Artificial Intelligence

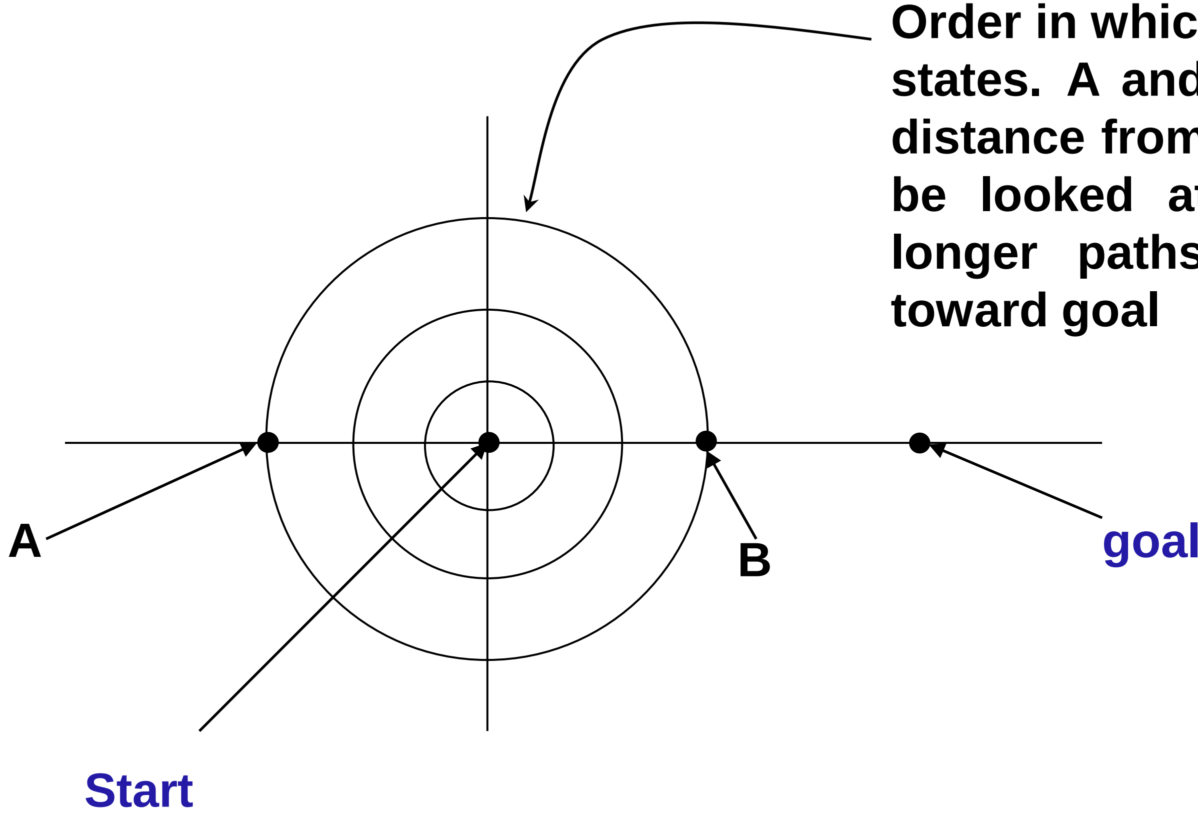
A* and informed searches

Zahoor Tanoli (PhD)

CUI Attock

Why use estimate of goal distance?

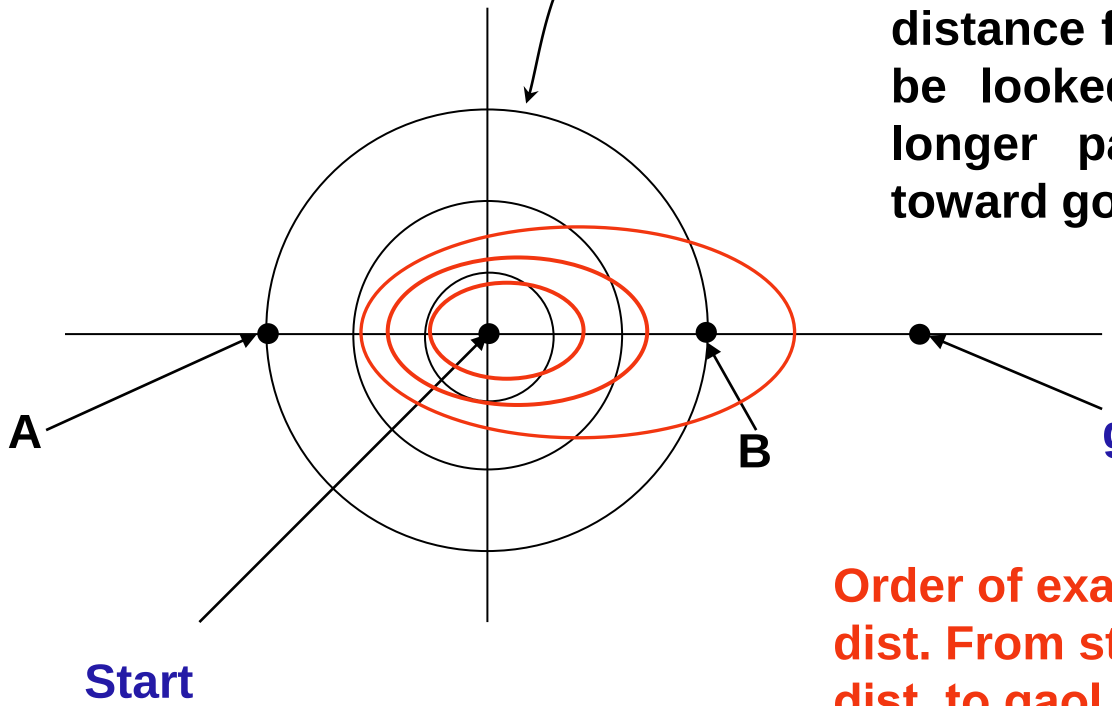
Order in which UC looks at states. A and B are same distance from start, so will be looked at before any longer paths. No “bias” toward goal



Assume states are points in the Euclidean plane

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” toward goal



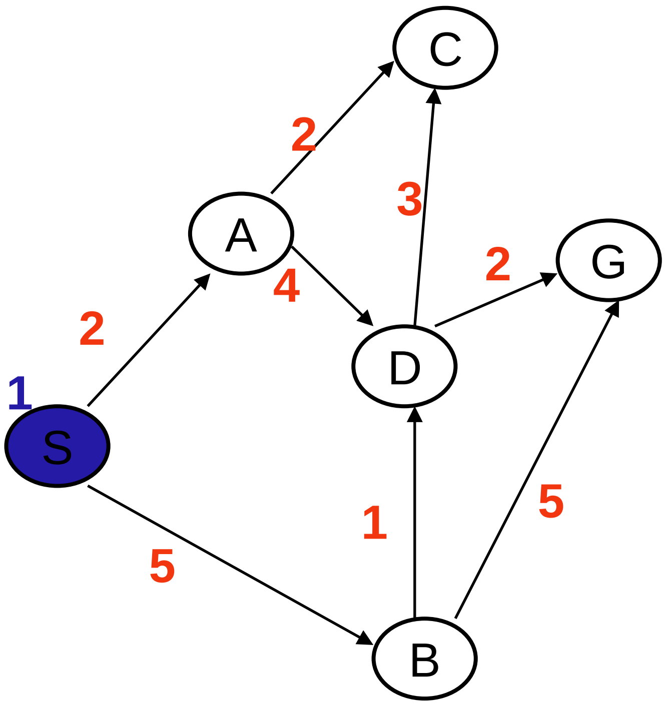
Order of examination using dist. From start + estimate of dist. to gaol. Note “bias” toward the goal; points away from goal look worse

