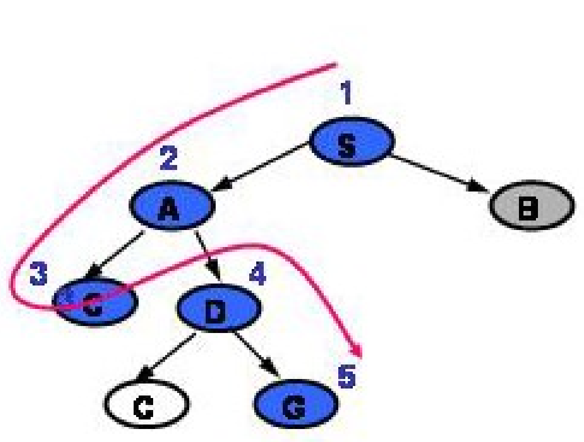


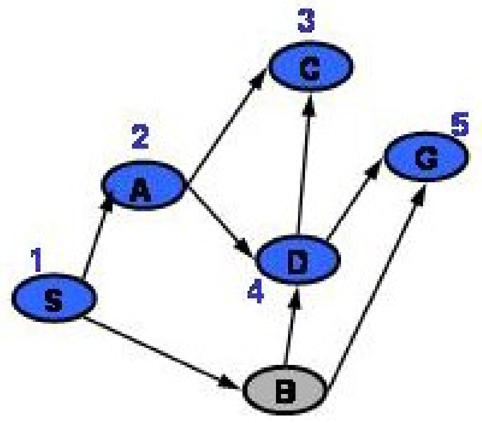
Added paths in blue

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

Depth-First

Another (easier?) way to see it





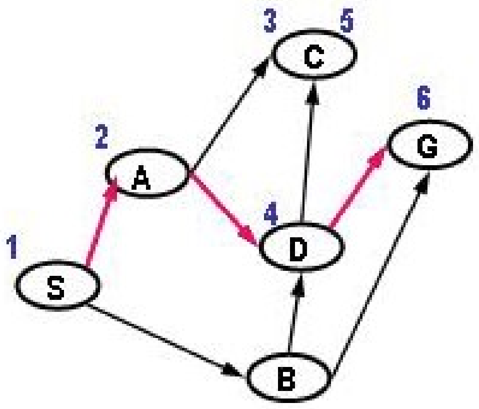
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded
Light gray fill = Visited

Depth-First (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	$\boxed{\text{(G D A S)}}$ (B S)

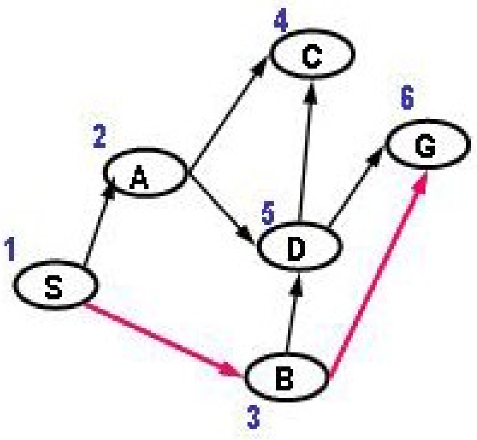


Added paths in blue
We show the paths in reversed order; the node's state is the first entry.
Do not extend a path to a state if the resulting path would have a loop.

