

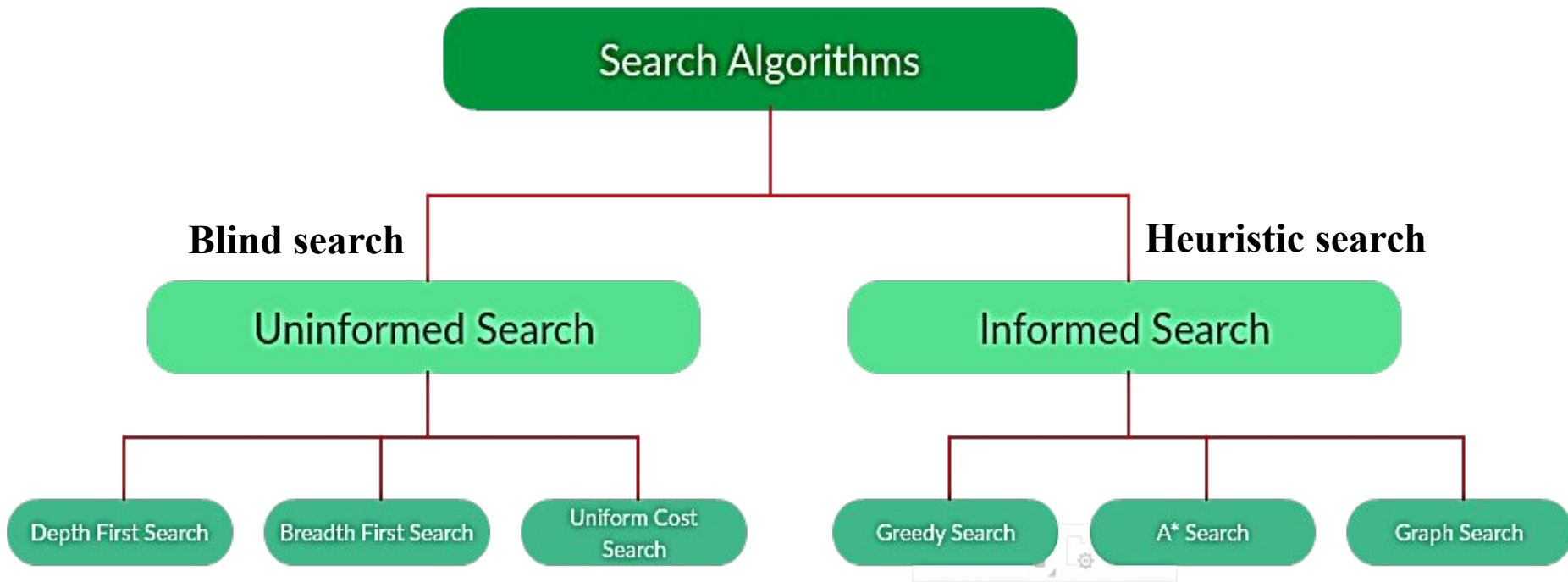
# Unit-3

## Search Techniques

# Search Algorithms in AI

- Artificial Intelligence is the study of building agents that act rationally.
  - Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.
- A search problem consists of:
  - A **State Space**. Set of all possible states where you can be.
  - A **Start State**. The state from where the search begins.
  - A **Goal Test**. A function that looks at the current state returns whether or not it is the goal state.
- The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

# Types of Search Algorithms



# Uninformed vs. informed search

- **Uninformed search strategies**

- Also known as “blind search,” uninformed search strategies use no information about the likely “direction” of the goal node(s)
- Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional

- **Informed search strategies**

- Also known as “heuristic search,” informed search strategies use information about the domain to (try to) head in the general direction of the goal node(s)
- Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A\*

# Uninformed/Blind search strategies

- Topics to be covered...
  - Breadth First Search
  - Depth First Search
  - Uniform Cost Search
  - Depth limited search
  - Iterative Deepening Depth First Search

### ***Algorithm: Breadth-First Search***

1. Create a variable called *NODE-LIST* and set it to the initial state.
2. Until a goal state is found or *NODE-LIST* is empty:
  - (a) Remove the first element from *NODE-LIST* and call it *E*. If *NODE-LIST* was empty, quit.
  - (b) For each way that each rule can match the state described in *E* do:
    - (i) Apply the rule to generate a new state,
    - (ii) If the new state is a goal state, quit and return this state.
    - (iii) Otherwise, add the new state to the end of *NODE-LIST*.

### ***Algorithm: Depth-First Search***

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
  - (a) Generate a successor, *E*, of the initial state. If there are no more successors, signal failure.
  - (b) Call Depth-First Search with *E* as the initial state.
  - (c) If success is returned, signal success. Otherwise continue in this loop.



A, B, C, D, E, F

**BFS**



A, B, D, C, E, F

**DFS**

- Advantages of Breadth-First Search:
  - BFS will not get trapped exploring a blind path. This contrasts with DFS, which may follow a single, unfruitful path for a long time.
  - If there is a solution, then BFS is guaranteed to find it.
- Advantages of Depth-First Search:
  - DFS requires less memory since only the nodes of current path are stored. This contrasts with BFS, where all of the tree that has so far been generated must be stored.
  - DFS may find a solution without examining much of the search space at all. DFS can stop when one of them is found.

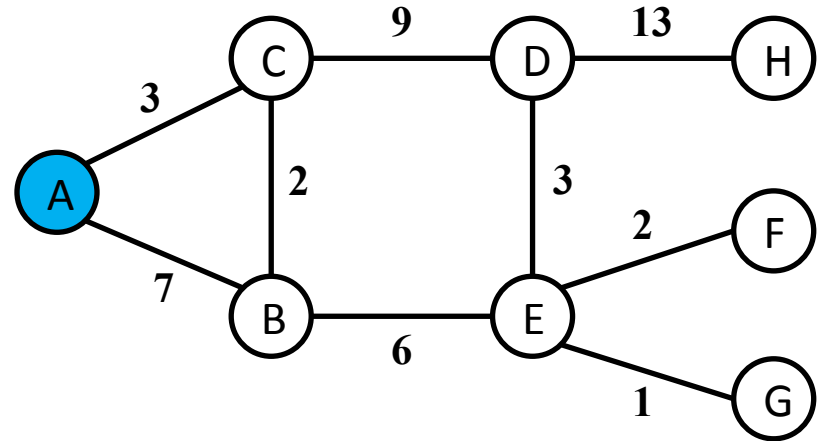
Example : TSP (Traveling Salesperson Problem)



# Uniform Cost Search

(uninformed search, backtracking)

- Used for **weighted** graph/tree
- Objective: find path to goal state with **lowest cumulative cost**
- Node expansion is based on path cost
- **Priority queue** is used for implementation (high priority to minimum cost)
- **Advantage**: optimal solution
- **Disadvantage**: may have infinite loop



If **G** is goal state:

**A -> C -> B -> E -> G**

**TOTAL COST: 3 + 2**

**+ 6 + 1 = 12**

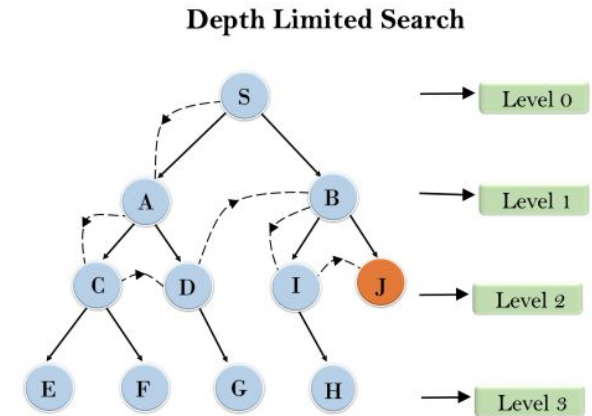
If **H** is goal state:

**A -> C -> B -> E -> G -> E -> F -> E -> D -> H**

**TOTAL COST: 3 + 2 + 6 + 1 + 1 + 2 + 2 + 3 + 13**

# Depth-Limited Search

- A depth-limited search algorithm is similar to depth-first search with a **predetermined limit**.
- Depth-limited search can solve the drawback of the **infinite** path in the Depth-first search.
- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
  - **Standard failure value**: It indicates that problem does not have any solution.
  - **Cutoff failure value**: It defines no solution for the problem within a given depth limit.
- Advantages:
  - Depth-limited search is **Memory efficient**.
- Disadvantages:
  - Depth-limited search also has a disadvantage of **incompleteness**.
  - It may **not be optimal** if the problem has more than one solution.



# Bidirectional Search

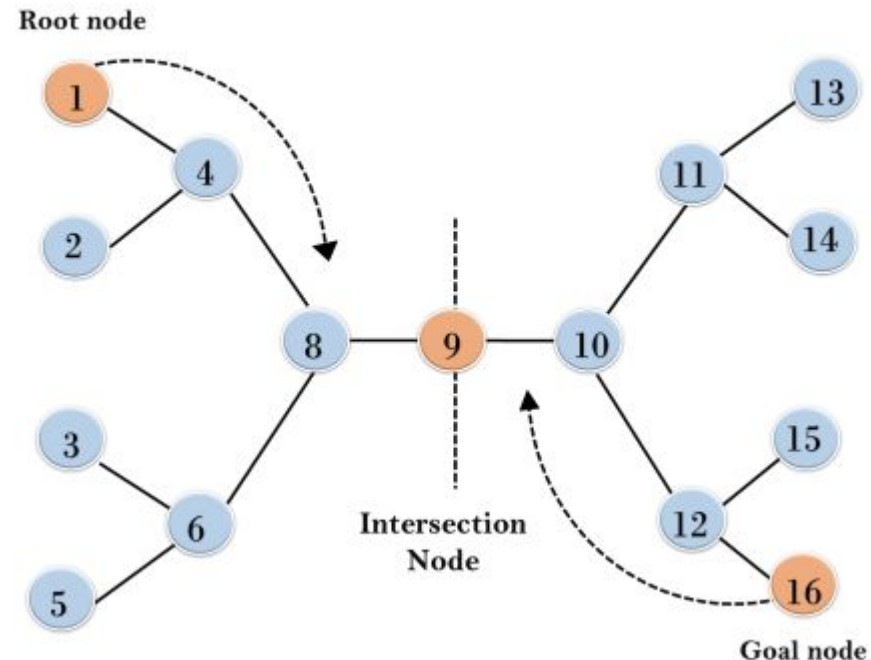
- Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

## Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

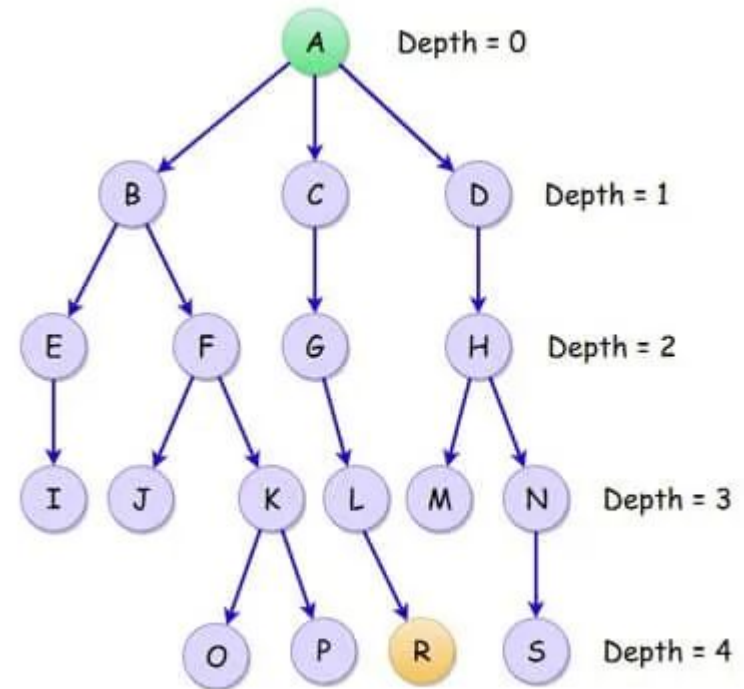
## Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.



# Iterative Deepening Depth-First Search

- The algorithm do a limited depth-first search up to a fixed “**limited depth**”.
- Then we keep on incrementing the depth (**iteratively**) limit by iterating the procedure unless we have found the **goal** node or have traversed the whole tree whichever is earlier.



## *DEPTH LIMITS*

0

1

2

3

4

## *IDDFS*

A

A B C D

A B E F C G D H

A B E I F J K C G L D H M N

A B E I F J K O P C G L R D H M N S

# Informed/Heuristic search strategies

## Topics to be covered...

- Introduction to heuristic functions
- Generate-and-test
- Best-first search
  - Greedy Best First Search
  - A\* Algorithm
- Problem reduction
  - General Approach
  - AO\* Algorithm
- Hill Climbing
  - Simple Hill Climbing
  - Steepest-Ascent Hill Climbing
  - Simulated Annealing
- Local Beam Search

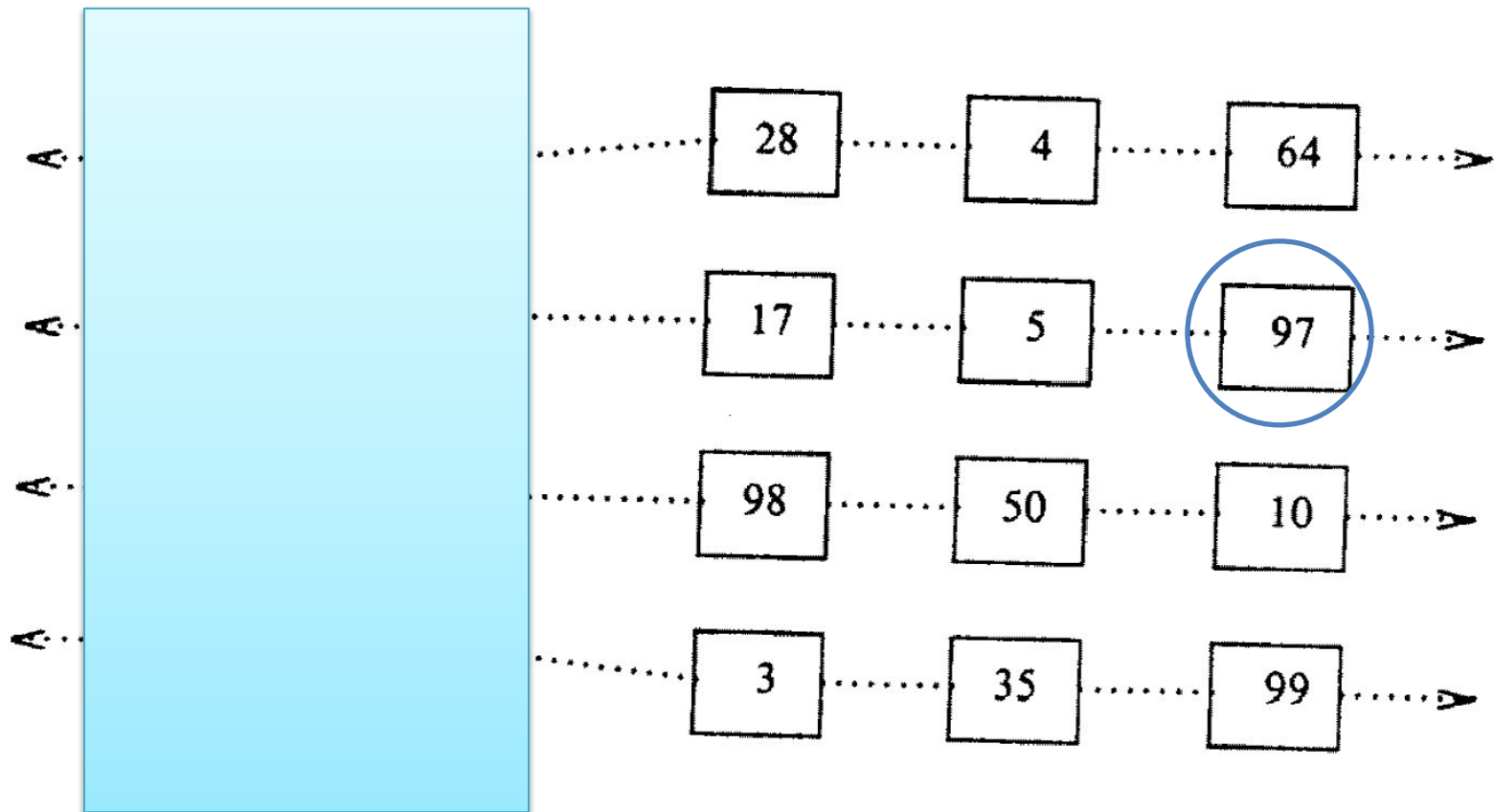
# Heuristic Search

- A **heuristic function** or simply **heuristic** is a technique which improves the efficiency of a search process, possibly by sacrificing claims of completeness.