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	<u>(0 S)</u>



Heuristic Values

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

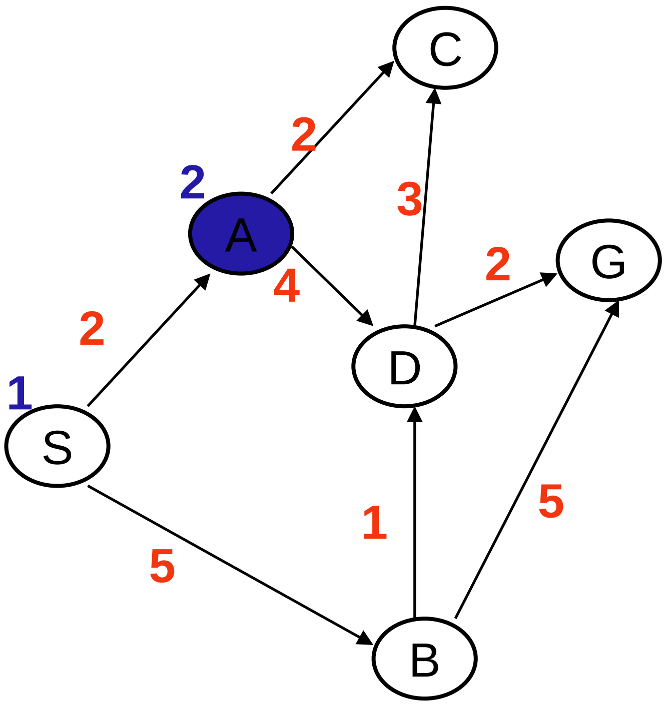
Added paths in blue; <u>underlined</u> paths are chosen for extension

Paths are shown in reverse 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	<u>(0 S)</u>
2	<u>(4 AS)</u> (8 BS)



Heuristic Values

A =2 C=1 S=0

B=3 D=1 G=0

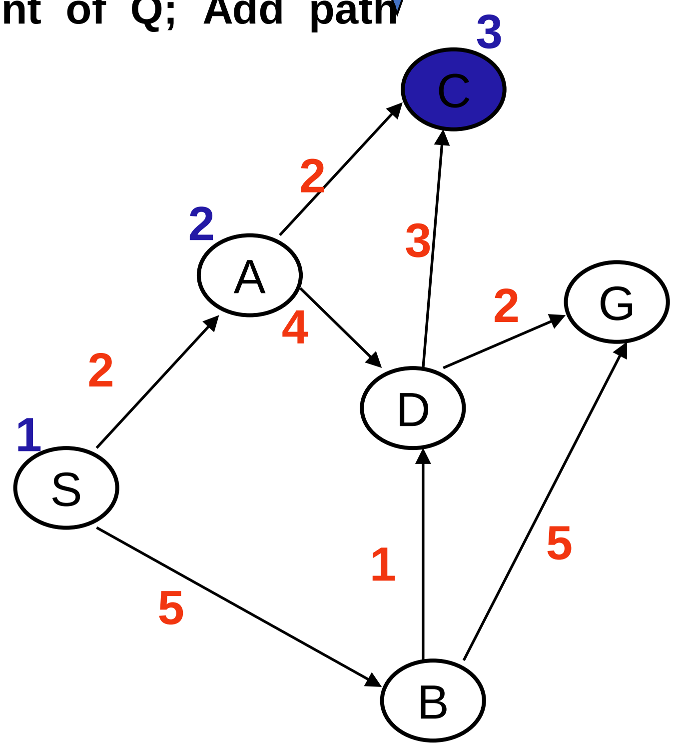
Added paths in blue; <u>underlined</u> paths are chosen for extension

Paths are shown in reverse 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	<u>(0 S)</u>
2	<u>(4 AS)</u> (8 BS)
3	(5 CAS) (7 DAS) (8 BS)



Heuristic Values

A =2 C=1 S=0

B=3 D=1 G=0

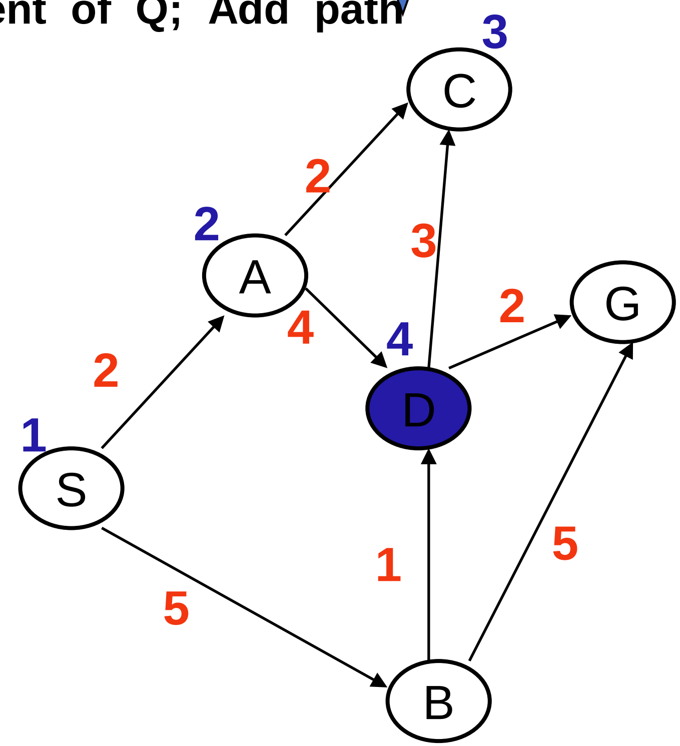
Added paths in blue; <u>underlined</u> paths are chosen for extension

Paths are shown in reverse 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 AS) (8 BS)
3	(5 CAS) (7 DAS) (8 BS)
4	(7 DAS) (8 BS)



Heuristic Values

A =2 C=1 S=0

B=3 D=1 G=0

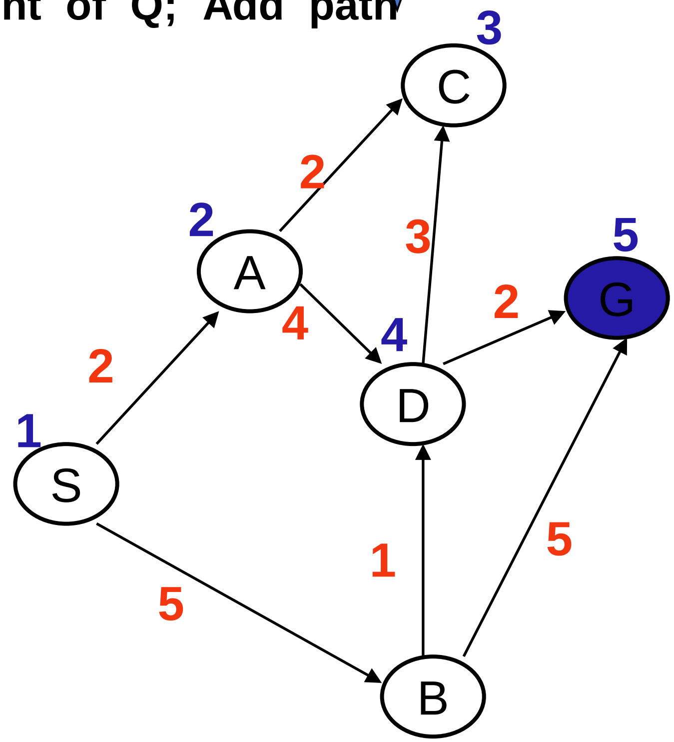
Added paths in blue; underlined paths are chosen for extension