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



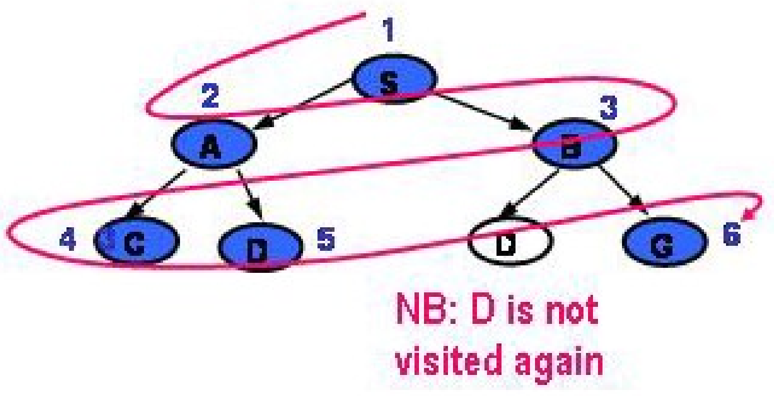
Added paths in blue

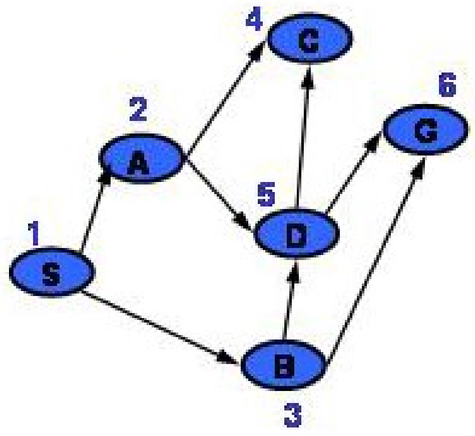
We show the paths in reversed order; the node's state is the first entry.

* We could have stopped here, when the first path to the goal was generated.

Breadth-First

Another (easier?) way to see it





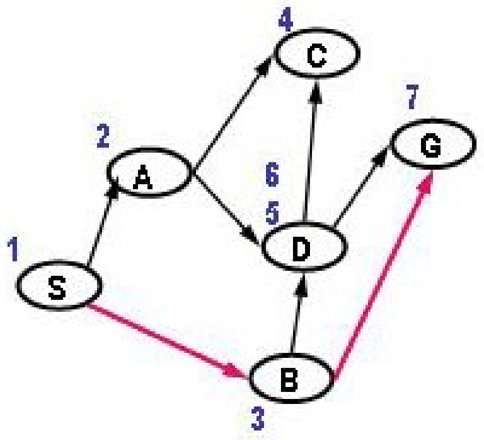
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded
Light gray fill = Visited

Breadth-First (without Visited list)

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

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



Added paths in blue

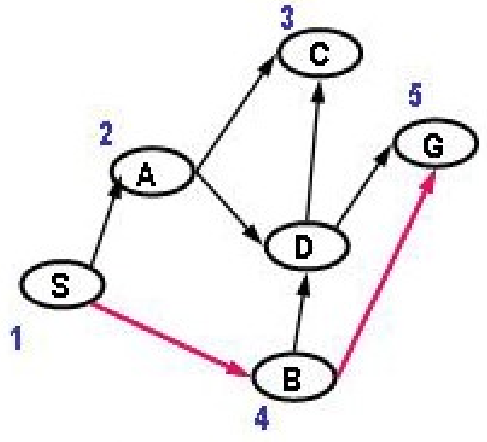
We show the paths in reversed order; the node's state is the first entry.

* We could have stopped here, when the first path to the goal was generated.

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	$\boxed{\text{(0 G B S)}}$ (4 D A S)	G,C,D,B,A,S