OR

- A **heuristic** is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.
- Heuristics are tour guides.

# Example (heuristic?)



- Search for block number  $97 = 97 \bmod 4 = 1$ .
- So block number 97 is available in bl no. 1

# Example: Heuristic Function

- Consider the following 8-puzzle problem where we have a start state and a goal state.
- Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state.
- There can be four moves either left, right, up, or down.
- There can be several ways to convert the current/start state to the goal state, but, we can use a **heuristic function  $h(n)$**  to solve the problem more efficiently.

1	2	3
8	6	
7	5	4

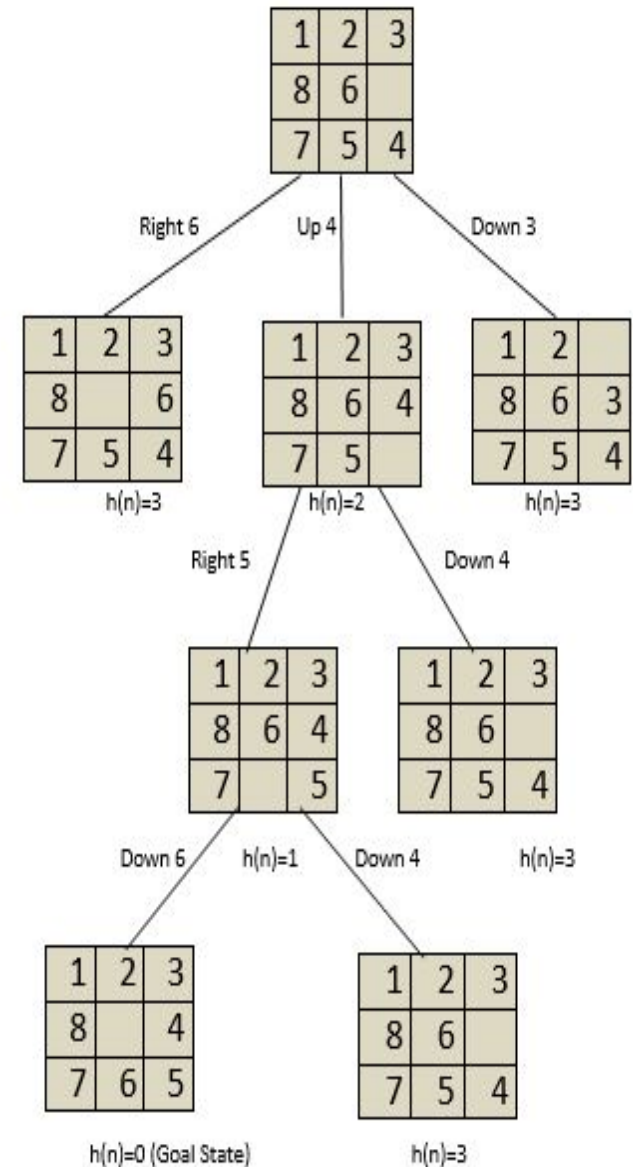
Start State

1	2	3
8		4
7	6	5

Goal State



- Two classes of local search algorithms exist.
- The first one is that of greedy or **non-randomized** algorithms. These algorithms proceed by changing the current assignment by always trying to decrease (or at least, non-increase) its cost.
- The main problem of these algorithms is the possible presence of **plateaus**, which are regions of the space of assignments where no local move decreases cost.
- The second class of local search algorithm have been invented to solve this problem. They escape these plateaus by doing random moves, and are called **randomized** local search algorithms.



- A heuristic function, or simply a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.
- A heuristic function, is a function that calculates an approximate cost to a problem (or ranks alternatives).
- For example the problem might be finding the shortest driving distance to a point. A heuristic cost would be the straight line distance to the point. It is simple and quick to calculate, an important property of most heuristics. The true distance would likely be higher as we have to stick to roads and is much harder to calculate.
- Well designed heuristic functions can play an important part in efficiently guiding a search process toward a solution.

- Examples of Heuristic Functions:
  - *Chess* : the material advantage of our side over the opponent
  - *Traveling Salesperson* : the sum of the distances so far
  - *Tic-tac-toe* : 1 for each row in which we could win and in which we already have one piece plus 2 for each such row in which we have two pieces.

# Generate-and-test

(Heuristic Search Techniques)

# Generate-and-test

(Heuristic technique, DFS with backtracing)

- This is the simplest of all approaches. It consists of the following steps:

## *Algorithm: Generate-and-test*

1. Generate a possible solution.
2. Test to see if this is actually a solution.
3. Quit if a solution has been found.  
Otherwise, return to step 1.

- Acceptable for simple problems.
- Inefficient for problems with large space.
- **Exhaustive** generate-and-test.
- **Heuristic** generate-and-test: not consider paths that seem unlikely to lead to a solution.

**Two digit – 3 parts PIN number**

e.g. 135379

00 00 00

00 00 01

.....

.....

**EXAMPLE:**

Exhaustive: Will take  $(100)^3$

Find good heuristic ???

(domain knowledge available)

For example, the PIN number uses only **prime** numbers

e.g.  $(25)^3$

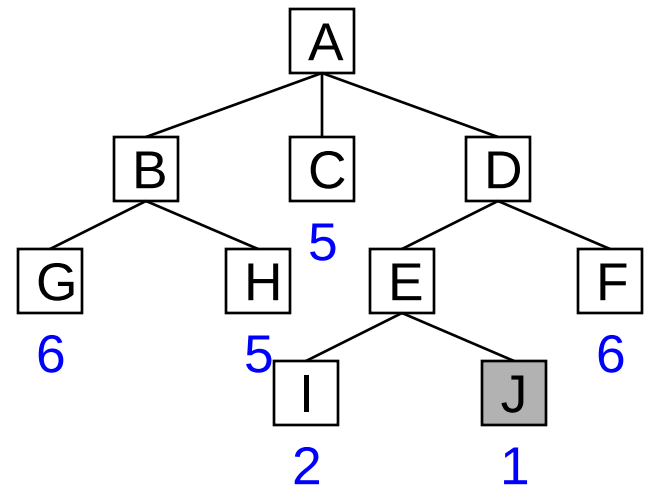
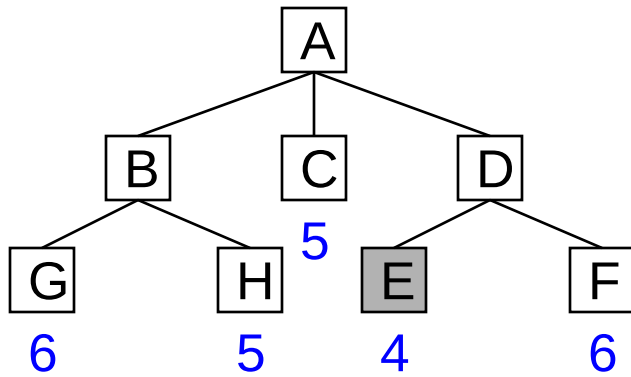
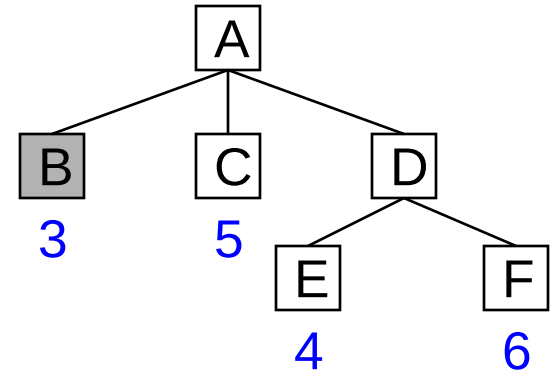
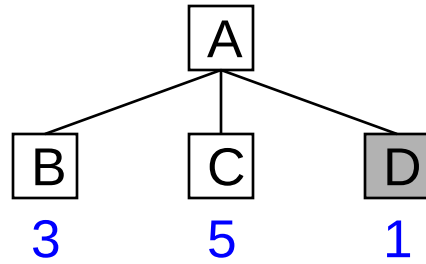
# Greedy Best-First Search

(BFS+DFS, Heuristic Search Techniques)

# Best-first search

- Best-first search – way of combining the advantages of BFS and DFS into a single method.
  - **Depth-first search**: not all competing branches having to be expanded.
  - **Breadth-first search**: not getting trapped on dead-end paths.
- ⇒ Combining the two is to **follow a single path at a time**, but **switch paths** whenever some competing path look more promising than the current one.





- **OPEN**: nodes that have been generated, but have not examined.
  - This is organized as a **priority queue**.
- **CLOSED**: nodes that have already been examined.
  - Whenever a new node is generated, **check** whether it has been **generated before**.

# Algorithm: Best-First Search

1.  $OPEN = \{\text{initial state}\}.$
2. Loop until a goal is found or there are no nodes left in OPEN:
  - (a) Pick the best node in **OPEN**
  - (b) Generate its successors
  - (c) For each successor do:
    - (i) If it has **not been generated** before, evaluate it, add it to OPEN, and record its parent.
    - (ii) If it has **been generated** before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

- We are in need of a heuristic function  $f'$  that estimates the merits of each node we generate.

$$f' = g + h'$$

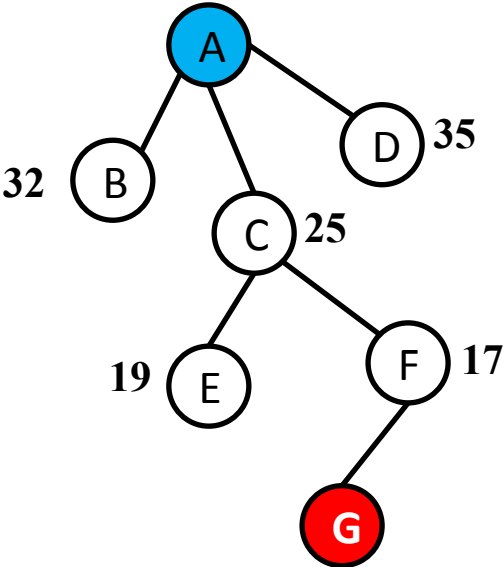
- where function  $g$  is a measure the cost of getting from the initial state to the current node, and
- Function  $h'$  is an estimate of the additional cost of getting the current node to goal state.

# Algorithm: Best-First Search

OPEN = {initial state}.

Loop until a goal is found or there are no nodes left in OPEN:

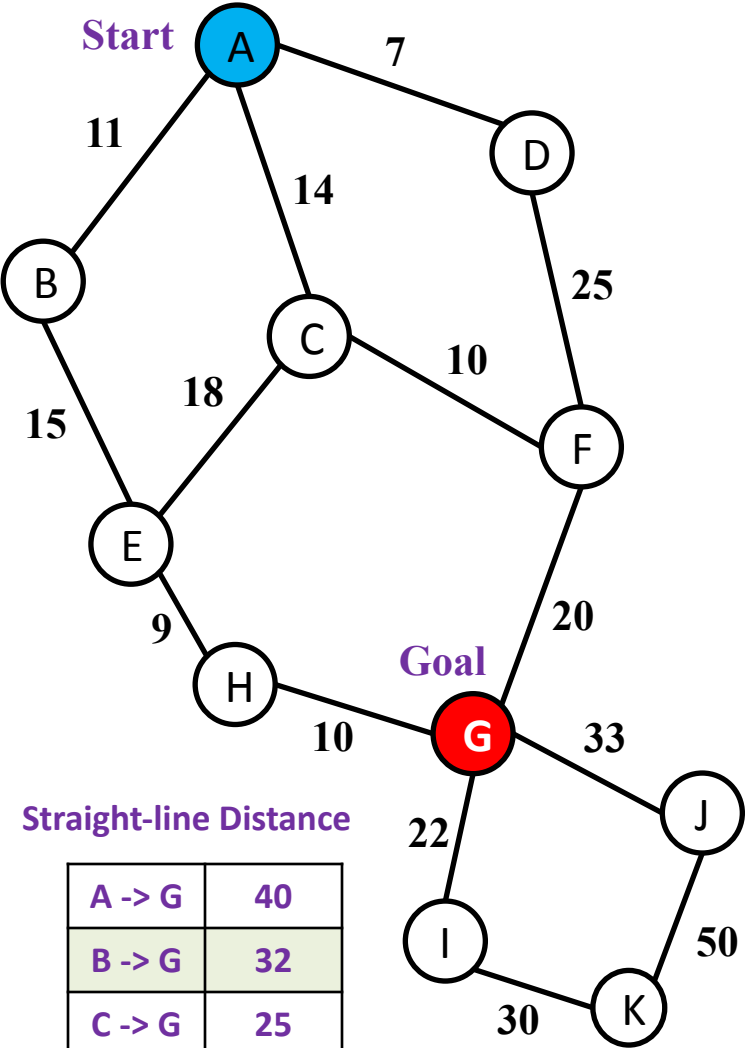
- (a) Pick the best node in **OPEN**
- (b) Generate its successors
- (c) For each successor do:
  - (i) If it has **not been generated** before, evaluate it, add it to OPEN, and record its parent.
  - (ii) If it has **been generated** before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.