Paths are shown in reverse 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	<u>(0 S)</u>
2	<u>(4 AS)</u> (8 BS)
3	(5 CAS) (7 DAS) (8 BS)
4	(7 DAS) (8 BS)
5	(8 GDAS) (10 CDAS) (8 BS)



Heuristic Values

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

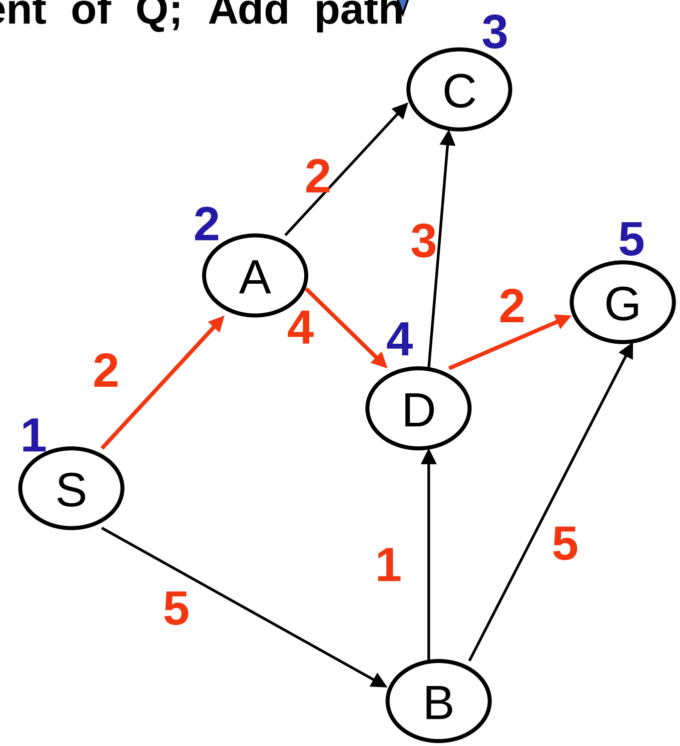
Added paths in blue; <u>underlined</u> paths are chosen for extension

Paths are shown in reverse 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	<u>(0 S)</u>
2	<u>(4 AS)</u> (8 BS)
3	(5 CAS) (7 DAS) (8 BS)
4	(7 DAS) (8 BS)
5	(8 GDAS) (10 CDAS) (8 BS)



Heuristic Values

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

Added paths in blue; <u>underlined</u> paths are chosen for extension

Paths are shown in reverse 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?