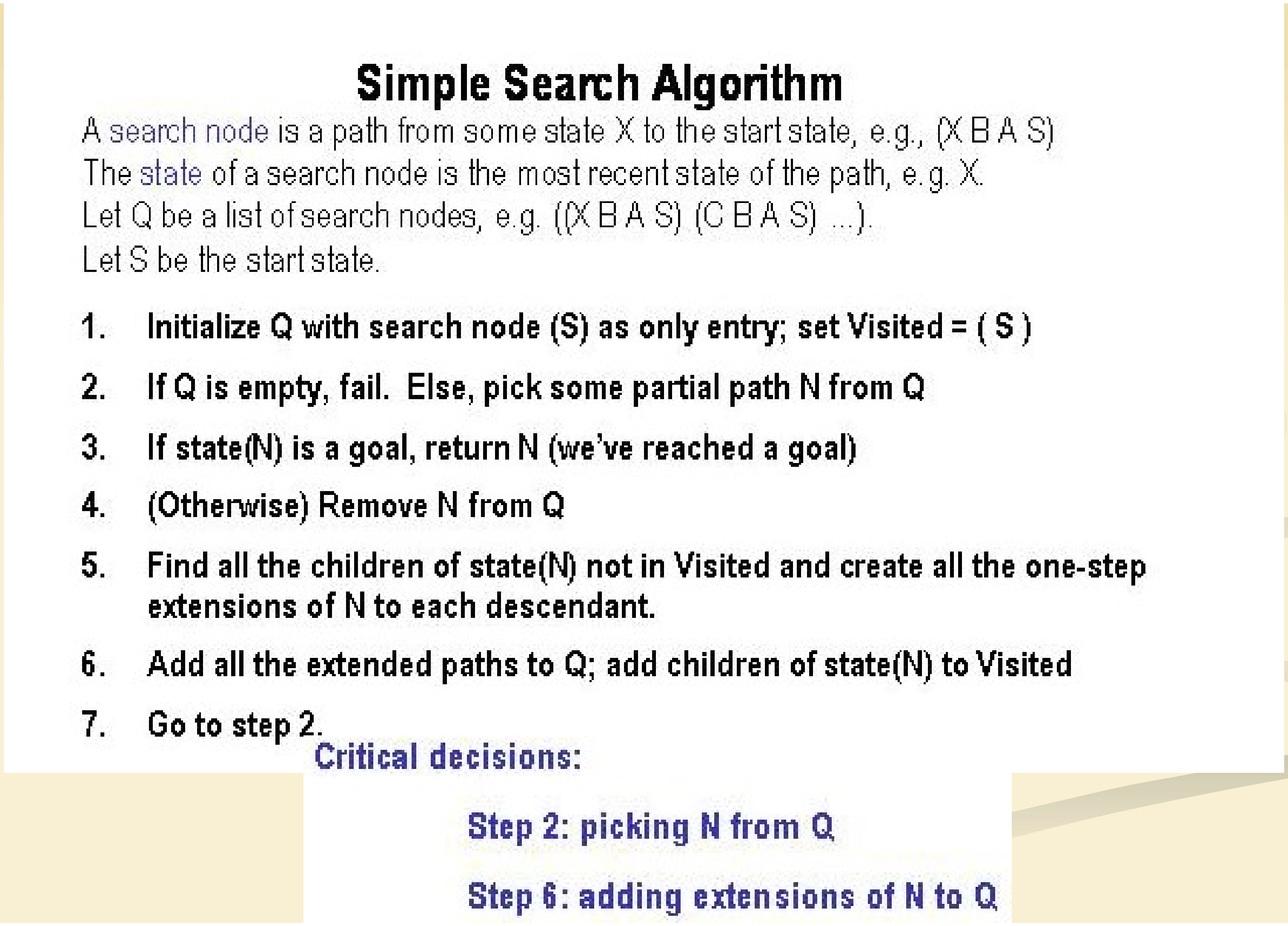


Heuristic Values

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

Added paths in blue; heuristic value of node's state is in front.

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



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

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 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.

Don't use Visited for Optimal Search

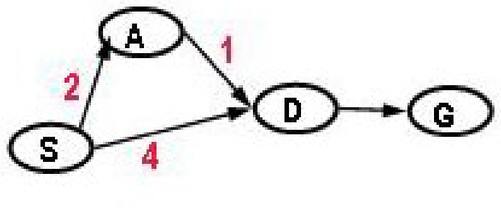
Critical decisions:

Step 2: picking N from Q

Step 6: adding extensions of N to Q

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)

Implementing Optimal Search Strategies

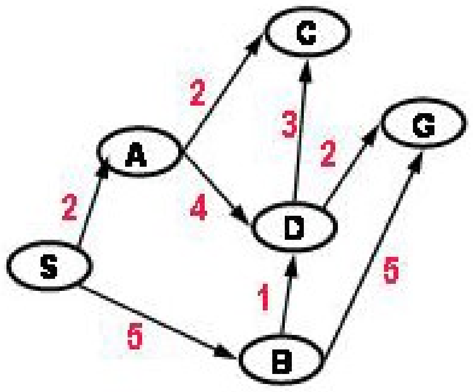
Uniform Cost:

Pick best (measured by path length) element of Q

Add path extensions anywhere in Q.