A -> C ->  
F -> G

Distance between two points  $P(x_1, y_1)$  and  $Q(x_2, y_2)$  is given by:

$$d(P, Q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad \{\text{Distance formula}\}$$



Straight-line Distance

A -> G	40
B -> G	32
C -> G	25
D -> G	35
E -> G	19
F -> G	17
H -> G	10
G -> G	0

# A\* Algorithm

(Heuristic Search Techniques)

# A\* Algorithm

- The best-first search algorithm is a simplification of A\* algorithm, which was first presented by Hart [1968, 1972].
- This algorithm uses the same  $f'$ ,  $g$  and  $h'$  functions as well as the lists OPEN and CLOSED.

- This algorithm was given by **Hart ,Nilsson & Rafael** in 1968.
- A\* is a best first search  **$f(n) = g(n) + h(n)$**   
where ,  $g(n)$  = *sum of edge costs from start to n.*  
 $h(n)$  = ***estimate of lowest cost path from n to goal***
- **$f(n)$  = actual distance so far + estimated distance remaining.**



OPEN  $\square$  (S); Parent (S)  $\square$  NIL; CLOSED  $\square$  NIL

While OPEN  $\neq$  NIL

pick best(lowest f value where  $f = g + h$ ) node  $n$  and add it to CLOSED.

If  $n$  is a goal node,

return success (*path p*)

else

Generate the successors of node  $n$ .

for each  $m$  in successors

case 1:  $m$  is not in OPEN and not in CLOSED.

compute  $h(m)$

parent( $m$ )  $\square$   $n$

$g(m) = g(n) + k(n, m)$  (where  $k$  is the cost from  $n$  to  $m$ )

$f(m) = g(m) + h(m)$

Add  $m$  to OPEN.

case 2:  $m$  is in OPEN

if  $g(n) + k(n, m) < g(m)$

parent( $m$ )  $\square$   $n$

$g(m) = g(n) + k(n, m)$

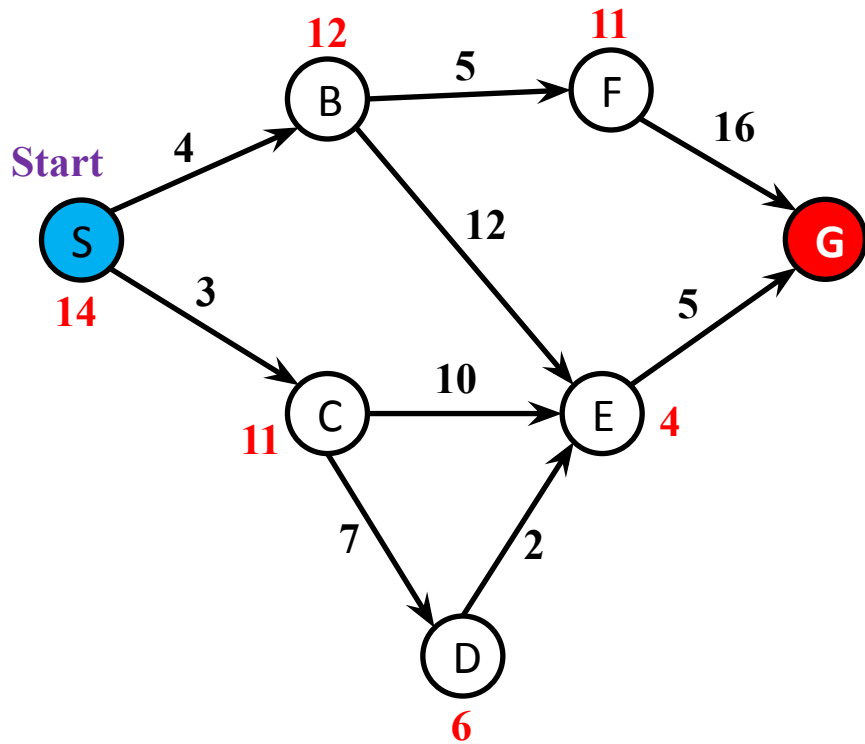
$f(m) = g(m) + h(m)$

case 3:  $m$  is in CLOSED

if better path found then like case 2,

and propagate improved cost to subtree below  $m$ .

# Working of A\* Algorithm



Goal

$$f(N) = g(N) + h(N)$$

Actual cost from  
start node to n

estimation cost from  
start node to n

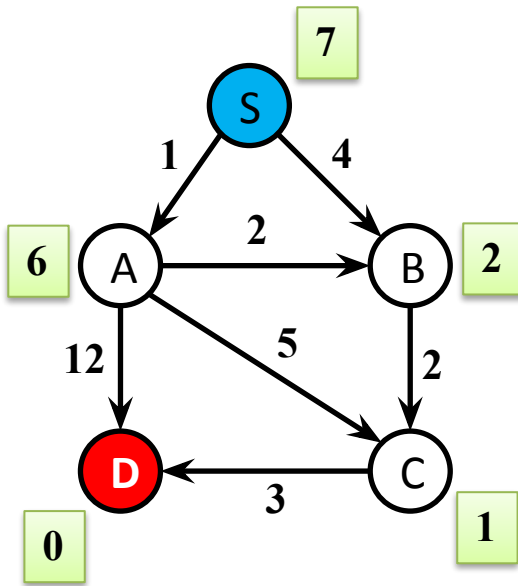
Start:	S □ B	S □ C
	4 + 12	3 + 11
	<u>16</u>	14

SC □ E	SC □ D
3+10+4	3+7+6
17	<u>16</u>

SB □ F	SB □ E
5+4+11	4+12+4
20	20

SCD □ E	SCDE □ G
3+7+2+4	3+7+2+5+0
16	<u>17</u>

# A\* Example



2.  $S \square B = 4 + 2 = 6$

3.  $S \square B \square C = 4 + 2 + 1 = 7$

4.  $S \square B \square C \square D = 4 + 2 + 3 + 0 = 9$

1.  $S \square A = 1 + 6 = 7$

5.  $S \square A \square B = 1 + 2 + 2 = 5$

6.  $S \square A \square C = 1 + 5 + 1 = 7$

7.  $S \square A \square D = 1 + 12 + 0 = 13$

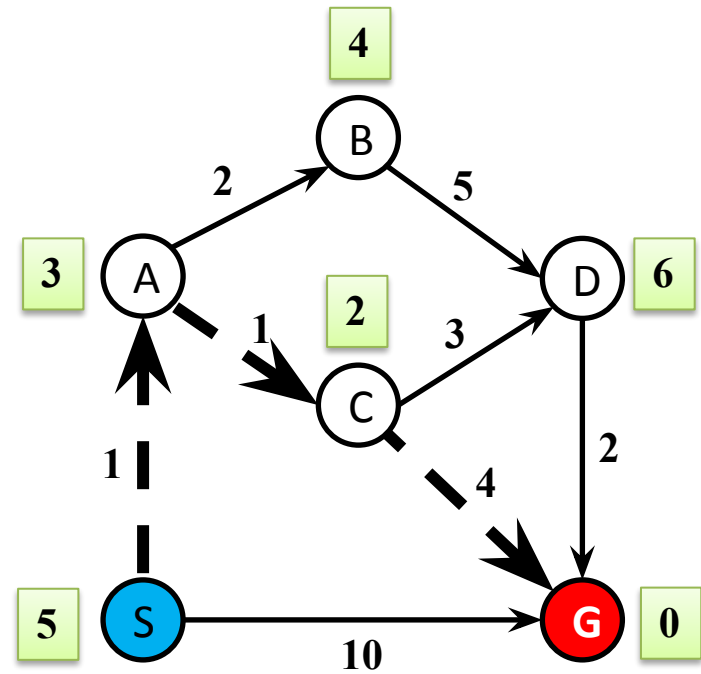
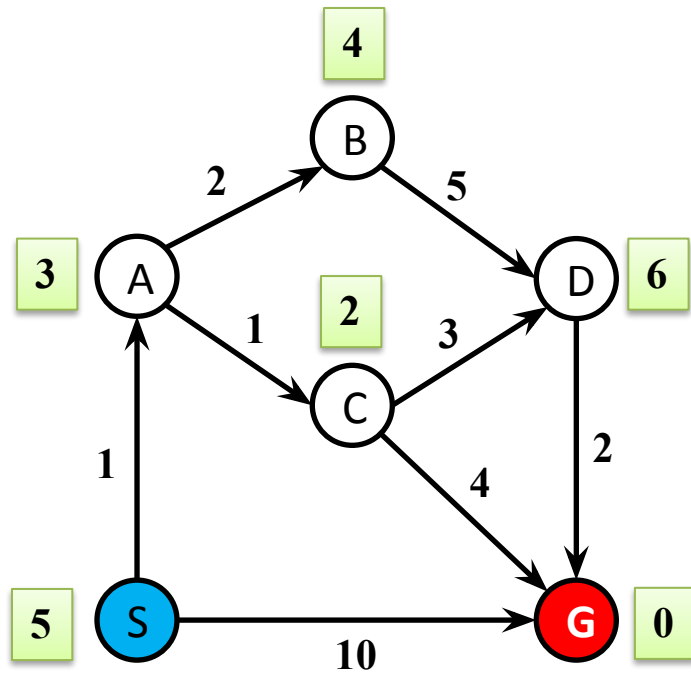
8.  $S \square A \square B \square C = 1 + 2 + 2 + 1 = 6$

9.  $S \square A \square B \square C \square D = 1 + 2 + 2 + 3 + 0 = 8$

10.  $S \square A \square C \square D = 1 + 5 + 3 + 0 = 9$

**Best Path:  $S \square A \square B \square C \square D = 1 + 2 + 2 + 3 + 0 = 8$**

## Another A\* Example



$$S \square A \square C \square G = 1 + 1 + 4 + 0 = 6$$

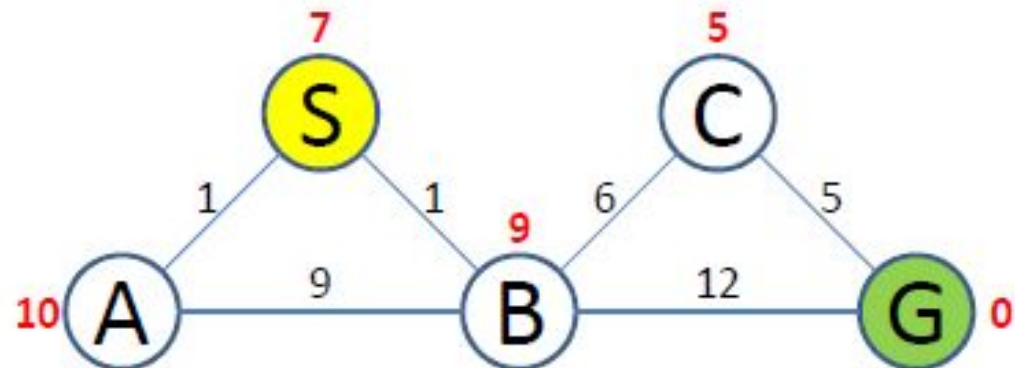
# A\* algorithm by Example

<sup>0</sup>  
<sup>7</sup> S <sup>7</sup>

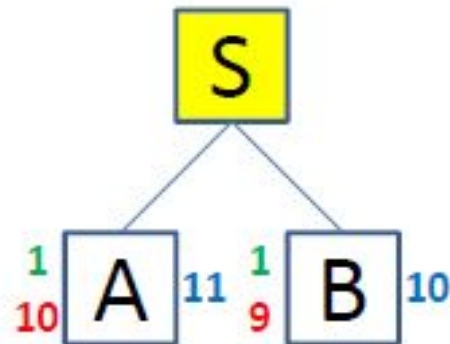
$f = \text{accumulated path cost} + \text{heuristic}$

QUEUE = path containing root

QUEUE: <S>



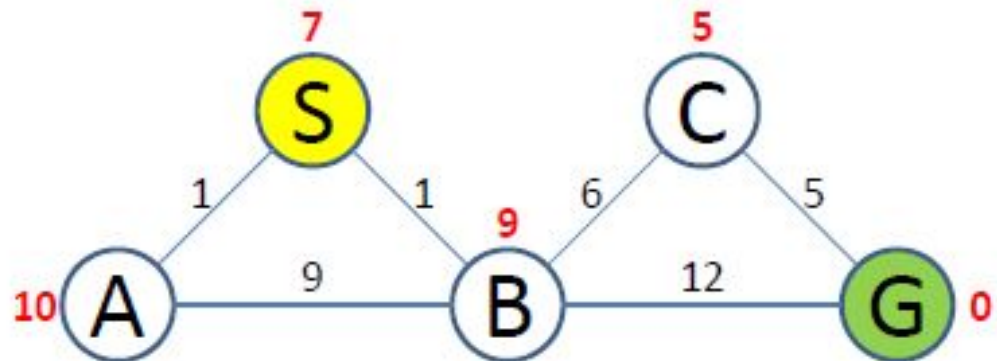
# A\* algorithm by Example



$$f = \text{accumulated path cost} + \text{heuristic}$$

Remove first path, Create paths to all children, Reject loops and Add paths.  
Sort QUEUE by f

QUEUE: <SB,SA>

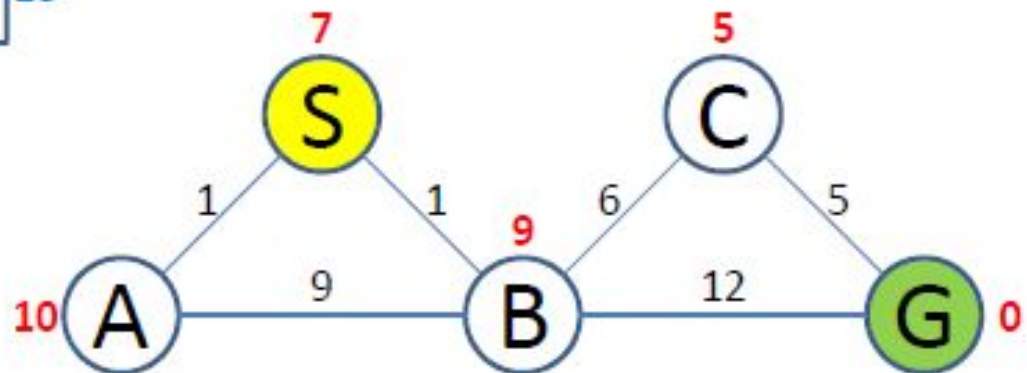
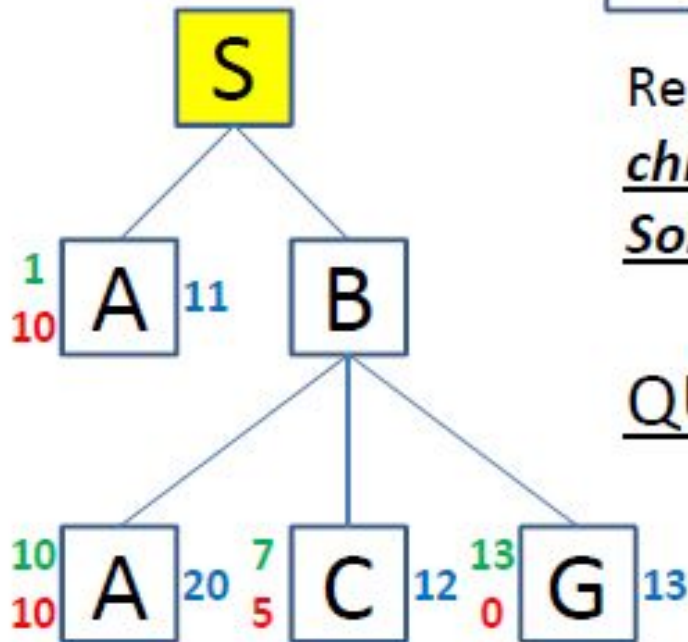