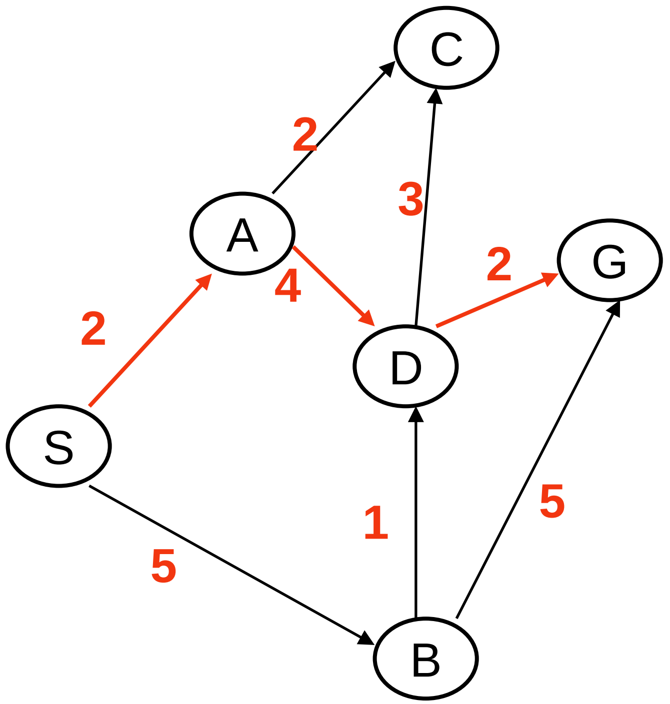
A is ok

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

6	2	8
	3	5
4	7	1

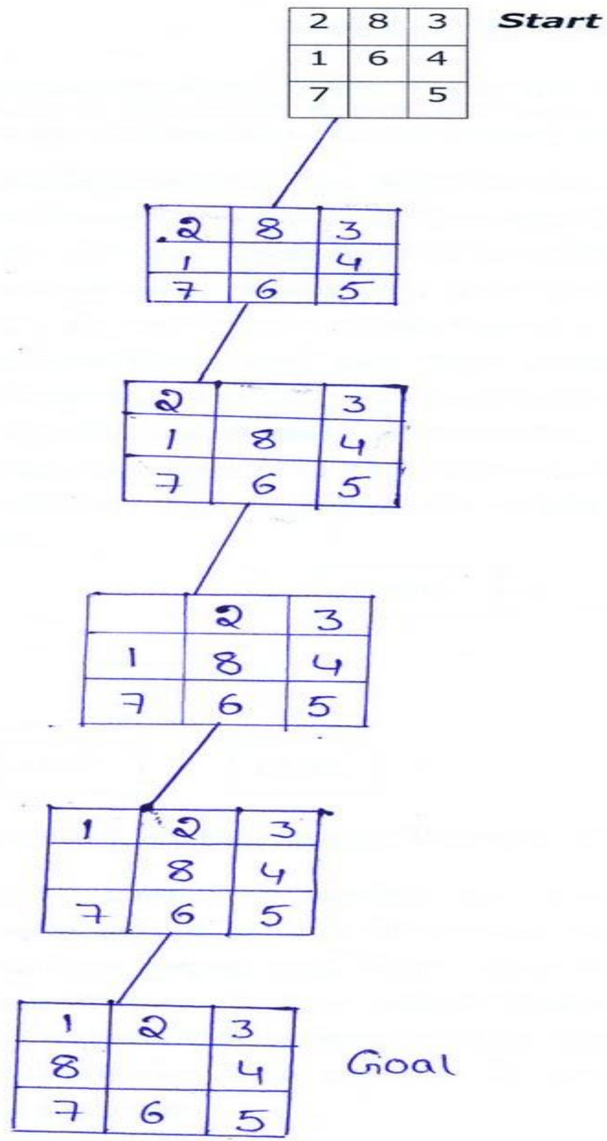
$\Rightarrow$

1	2	3
8		4
7	6	5

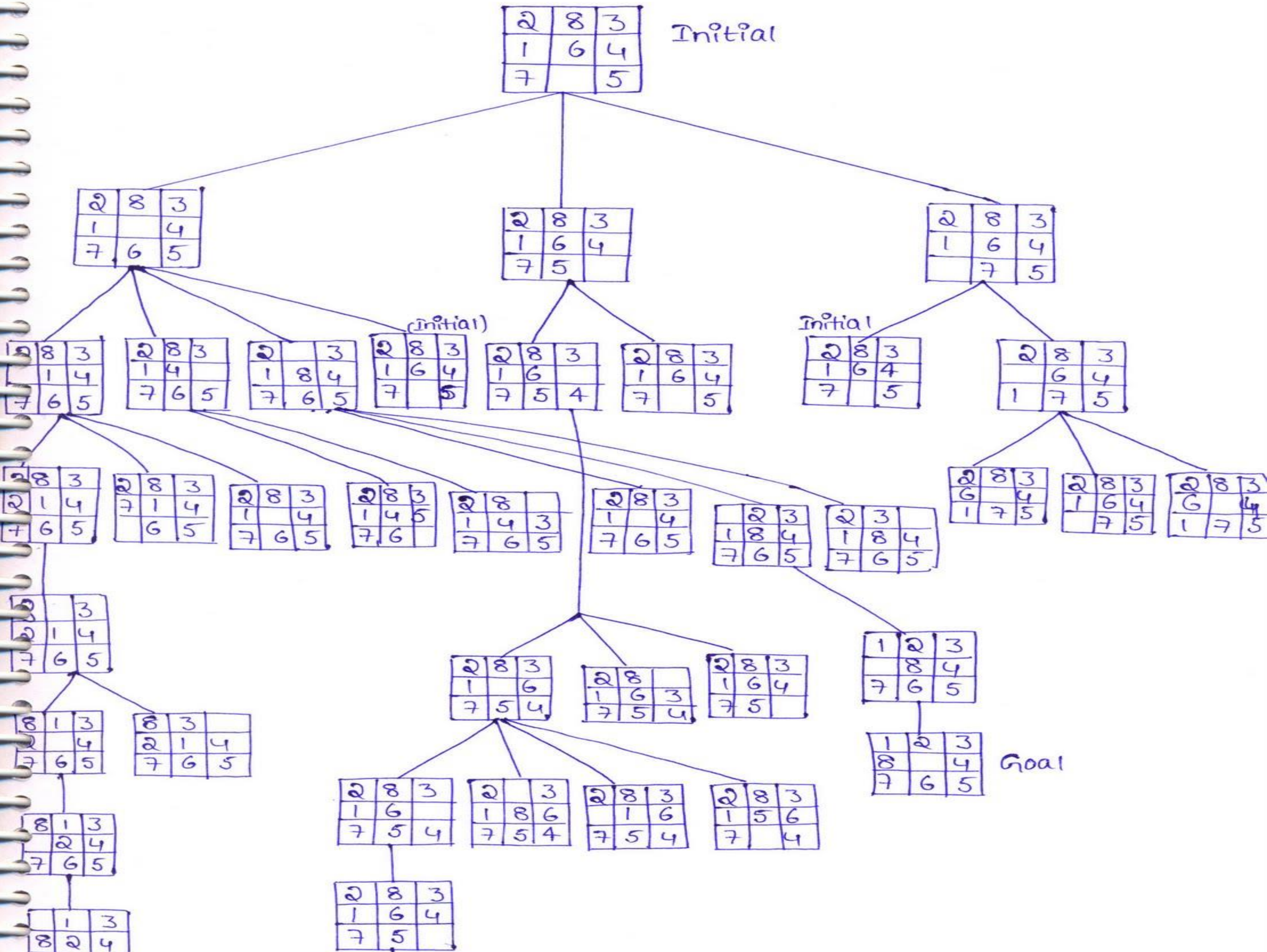
Alternative underestimates of "distance" (number of moves) to goal:

1. **Number of misplaced tiles (7 in example above)**
2. **Sum of Manhattan distance of tiles 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**

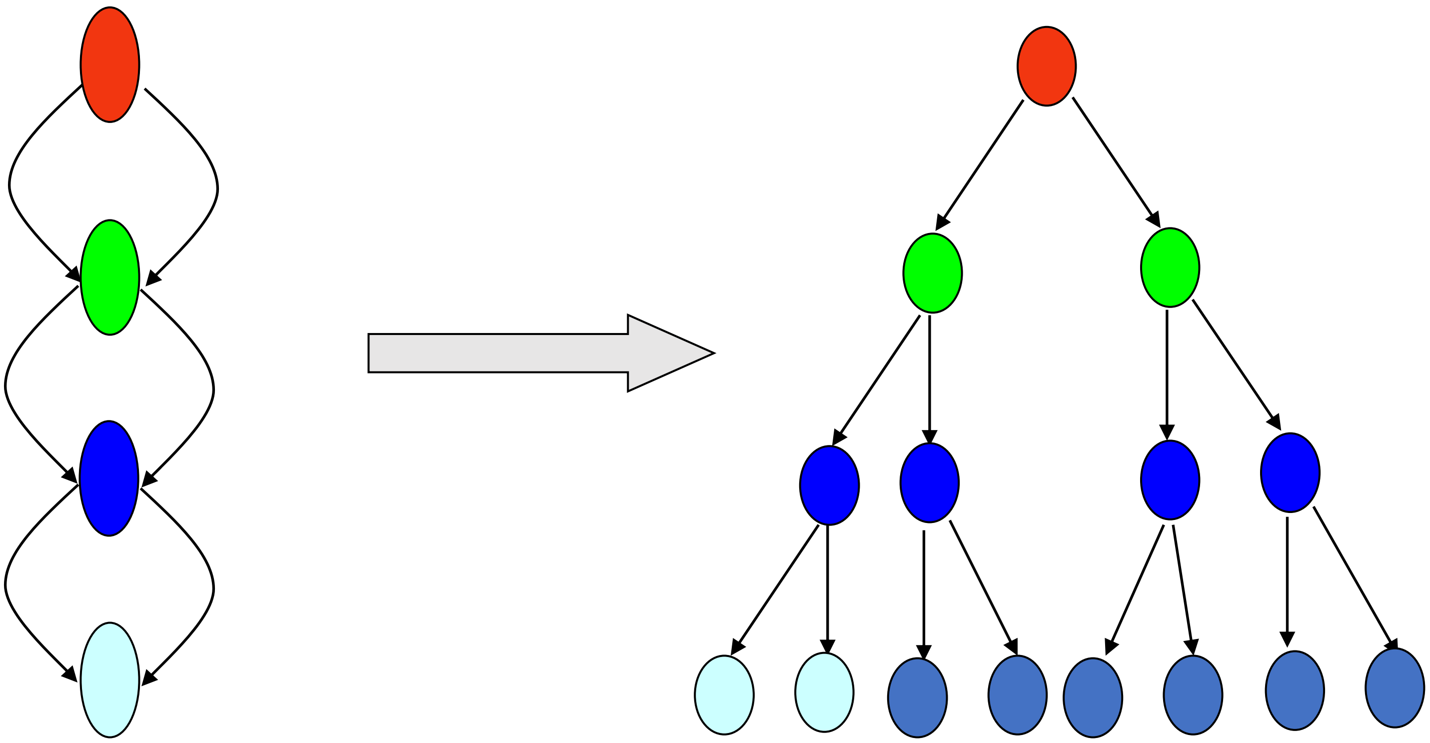
The second of these is much better at predicting actual number of moves