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



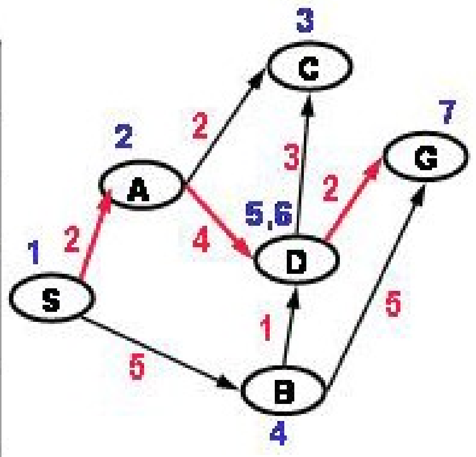
Total path cost:

(S A C)	4
(S B D G)	8
(S A D C)	9

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)



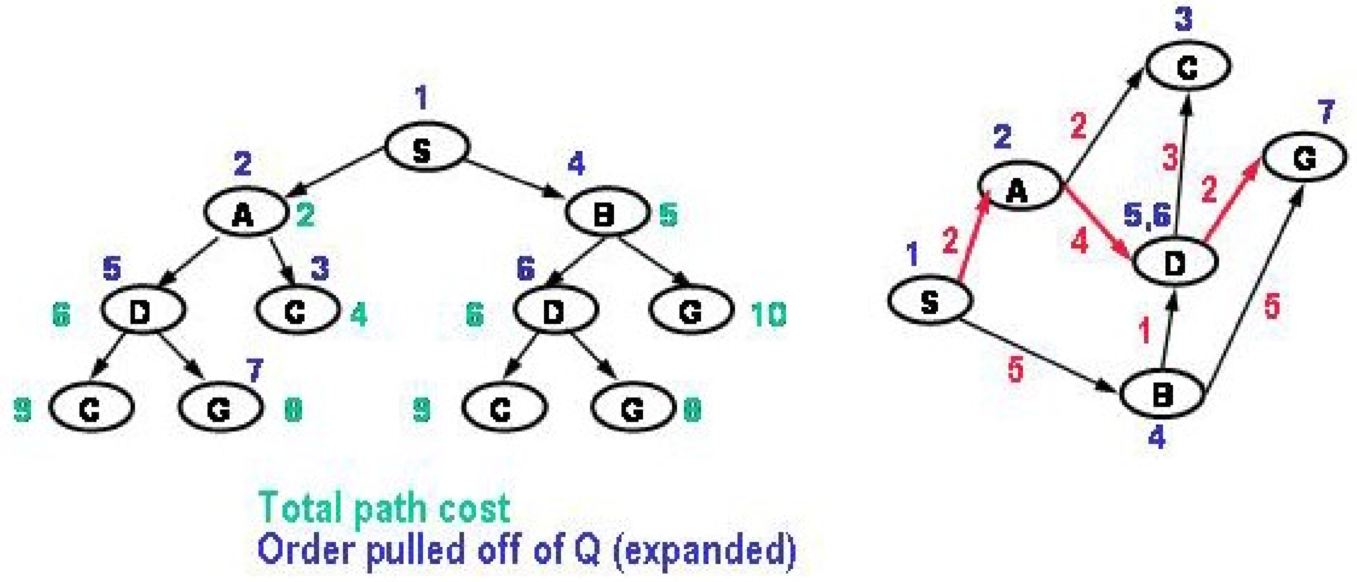
Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

Why not stop on first visiting a goal?

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

Uniform Cost

Another (easier?) way to see it



UC enumerates paths in order of total path cost!