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

Remove first path, Create paths to all children, Reject loops and Add paths.  
Sort QUEUE by f

QUEUE: <SA,SBC,SBG,SBA>

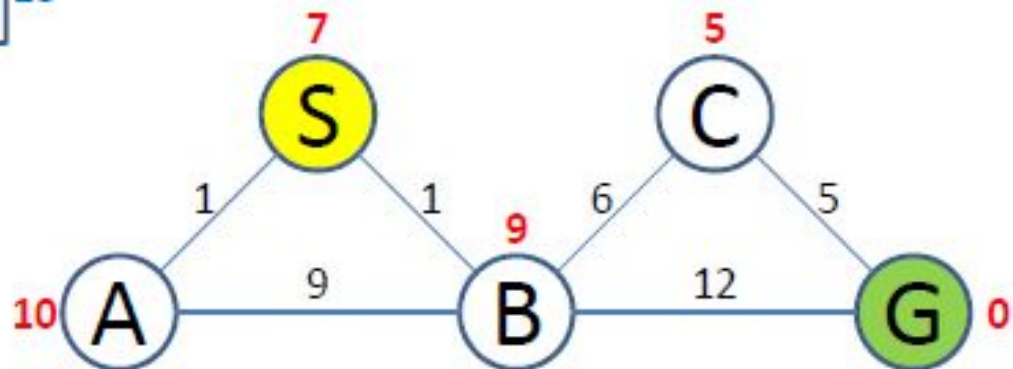
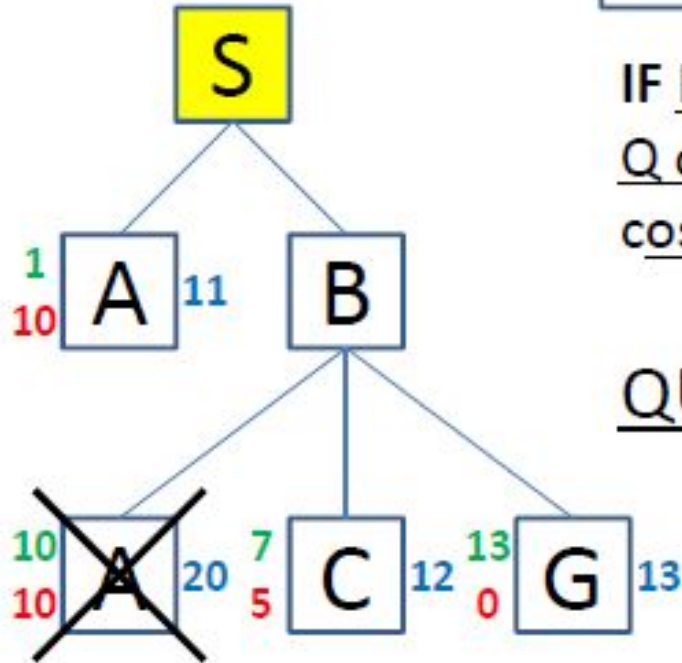


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

IF P terminating in I with cost\_P &&  
Q containing I with cost\_Q AND  
cost\_P  $\geq$  cost\_Q THEN remove P

QUEUE: <SA, SBC, SBG, **SBA**>



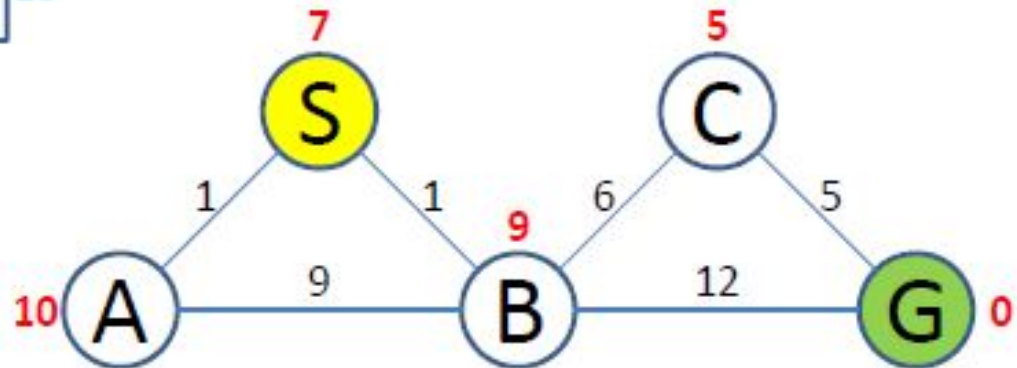
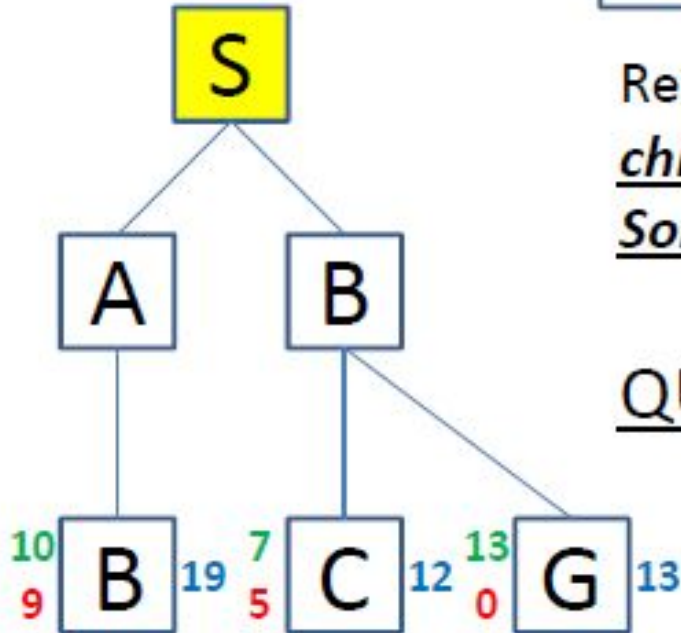


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

Remove first path, Create paths to all children, Reject loops and Add paths.  
Sort QUEUE by f

QUEUE: <SBC,SBG,SAB>

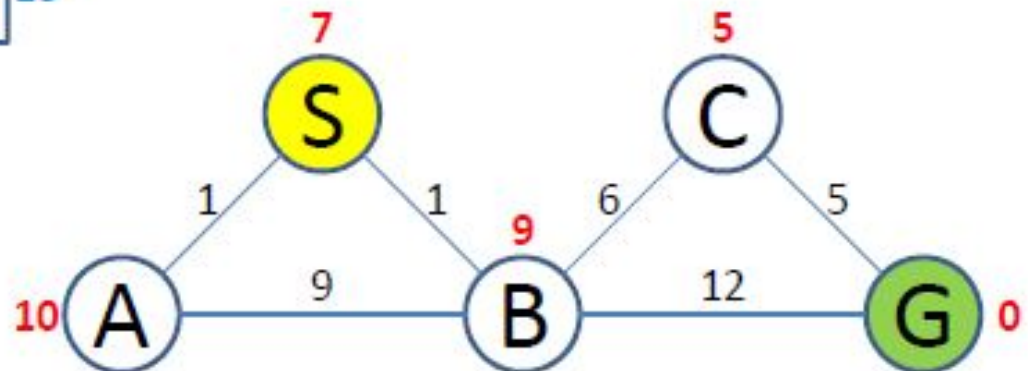
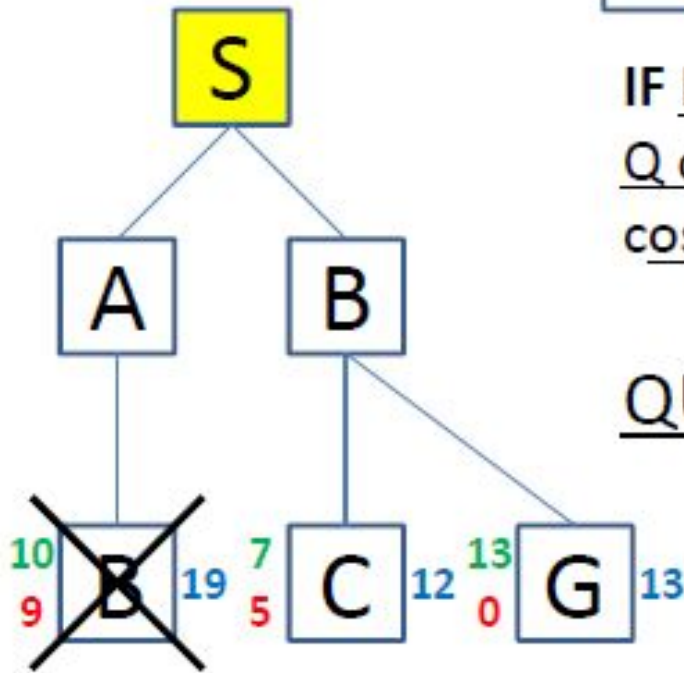


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

IF P terminating in I with cost\_P &&  
Q containing I with cost\_Q AND  
cost\_P  $\geq$  cost\_Q THEN remove P

QUEUE: <SBC, SBG, **SAB**>

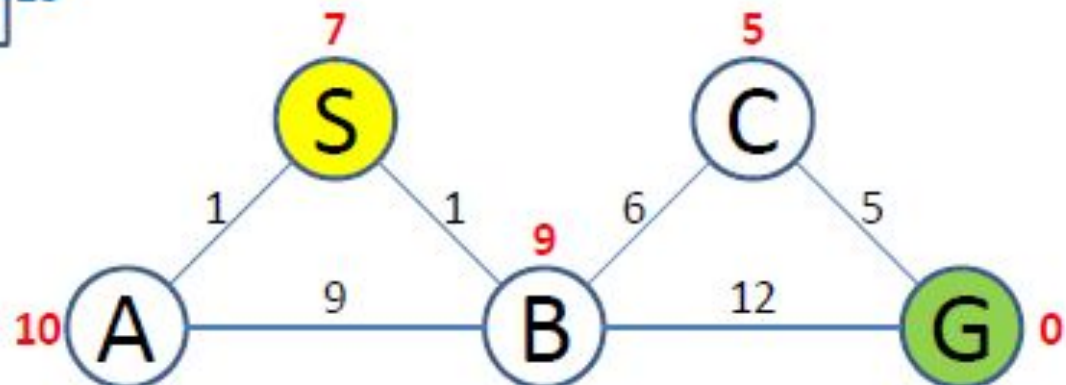
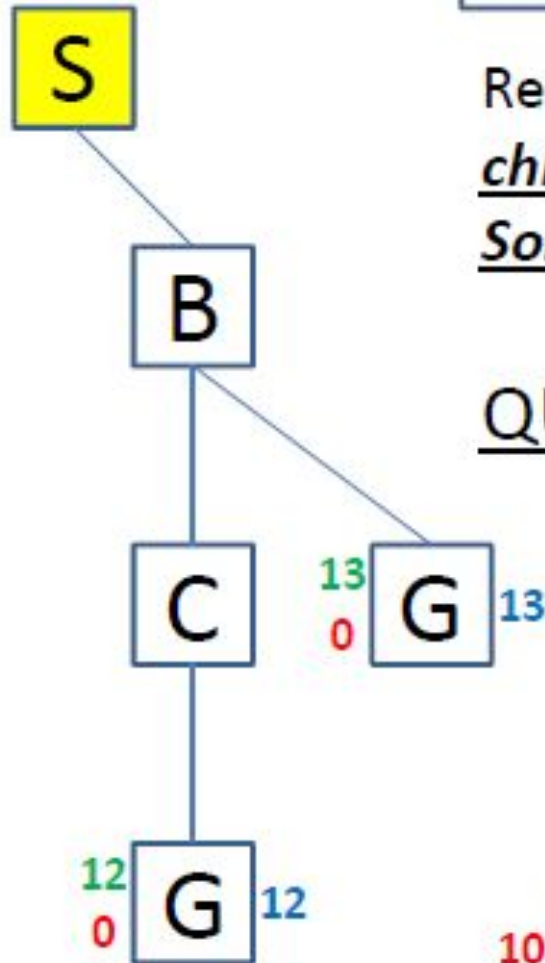


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

Remove first path, Create paths to all children, Reject loops and Add paths.  
Sort QUEUE by f

QUEUE: <SBCG, SBG>

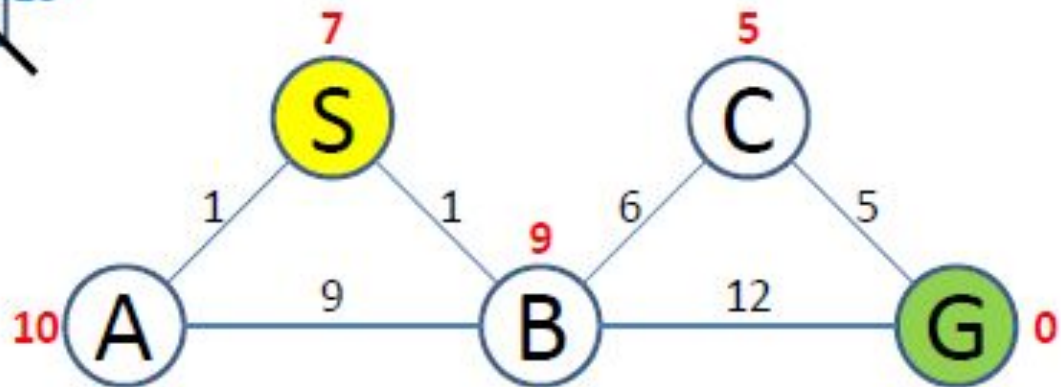
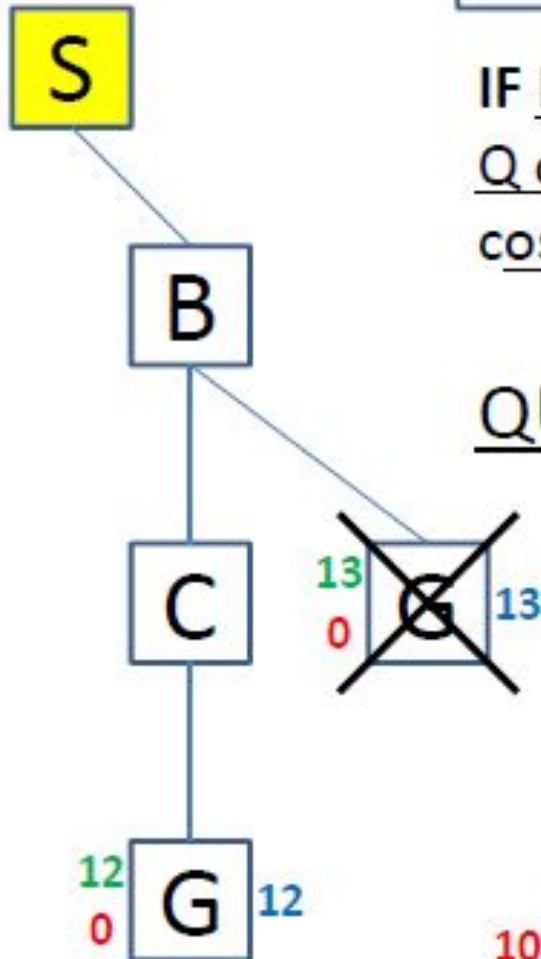