Searrh tree for 8-puzzle


Initial
(Initial)
Initial
Goal

States Vs Paths



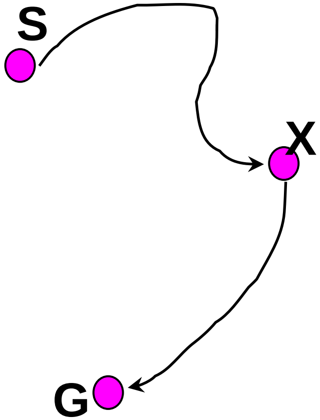
Optimality can be achieved without Visited List but is there any thing else we can use to avoid worst case cost

Dynamic Programming Optimality Principal

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 Principal”



Dynamic Programming Optimality Principal

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 Principal"

- **This means that we only need to keep the single best path from S to any state X; If we find a new path to a state already in Q, discard the longer one.**

- **Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.**

- **So, once expand one path to State X, we don't need to consider (extend) any other paths to X. We can keep a list of these States, Call it Expanded. If the State of the search node we pull off of Q is in the Expanded list. We discard the node. When we use the Expanded list this way, we call it "Strict".**

- **Note that UC without this is still correct, but inefficient for searching graphs.**

Simple Optimal Search Algorithm (Uniform Cost)

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

1. Initialize Q with search node (S) as only entry;

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 a goal)

4. (Otherwise) Remove N from Q
5. -
6. Find all the children of state (N) and create all the one-step extensions of N to each descendent.

7. Add all the extended paths to Q;
8. Go to Step 2.

Simple Optimal Search Algorithm (UC+ 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 a goal)

4. (Otherwise) Remove N from Q

5. If State (N) is 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.

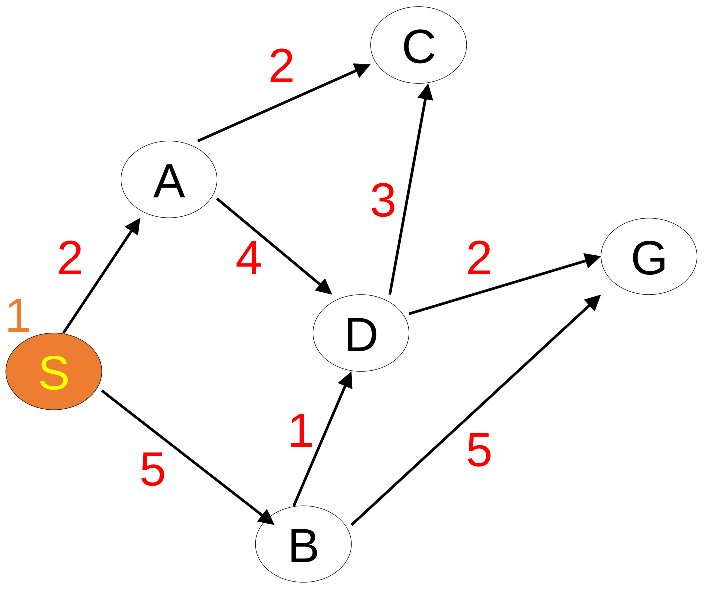
7. Add all the extended paths to Q, If descendent state already in Q, keep only shorter path to the State in Q.

8. Go to step2.

Uniform Cost (With Strict Expanded List)

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

	Q	Expanded
1	**<u>(0 S)</u>**	

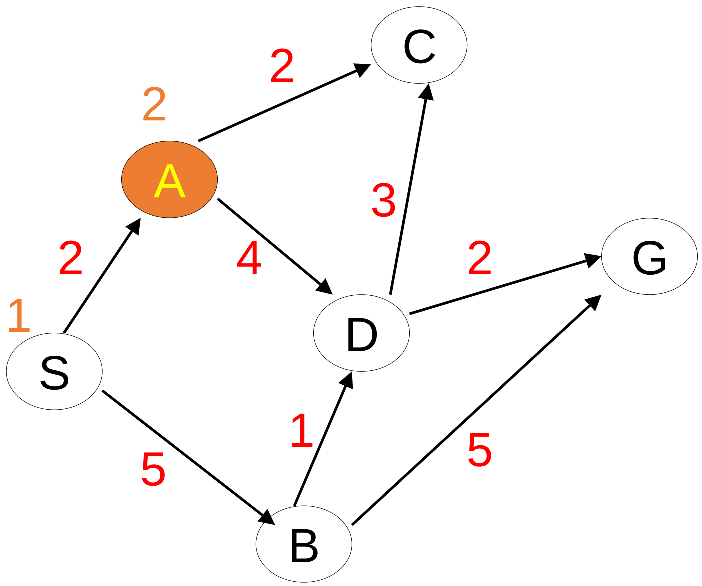


Added paths in blue; <u>Underlined paths</u> are chosen for extension.
Paths are shown in Reversed Order, the node's state is the first entry.

Uniform Cost (With Strict Expanded List)

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

	Q	Expanded
1	**<u>(0 S)</u>**	
2	**<u>(2 AS)</u> (5 BS)**	**S**

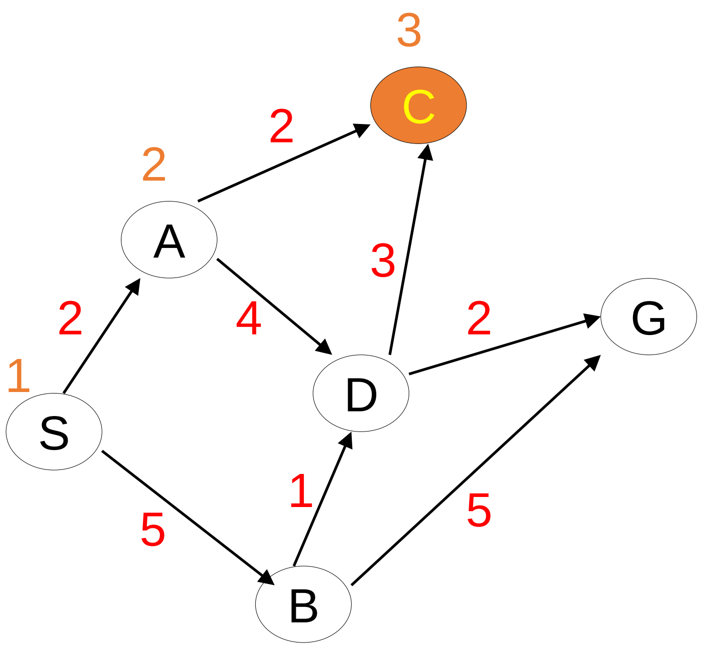


Added paths in blue; <u>Underlined paths</u> are chosen for extension.
Paths are shown in Reversed Order, the node's state is the first entry.