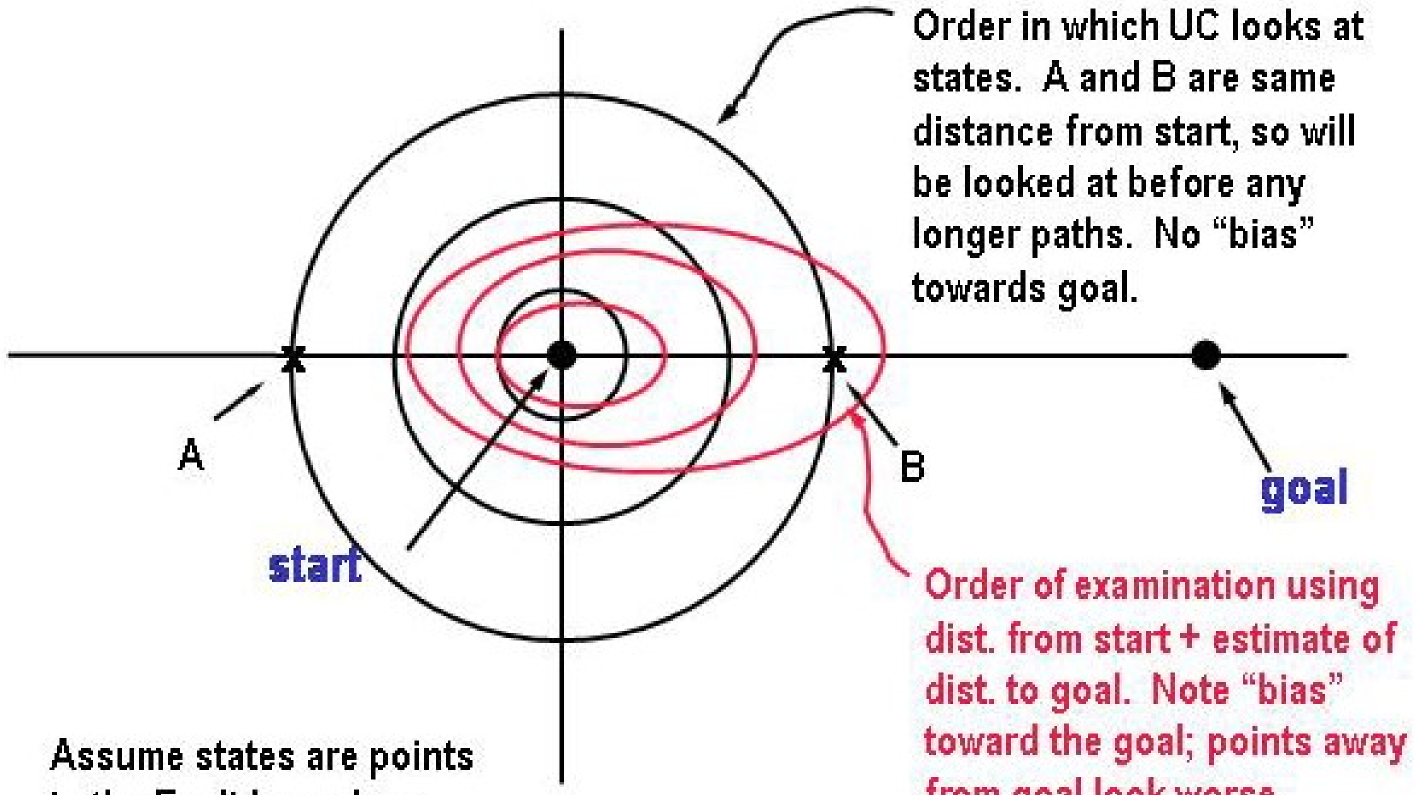
Classes of Search

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

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 <u>admissible</u>. 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.

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.

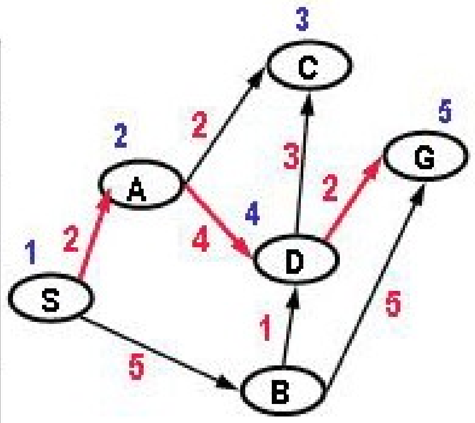
Order of examination using dist. from start + estimate of dist. to goal. Note "bias" toward the goal; points away from goal look worse.

Assume states are points in the Euclidean plane.

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.

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!