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

IF P terminating in I with cost\_P &&  
Q containing I with cost\_Q AND  
cost\_P  $\geq$  cost\_Q THEN remove P

QUEUE: <SBCG, **SBG**>



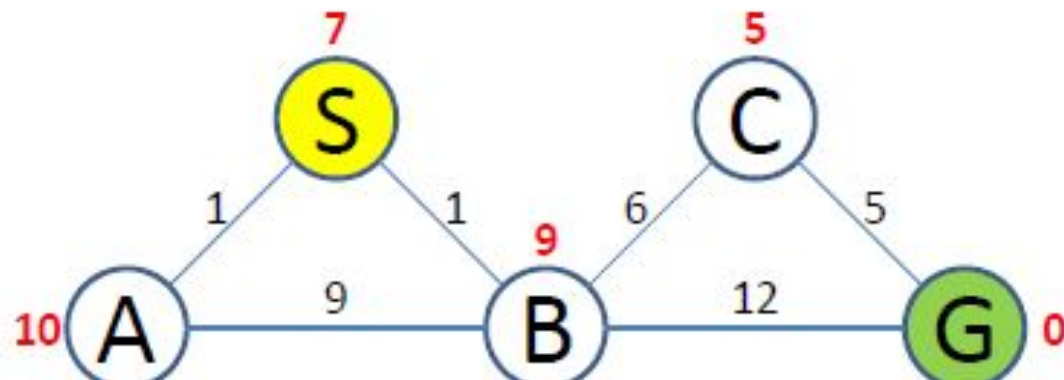
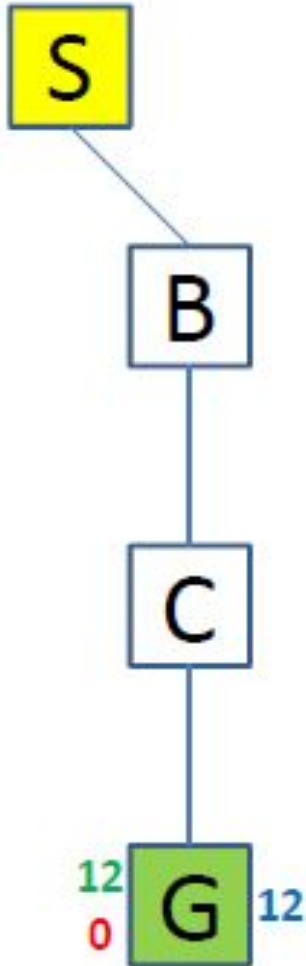


# A\* algorithm by Example

$$f = \text{accumulated path cost} + \text{heuristic}$$

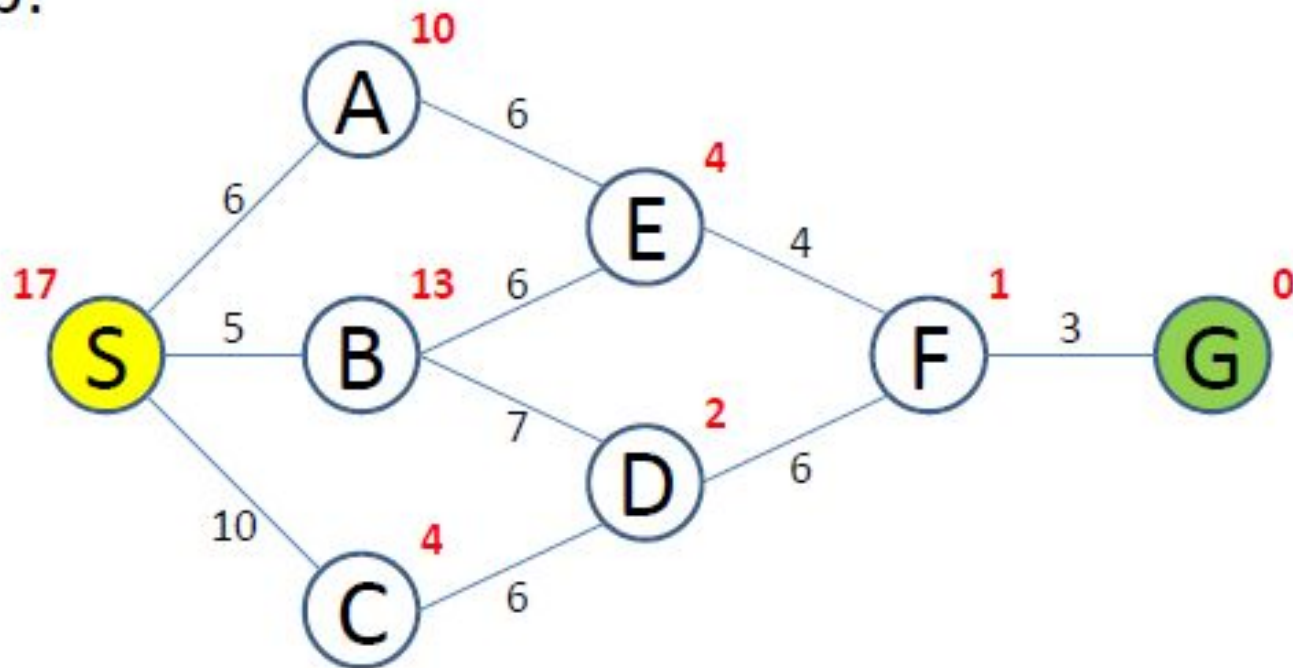
SUCCESS

QUEUE: <SBCG>

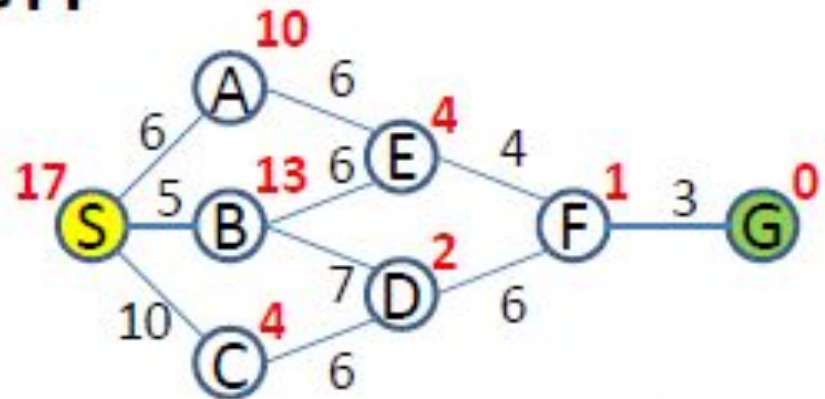


# Problem

- Perform the A\* Algorithm on the following figure. Explicitly write down the queue at each step.



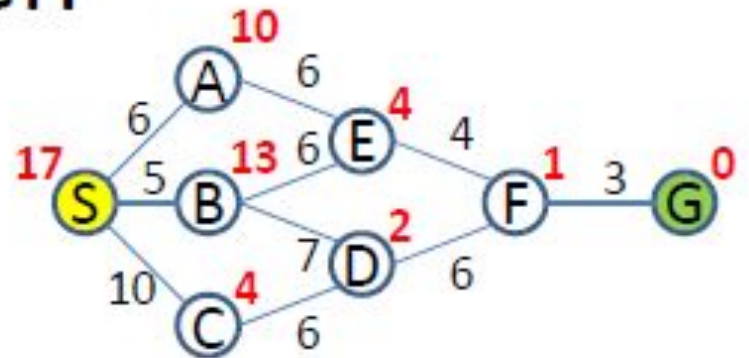
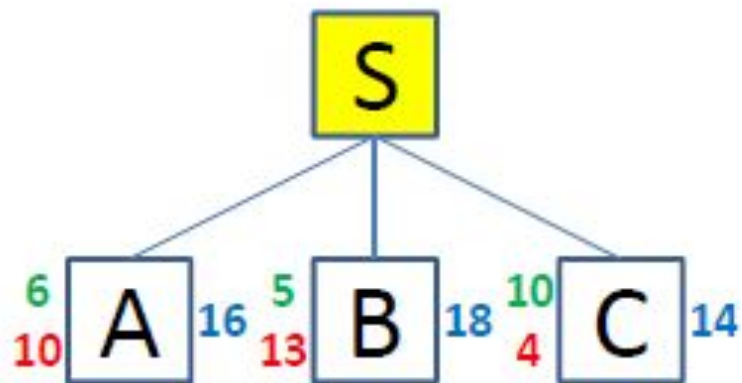
# A\* Search



QUEUE:

S

# A\* Search



QUEUE:

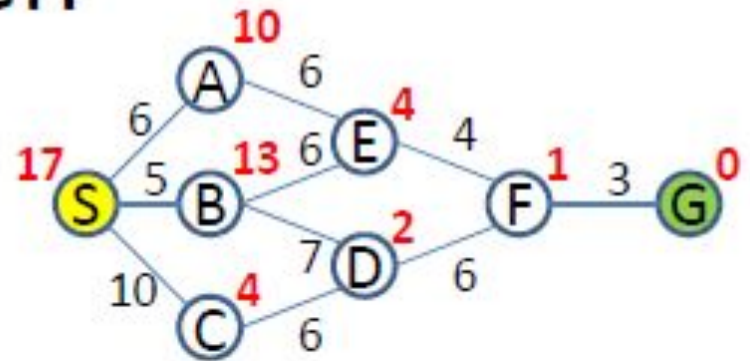
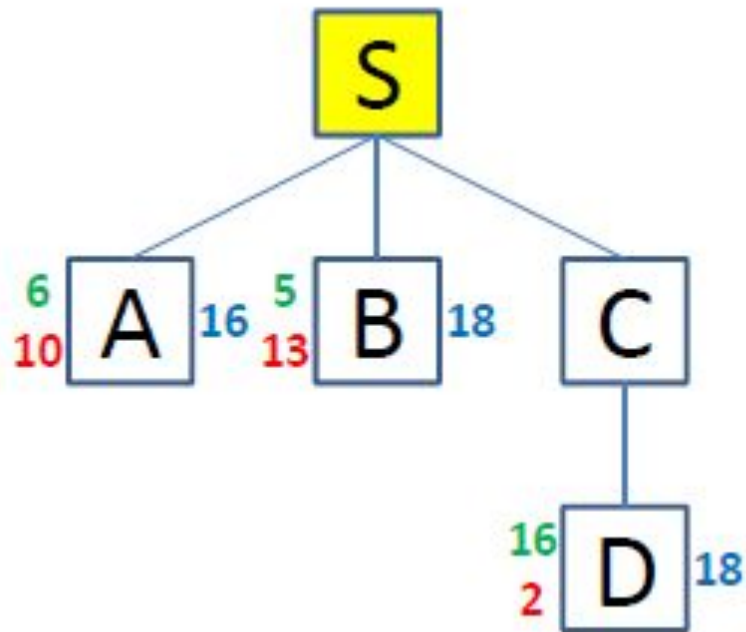
SC

SA

SB



# A\* Search



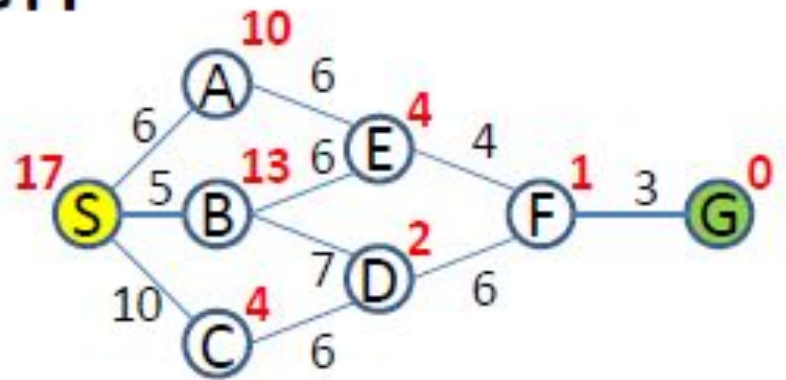
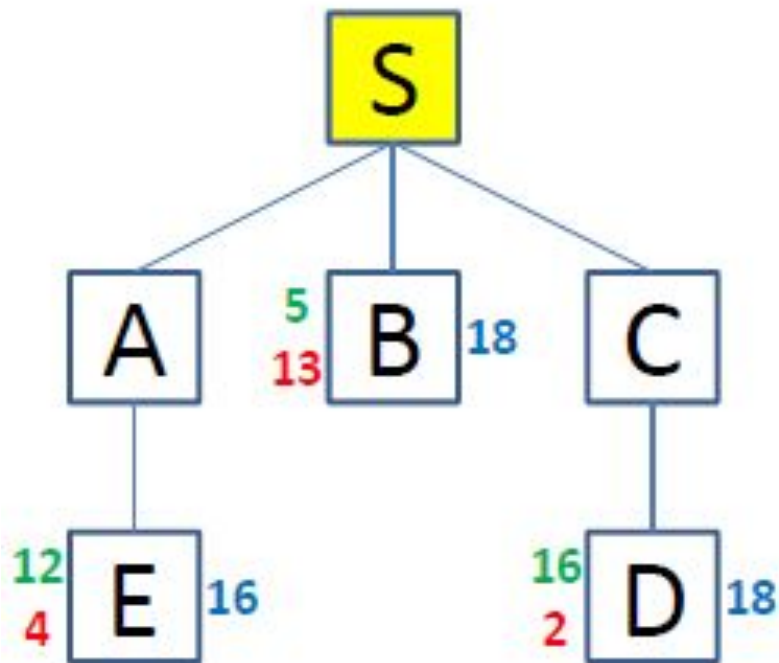
QUEUE:

SA

SCD

SB

# A\* Search



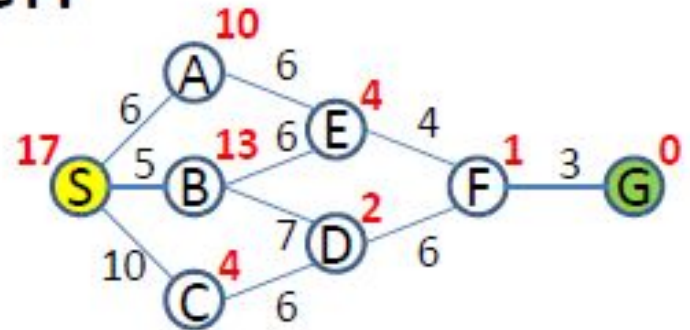
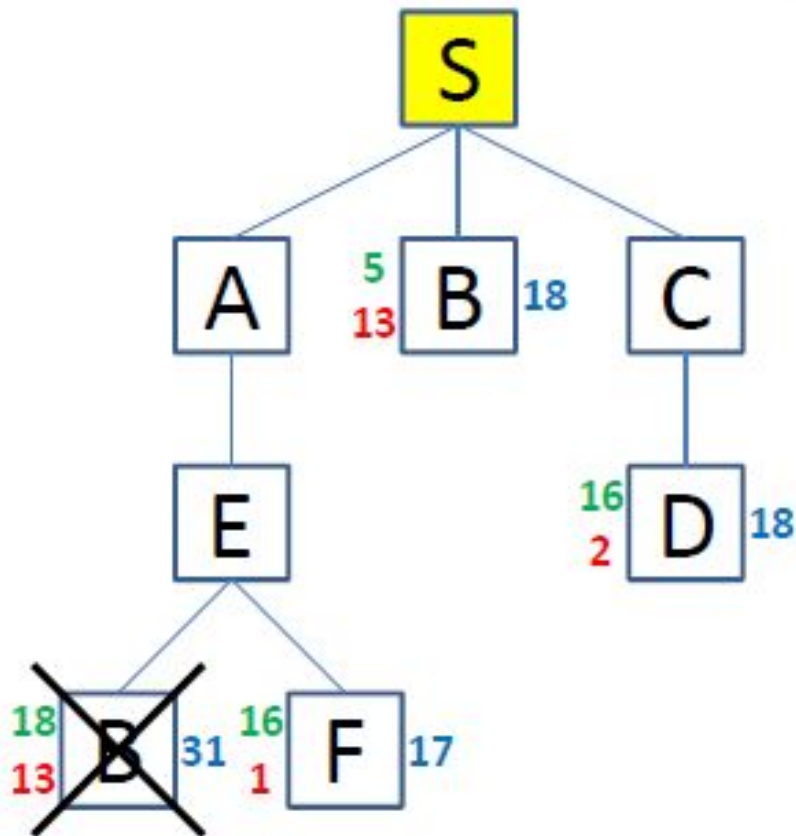
QUEUE:

SAE

SCD

SB

# A\* Search



QUEUE:

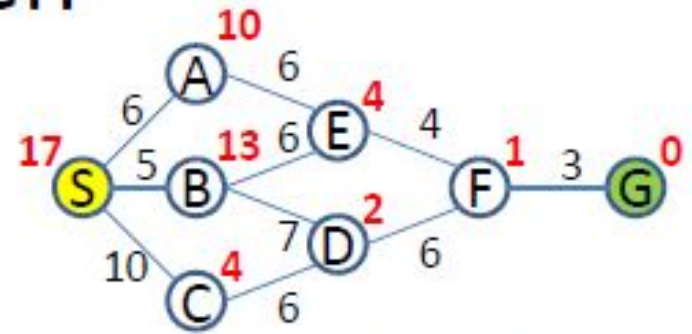
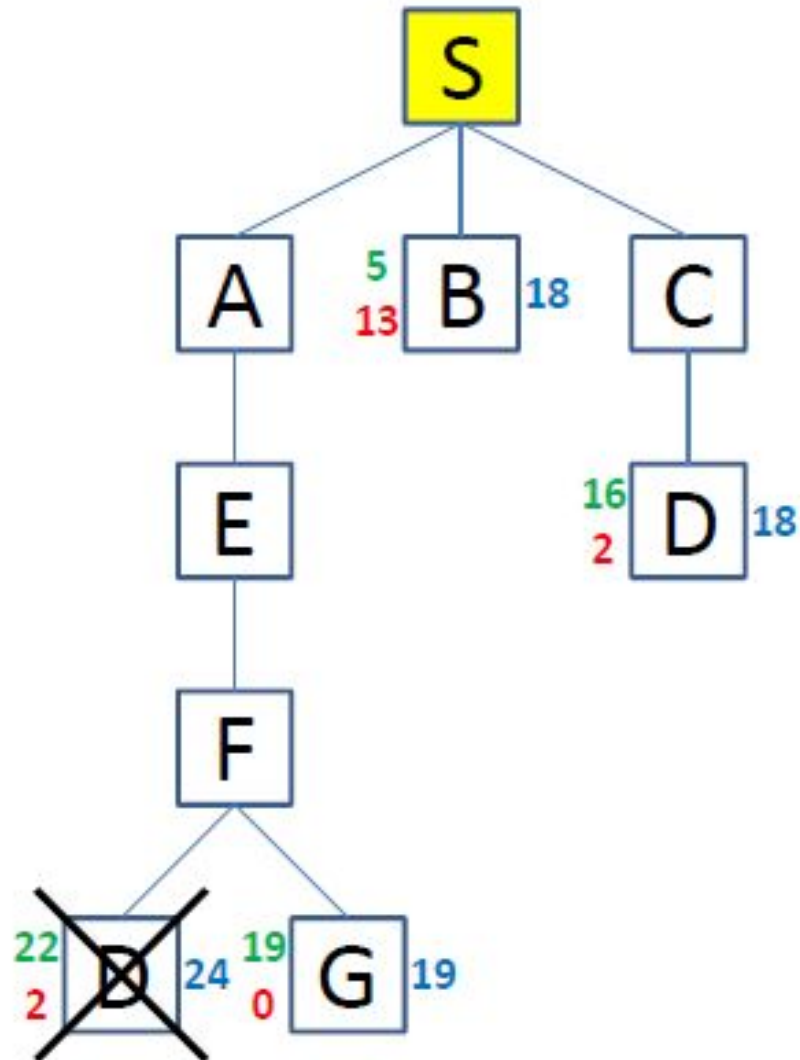
SAEF

SCD

SB

**SAEB**

# A\* Search



QUEUE:

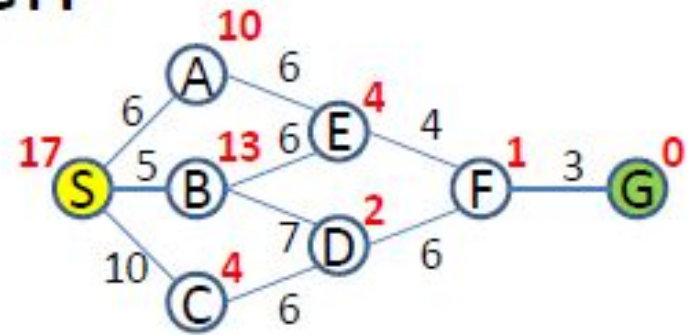
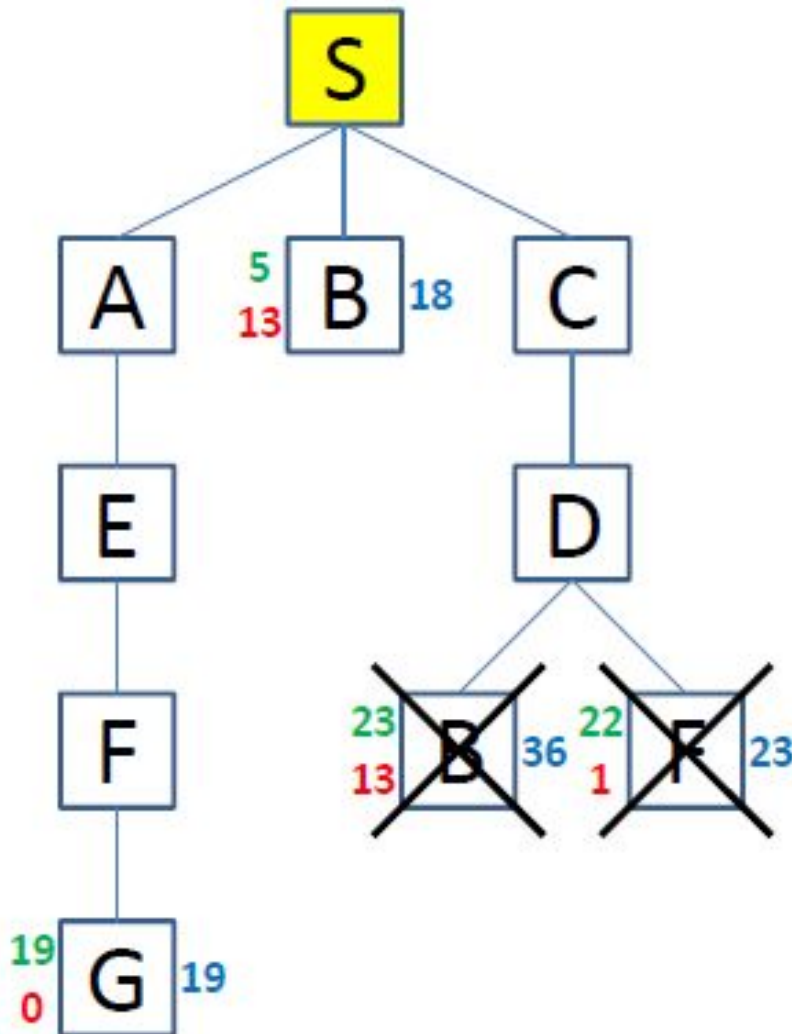
SCD

SB

SAEFG

**SAEFD**

# A\* Search



QUEUE:

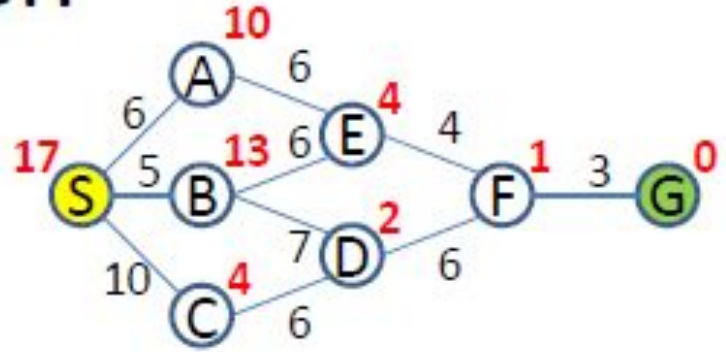
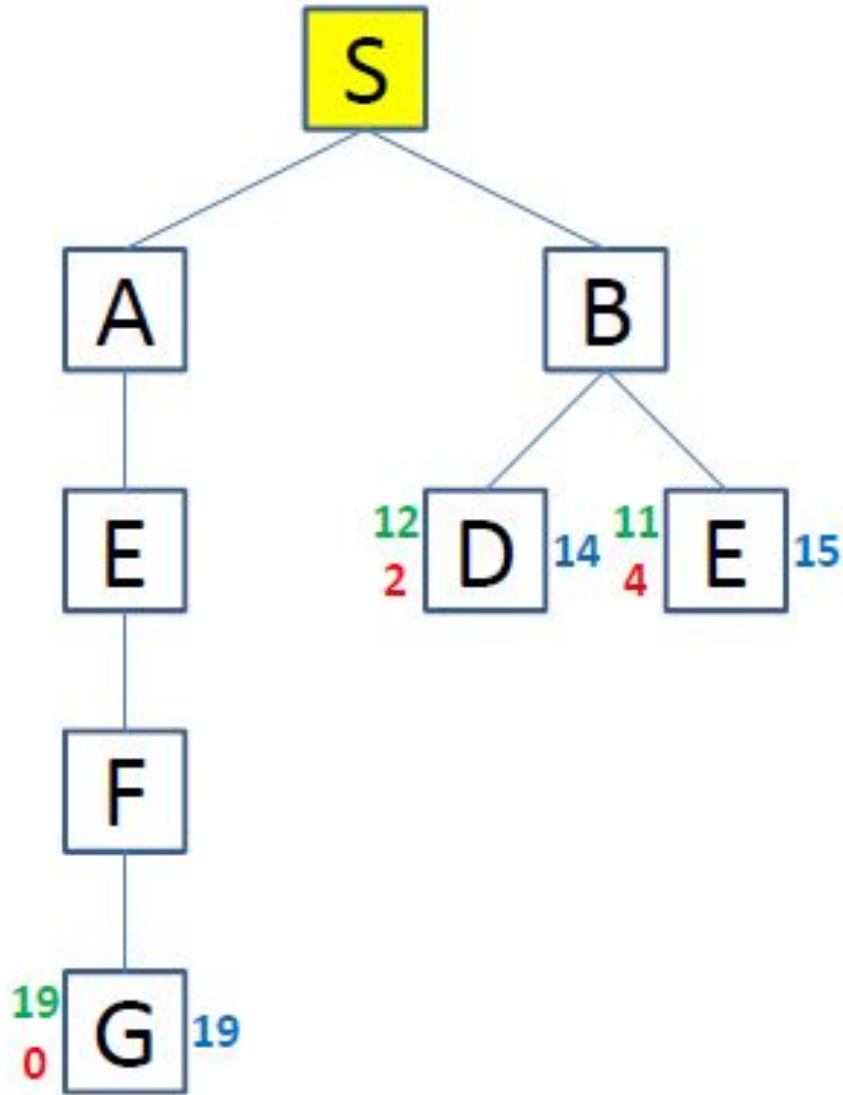
SB