Uniform Cost (With Strict Expanded List)

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

	Q	Expanded
1	**<u>(0 S)</u>**	
2	**<u>(2 AS)</u> (5 BS)**	**S**
3	**<u>(4 CAS)</u> (6 DAS) (5 BS)**	**S, A**

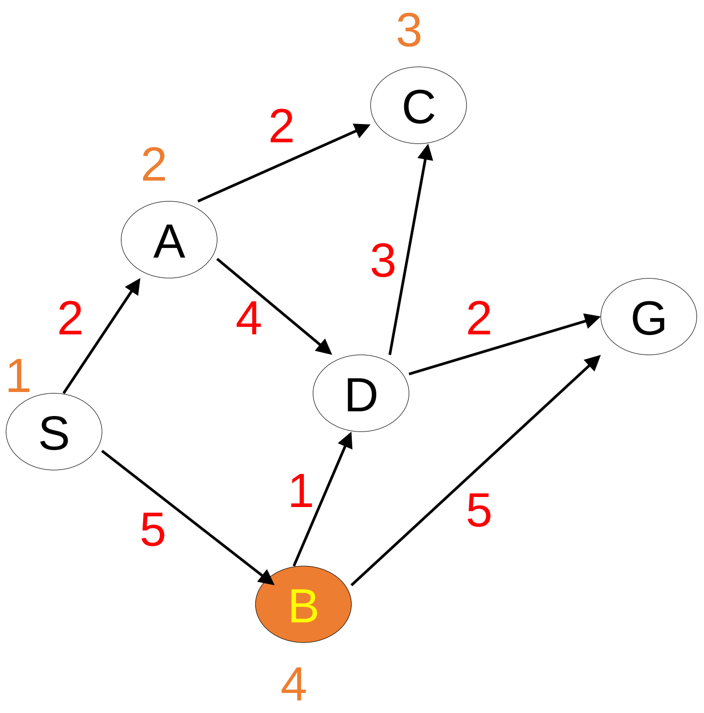


Added paths in blue; <u>Underlined paths</u> are chosen for extension.
Paths are shown in Reversed Order, the node's state is the first entry.

Uniform Cost (With Strict Expanded List)

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

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



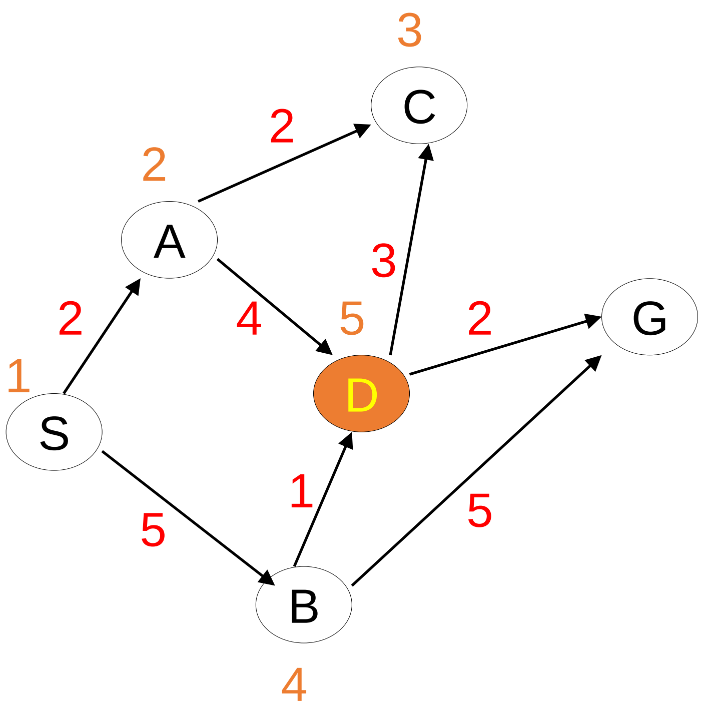
Added paths in blue; <u>Underlined paths</u> are chosen for extension.

Paths are shown in Reversed Order, the node's state is the first entry.

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	**(4CAS)(6DAS)(5BS)**	**S, A**
4	**(6DAS)(5BS)**	**S,A,C**
5	**~~(6 DBS)~~(10 GBS)(6 DAS)**	**S,A,C,B**

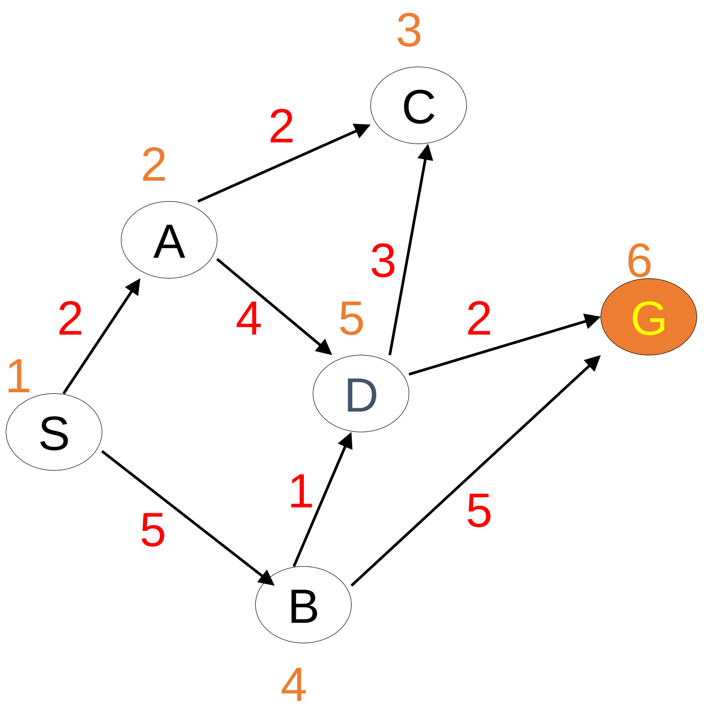


Added paths in blue; Underlined paths are chosen for extension.
Paths are shown in Reversed Order, the node’s state is the first entry.

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 CAS) (6 DAS) (5 BS)**	**S, A**
4	**(6 DAS) (5 BS)**	**S,A,C**
5	**~~(6 DBS)~~ (10GBS) (6 DAS)**	**S,A,C,B**
6	**(8 GDAS) ~~(9 CDAS)~~ ~~(10 GBS)~~**	**S,A,C,B.D**

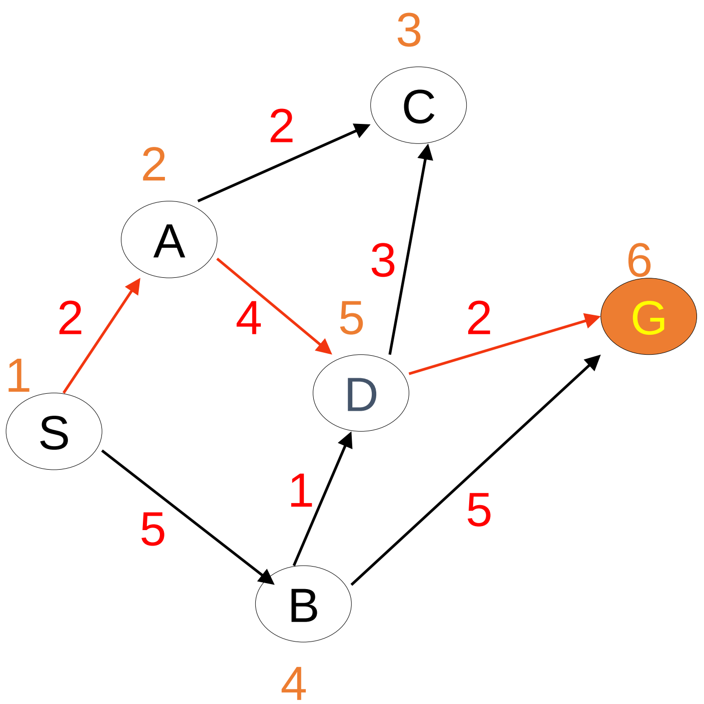


Added paths in blue; Underlined paths are chosen for extension.
Paths are shown in Reversed Order, the node's state is the first entry.