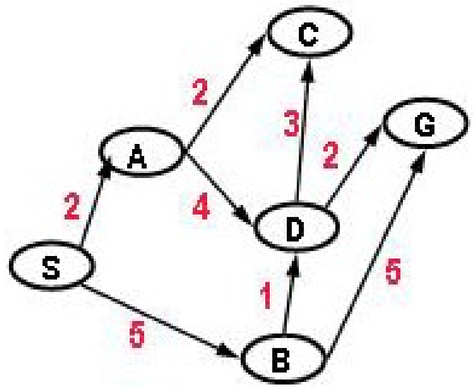
A is ok

B is ok

C is ok

D is too big, needs to be <= 2

S is too big, can always use 0 for start

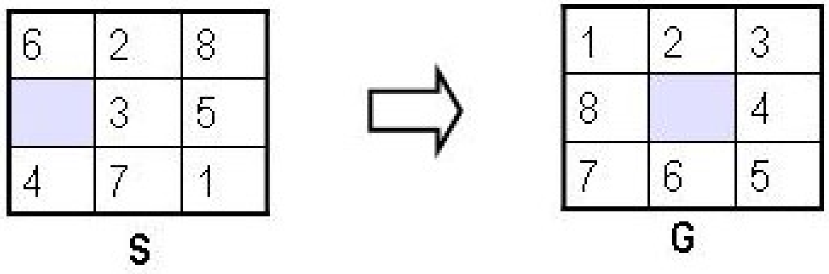


Heuristic Values

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

Admissible Heuristics

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



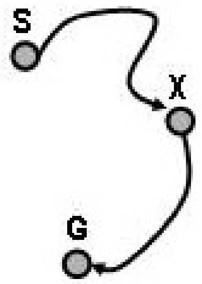
Alternative underestimates of "distance" (number of moves) to goal:

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

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

Dynamic Programming Optimality Principle
and the Expanded list

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



Dynamic Programming Optimality Principle

and the Expanded list

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

Simple Optimal Search Algorithm

Uniform Cost + Strict Expanded List

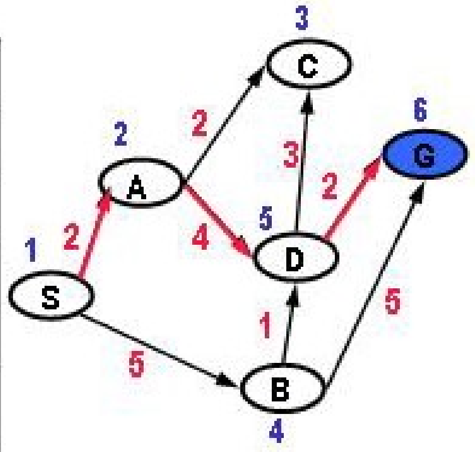
A search node is a path from some state X to the start state, e.g., (X B A S)
The state of a search node is the most recent state of the path, e.g. X.
Let Q be a list of search nodes, e.g. ((X B A S) (C B A S) ...).
Let S be the start state.

1. **Initialize Q with search node (S) as only entry; set Expanded = ( )**
2. **If Q is empty, fail. Else, pick least cost 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. **if state(N) in Expanded, go to step 2, otherwise add state(N) to Expanded.**
6. **Find all the children of state(N) (Not in Expanded) and create all the one-step extensions of N to each descendant.**
7. **Add all the extended paths to Q; if descendant state already in Q, keep only shorter path to the state in Q.**
8. **Go to step 2.**

Uniform Cost (with strict expanded list)

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

	Q	Expanded
1	(0 S)	
2	(2 A S) (5 B S)	S
3	(4 C A S) (6 D A S) (5 B S)	S,A
4	(6 D A S) (5 B S)	S,A,C
5	~~(6 D B S)~~ (10 G B S) (6 D A S)	S,A,C,B
6	$\boxed{\text{(8 G D A S)}}$ ~~(9 C D A S)~~ ~~(10 G B S)~~	S,A,C,B,D



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

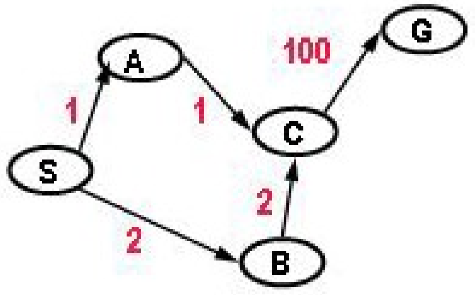
A* (without expanded list)

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

A* and the strict Expanded List

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

	Q	Expanded
1	(0 S)	
2	(3 B S) (101 A S)	S
3	(94 C B S) (101 A S)	B, S
4	(101 A S) (104 G C B S)	C, B, S
5	$\boxed{\text{(104 G C B S)}}$	A, C, B, S



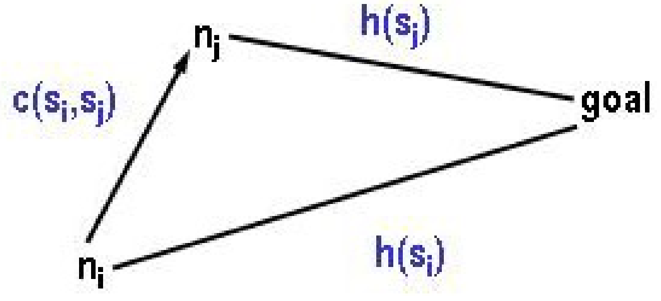
Heuristic Values

A=100	C=90	S=0
B=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.