SAEFG

**SCDF**

**SCDB**

# A\* Search



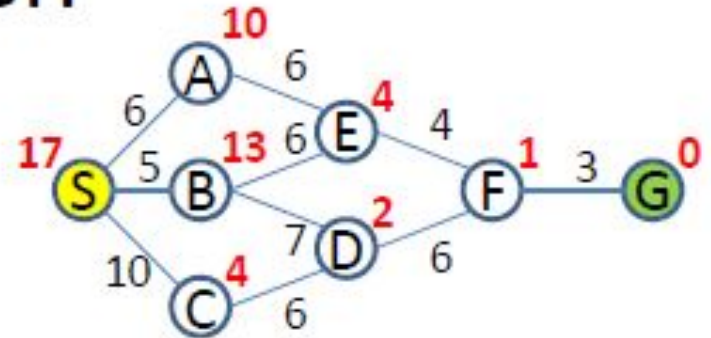
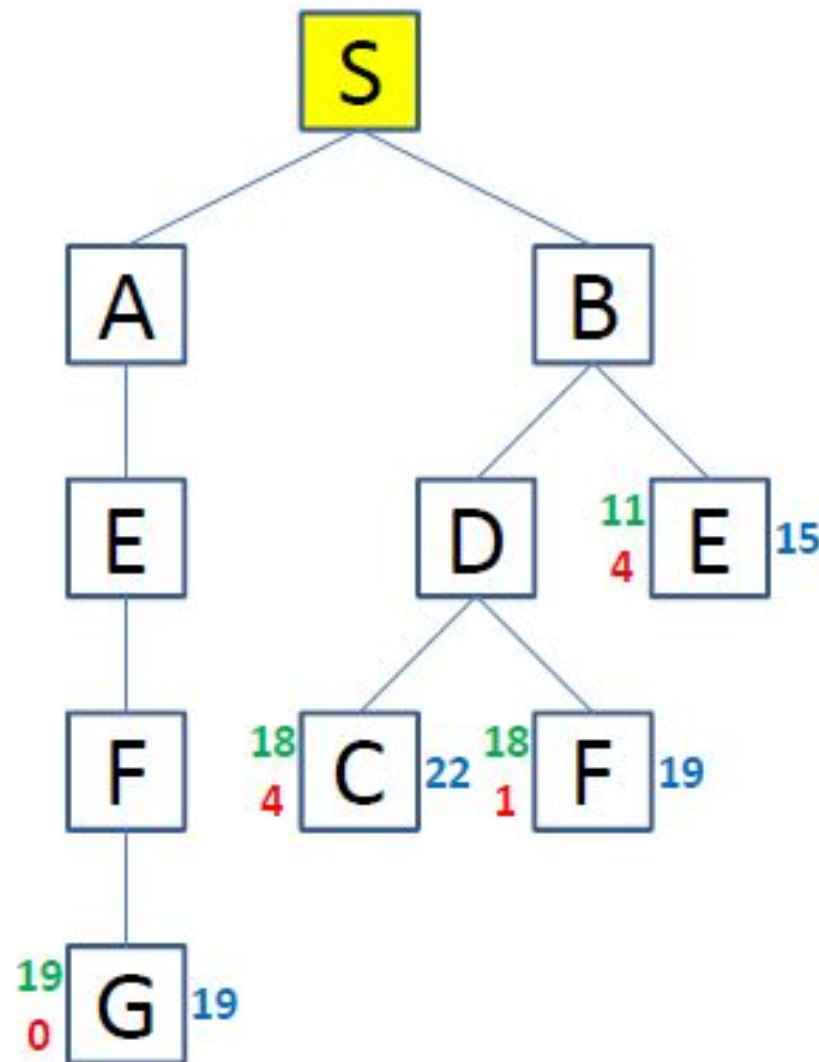
QUEUE:

SBD

SBE

SAEFG

# A\* Search



QUEUE:

SBE

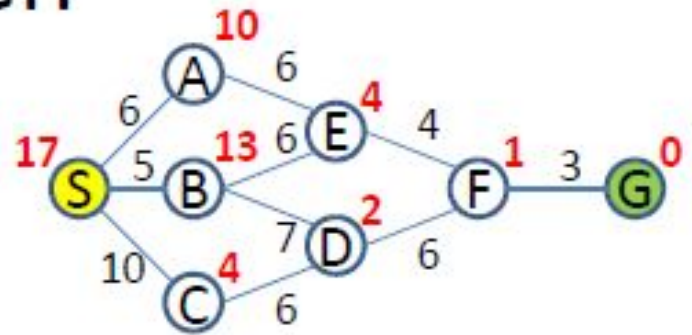
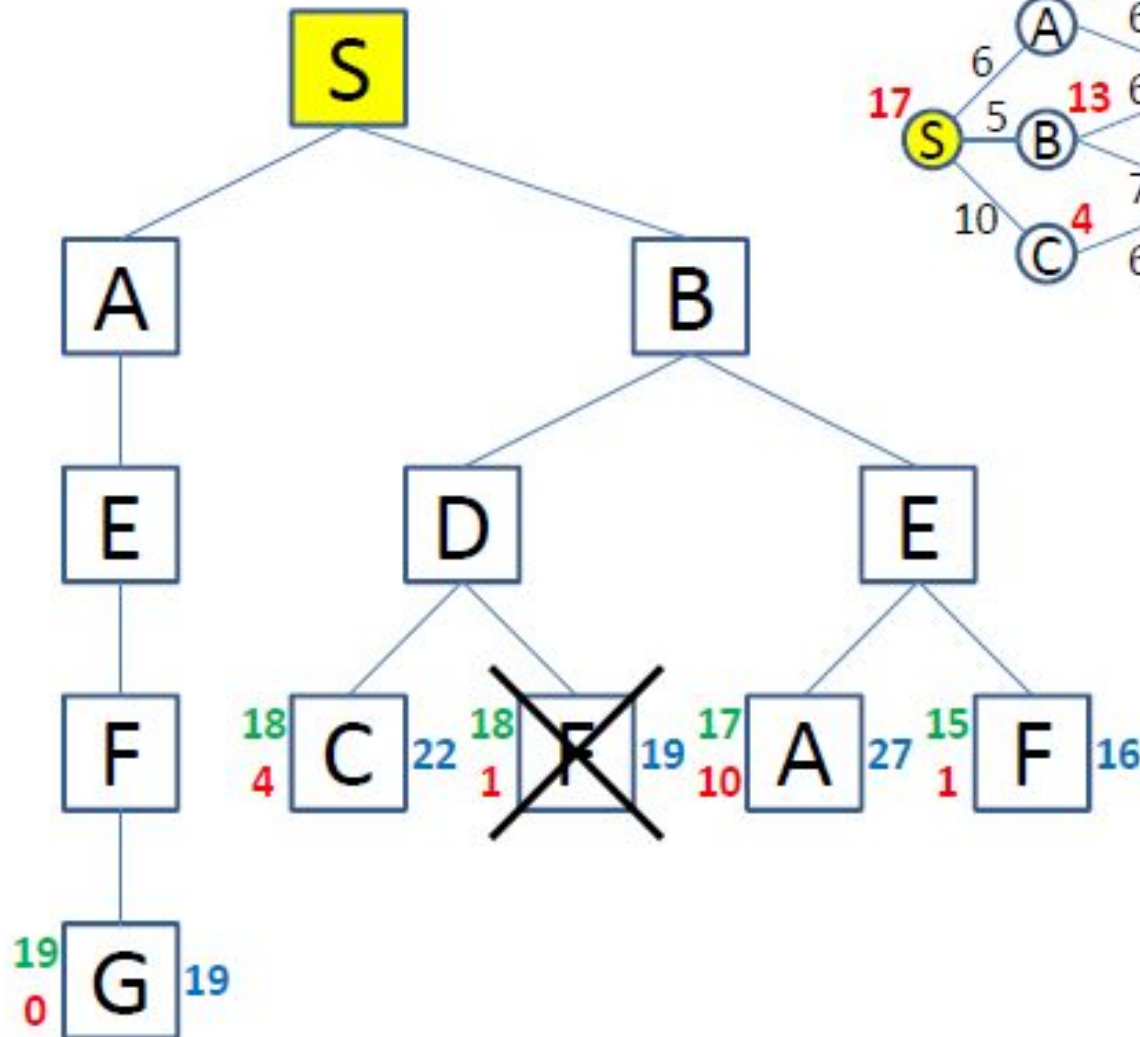
SBDF

SAEFG

SBDC



# A\* Search



QUEUE:

SBEF

SAEFG

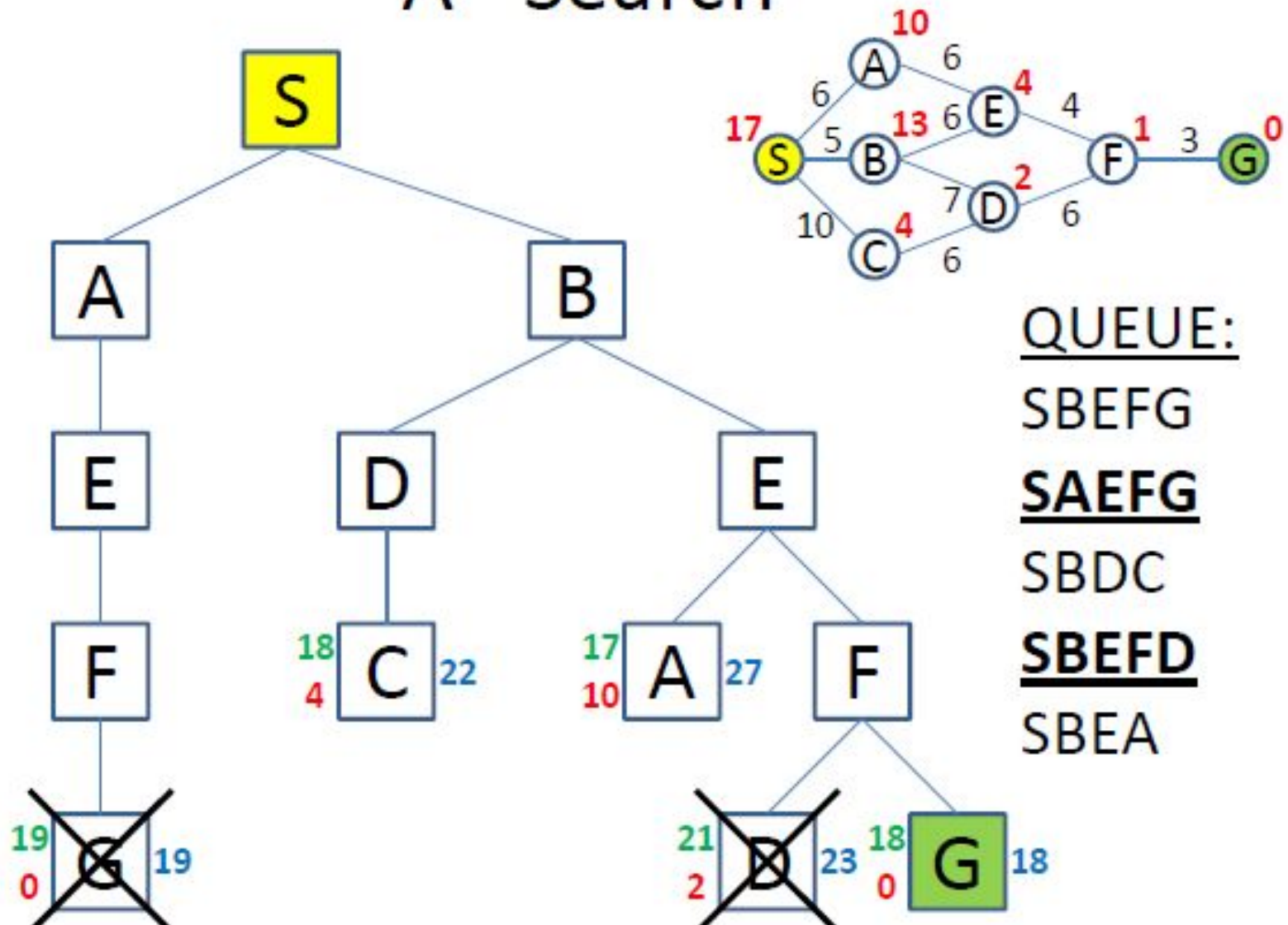
**SBDF**

SBDC

SBEA



# A\* Search



# Problem Reduction

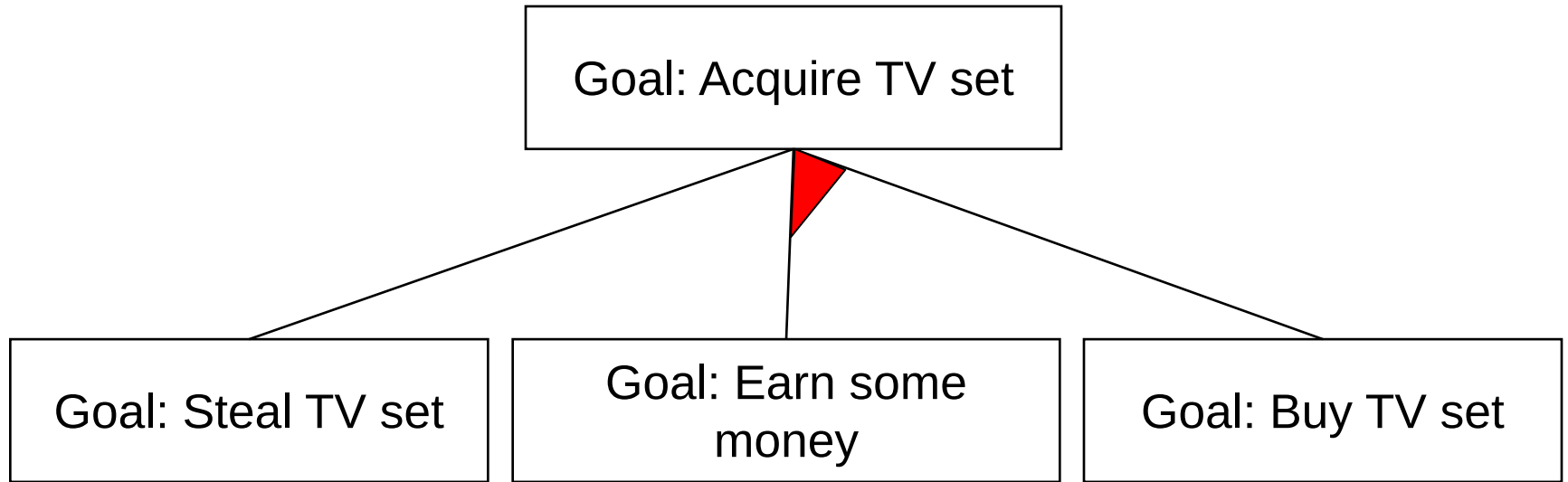
(Heuristic Search Techniques)

# Problem reduction

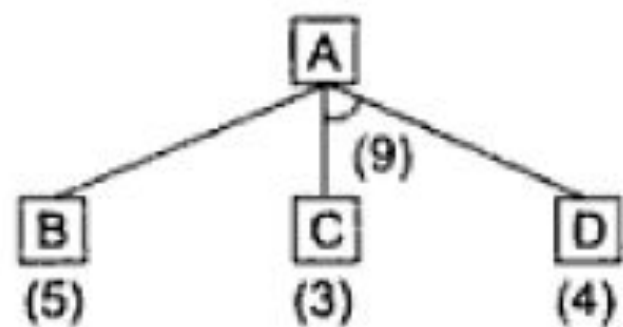
- Sometimes problems only seem hard to solve.
- A hard problem may be one that can be reduced to a number of simple problems...and, when each of the simple problems is solved, then the hard problem has been solved.
- This is the basic perception behind the method of problem reduction.

- To represent problem reduction techniques we need to use an AND-OR graph/tree.
- Problem reduction technique using AND-OR graph is useful for representing a solution of problem that can be solved by decomposing it into a set of smaller (sub)problems all of which must be solved.