Uniform Cost (With Strict Expanded List)

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

	Q	Expanded
1	**<u>(0 S)</u>**	
2	**<u>(2 A S)</u> (5 B S)**	**S**
3	**<u>(4 CAS)</u> (6 DAS) (5 BS)**	**S, A**
4	**(6 DAS) <u>(5 BS)</u>**	**S,A,C**
5	**~~(6 DBS)~~ (10GBS) <u>(6 DAS)</u>**	**S,A,C,B**
6	**$\boxed{\text{(8 GDAS)}}$ ~~(9 CDAS)~~ ~~(10 GBS)~~**	**S,A,C,B.D**



Added paths in blue; <u>Underlined paths</u> are chosen for extension.
Paths are shown 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.

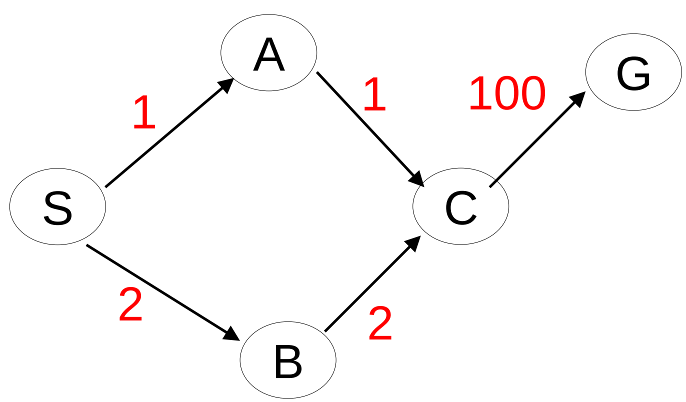
- A* (without Expanded List) and with admissible heuristic is guaranteed to find optimal paths--- those with smallest path cost.

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.

Lets Consider the following Example:

- 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.

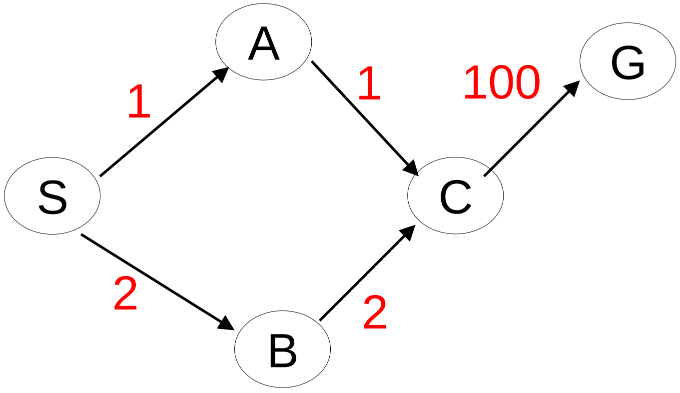


Heuristic Values

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

A* and the strict Expanded List

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



Heuristic Values

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

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

    Goal state have a heuristic estimate of Zero

  - $h(s_i) - h(s_j) \leq c(s_i, s_j)$, for n_j a child of n_i

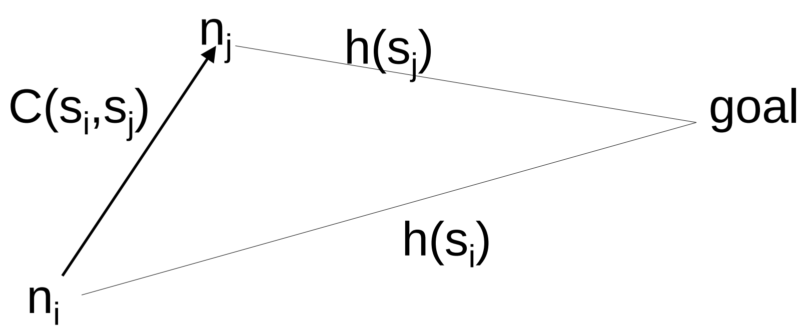
    The difference in the heuristic estimate between one state and its descendant must be less than or equal to the actual path cost on the edge connecting them

Consistency

- To enable implementing A* using the strict Expanded list, h needs to satisfy the following **consistency** (also known as **monotonicity**) conditions.

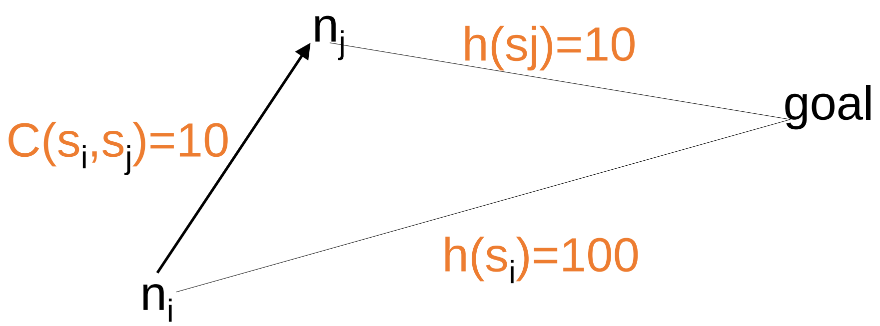
  - $h(s_i) = 0$, if n_i is a goal
  - $h(s_i) - h(s_j) \leq c(s_i, s_j)$, for n_j a child of n_i

- That is, the heuristic cost in moving from one entry to the next cannot decrease by more than the arc cost between the states. This is a kind of triangle inequality. This condition is a highly desirable property of a heuristic function and often simply assumed.



Consistency Violation

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



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.

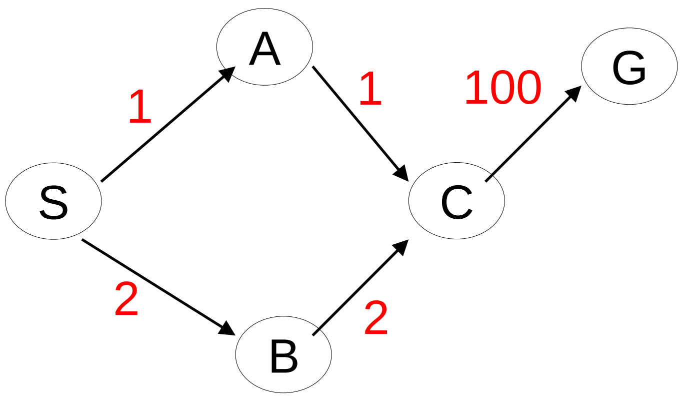
- A* (without Expanded List) and with admissible heuristic is guaranteed to find optimal paths--- those with smallest path cost.

- This is true even if heuristic is not consistent.

A* (Without Expanded List)

Note that the heuristic is admissible but not consistent

	Q
1	(0 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	$\boxed{\text{(102 G C A S)}}$ (104 G 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.