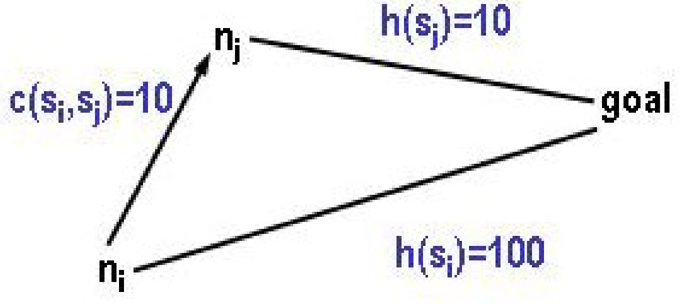
Consistency

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



Consistency Violation

- A simple example of a violation of consistency.
- $h(s_i) - h(s_j) \cdot c(s_i,s_j)$
- In example, 100-10 > 10
- If you believe goal is 100 units from n_i, then moving 10 units to n_j should not bring you to a distance of 10 units from the goal.



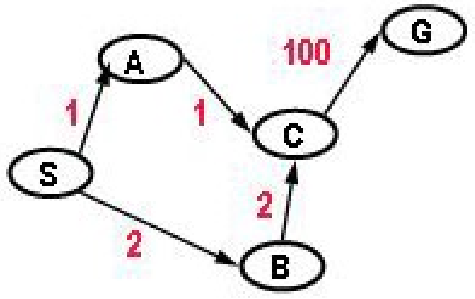
A* (without expanded list)

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

A* (without expanded list)

Note that heuristic is admissible but not consistent

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



Heuristic Values

A=100	C=90	S=90
B=1		G=0

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

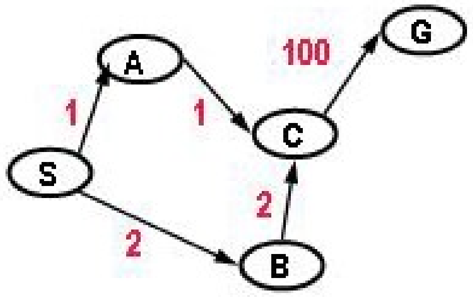
A* (with strict expanded list)

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

A* (with strict expanded list)

Note that this heuristic is admissible and consistent

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



Heuristic Values

A=100	C=100	S=90
B=88		G=0

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

Dealing with inconsistent heuristic

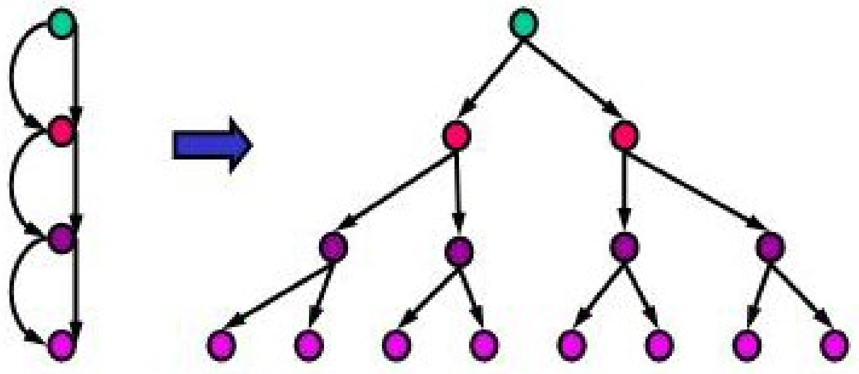
- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify A* so that it detects and corrects when inconsistency has led us astray:
- Assume we are adding $node_1$ to Q and $node_2$ is present in Expanded list with $node_1$.state = $node_2$.state.
- Strict –
  - do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

Worst Case Complexity

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

Worst Case Complexity

- A state space with N states may give rise to a search tree that has a number of nodes that is exponential in N, as in this example.



- Searches without a visited (expanded) list may, in the worst case, visit (expand) every node in the search tree.
- Searches with strict visited (expanded lists) will visit (expand) each state only once.

Optimality & Worst Case Complexity

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

N is number of states in graph