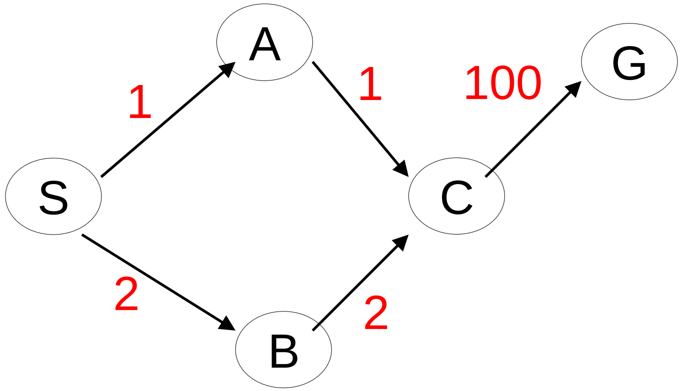
A* (with strict Expanded List)

- 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 descendent 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	(91 B S)(90 A S)	S
3	(90 C A S) (91 B S)	A, S
4	(102 G C A S)(91 B S)	C, A, S
5	(102 G C A S)	G, C, A, S



Heuristic Values A=89 C=88, S=90, B=89, 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

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

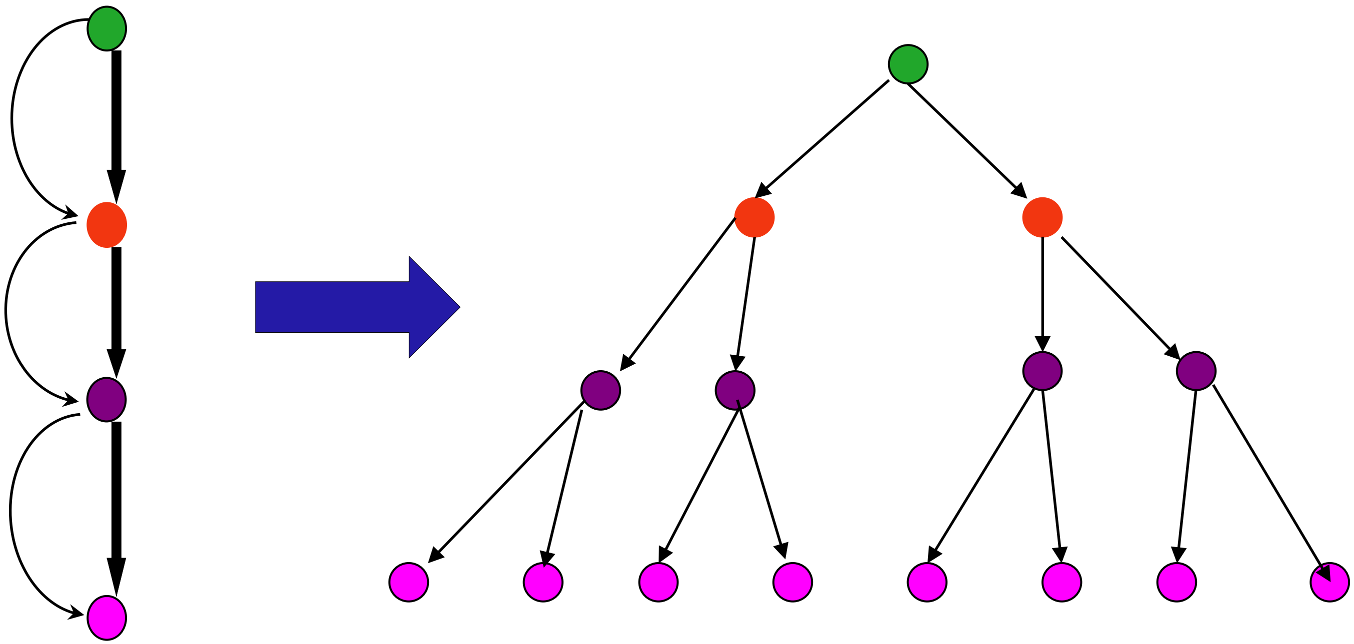
- Non-Strict Expanded list-
  - If $node_1$.path_length< $node_2$.path_length, then
    - Delete node2 from Expanded list
    - Add $node_1$ to Q

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

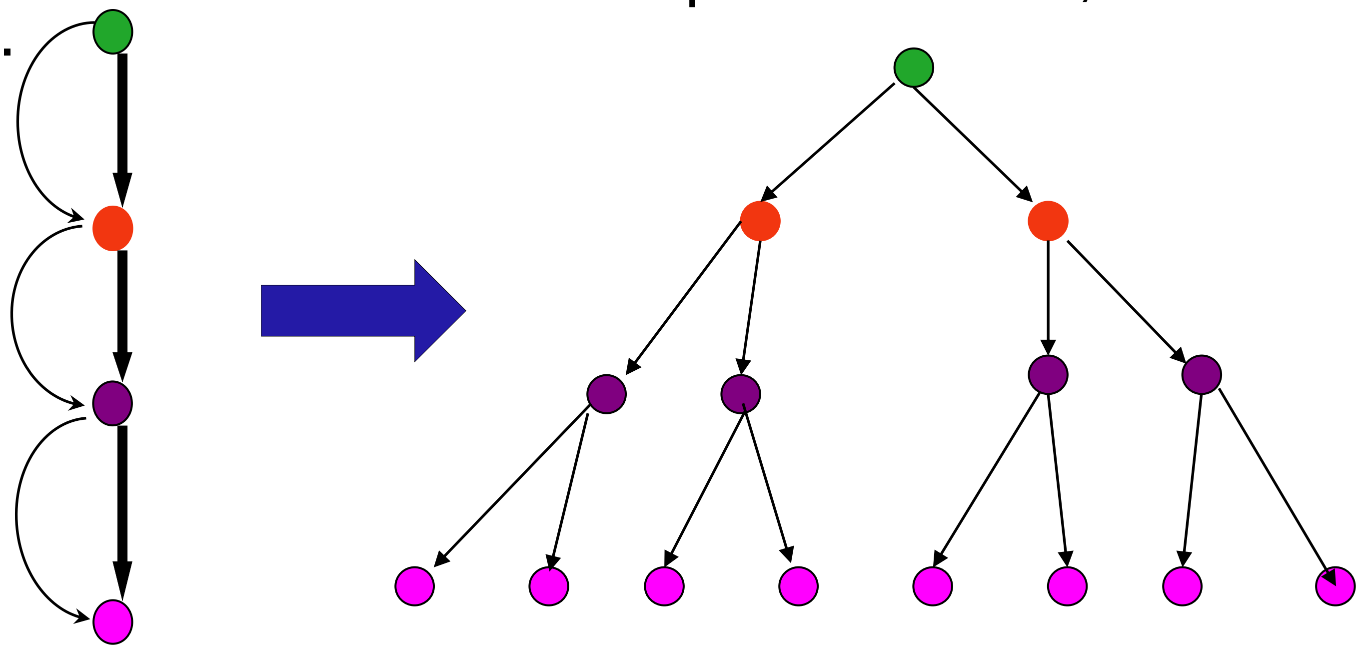
- A state space with N states may give rise to a search tree that has a number of nodes that is exponential in N, as in this example.



N-States **2^N -Nodes**

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	Not Strict	Yes	>N

N is number of states in Graph

Questions

?