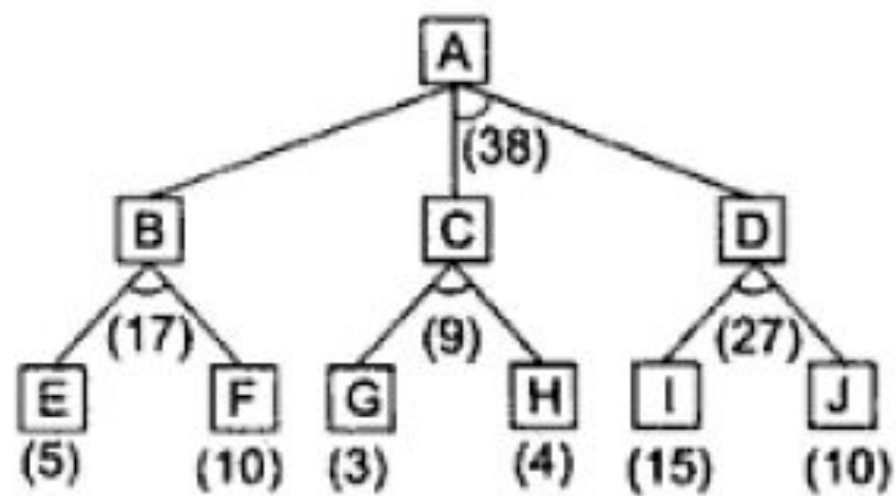
- This decomposition or reduction, generates arcs that we call AND arcs.
- One AND arc may point to a number of successor nodes, all of which must be solved in order for the arc to point to a solution.
- As in OR graph, several arcs may emerge from a single node, indicating the variety of ways in which the original problem might be solved. That is why is called AND-OR graph



## AND-OR Graphs



(a)



(b)

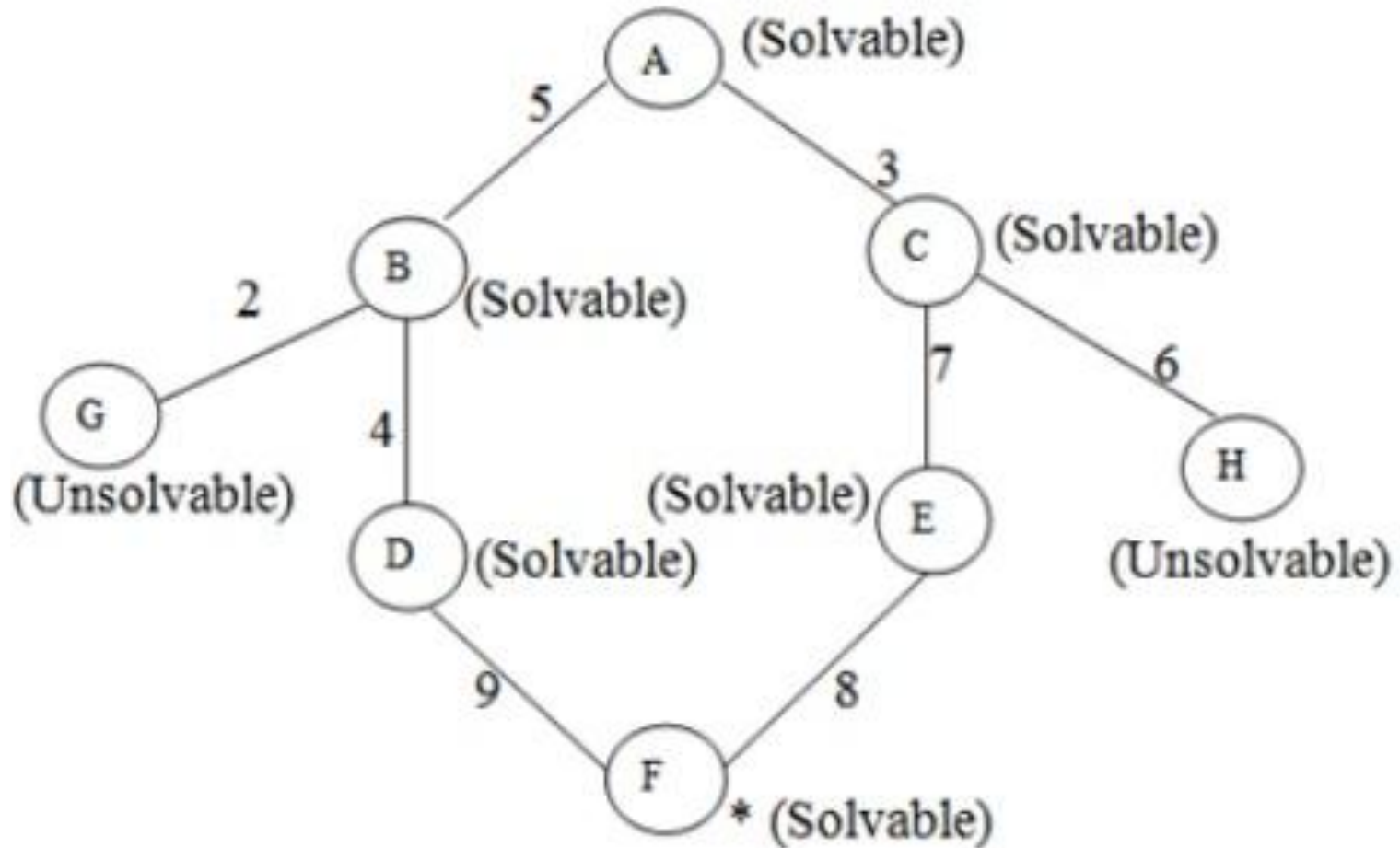
**Fig. 3.7** *AND-OR Graphs*

# Problem reduction: Algorithm

1. Initialize the graph to the starting node.
2. Loop until the starting node is labelled **SOLVED** or until its cost goes above **FUTILITY**:
  - i. Traverse the graph, starting at the initial node and following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded.
  - ii. Pick one of these unexpanded nodes and expand it. If there are no successors, assign **FUTILITY** as the value of this node. Otherwise, add its successors to the graph and for each of them compute  $f(n)$ . If  $f(n)$  of any node is  $O$ , mark that node as **SOLVED**.
  - iii. Change the  $f(n)$  estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backwards through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as **SOLVED**.



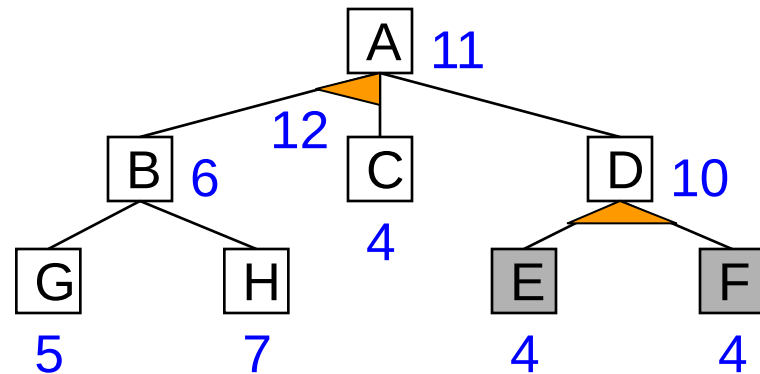
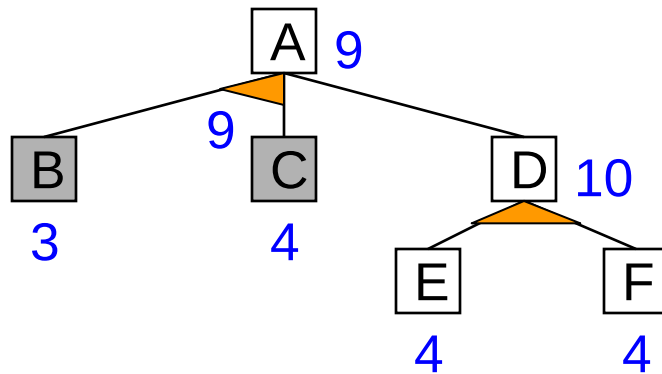
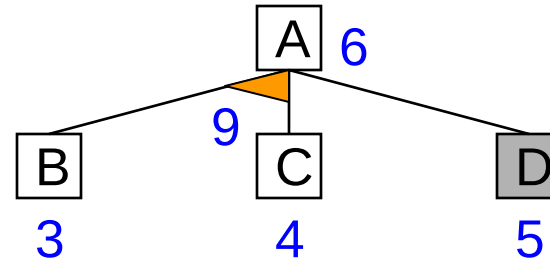
# Example: Problem Reduction



## **ALGORITHM: AO\***

1. Let  $G$  be a graph with only starting node  $INIT$ .
2. Repeat the followings until  $INIT$  is labeled SOLVED or  $h(INIT) > FUTILITY$ 
  - a) *Select an unexpanded node from the most promising path from  $INIT$  (call it  $NODE$ )*
  - b) Generate successors of  $NODE$ . If there are none, set  $h(NODE) = FUTILITY$  (i.e.,  $NODE$  is unsolvable); otherwise for each  $SUCCESSOR$  that is not an ancestor of  $NODE$  do the following:
    - i. Add  $SUCCESSOR$  to  $G$ .
    - ii. If  $SUCCESSOR$  is a terminal node, label it SOLVED and set  $h(SUCCESSOR) = 0$ .
    - iii. If  $SUCCESSOR$  is not a terminal node, compute its  $h$
  - c) Propagate the newly discovered information up the graph by doing the following: let  $S$  be set of SOLVED nodes or nodes whose  $h$  values have been changed and need to have values propagated back to their parents. Initialize  $S$  to  $Node$ . Until  $S$  is empty repeat the followings:
    - i. Remove a node from  $S$  and call it  $CURRENT$ .
    - ii. Compute the cost of each of the arcs emerging from  $CURRENT$ . Assign minimum cost of its successors as its  $h$ .
    - iii. Mark the best path out of  $CURRENT$  by marking the arc that had the minimum cost in step ii
    - iv. Mark  $CURRENT$  as SOLVED if all of the nodes connected to it through new labeled arc have been labeled SOLVED
    - v. If  $CURRENT$  has been labeled SOLVED or its cost was just changed, propagate its new cost back up through the graph. So add all of the ancestors of  $CURRENT$  to  $S$ .

# AO\* Example



# Hill Climbing

(local search algo, greedy approach, no backtrack)

(Heuristic Search Techniques)

# Hill Climbing

- Searching for a **goal state** = Climbing to the **top of a hill**
- Generate-and-test + **direction to move**.
- **Heuristic function** to estimate how close a given state is to a goal state.
- Types:
  - *Simple Hill Climbing*
  - *Steepest-Ascent Hill Climbing*
  - *Simulated Annealing*

# (1) Simple Hill Climbing :

## Algorithm :

1. Evaluate the initial state. If it is a goal state, then return it and quit it. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or there are no new operators left to be applied in the current state:
  - (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
  - (b) Evaluate the new state:
    - (i) **If it is a goal state, then return it and quit.**
    - (ii) **If it is not a goal state but it is better than the current state, then make it the current state.**
    - (iii) **If it is not better than the current state, then continue in the loop.**

## (2) Steepest-Ascent Hill Climbing (Gradient Search)

- Considers **all the moves** from the current state.
- Selects **the best one** as the next state.

### Simple vs. steepest-ascent hill climbing

- simple hill climbing is an algorithm that helps to climb a mountain in 2D Space.
- In Steepest ascent hill climbing, you can conceptualize that the mountain is in 3D space.

## Algorithm

1. Evaluate the initial state. If it is a goal state, then return it and quit it. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or a complete iteration produces no change to current state:
  - SUCC = a state such that any possible successor of the current state will be better than SUCC (the worst state).
  - For each operator that applies to the current state, evaluate the new state:
    - goal  $\rightarrow$  quit
    - better than SUCC  $\rightarrow$  set SUCC to this state
  - SUCC is better than the current state  $\rightarrow$  set the current state to SUCC.



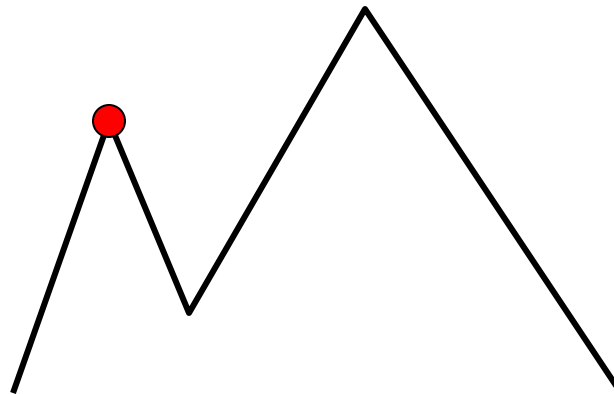
- **Both basic and steepest-ascent hill climbing**
  - May fail to find a solution.
  - Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated.
  - This will happen if the program has reached either a local maximum, a plateau, or a ridge.
- **Both Hill climbing and Steepest ascent hill climbing suffers**
  - the limitation of getting stuck in local maximum which can be addressed by simulated annealing.

- A *local maximum* is a state that is better than all its neighbors but is not better than some state farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called *foothills*.
- A *plateau* is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.
- A *ridge* is a special kind of local maximum. It is an area of the search space that is higher than surroundings areas and that itself has a slope.

# Hill Climbing: Disadvantages

## Local maximum

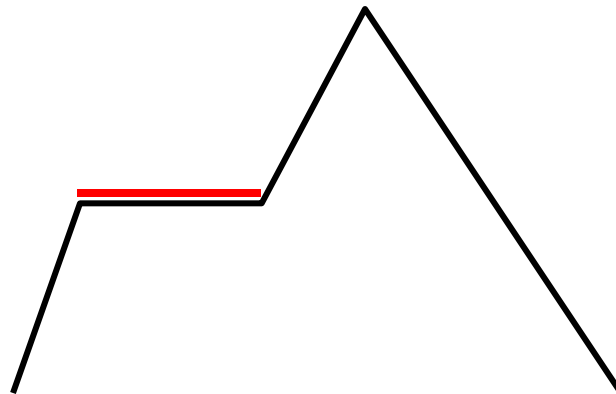
A state that is better than all of its neighbours, but not better than some other states far away.



# Hill Climbing: Disadvantages

## Plateau

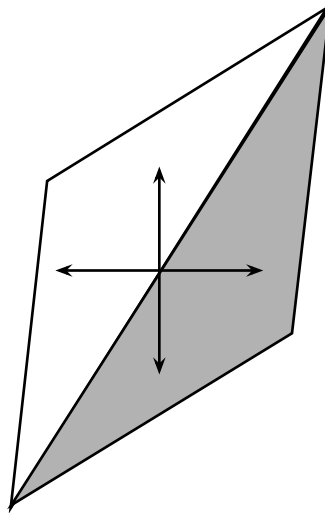
A flat area of the search space in which all neighbouring states have the same value.



# Hill Climbing: Disadvantages

## Ridge

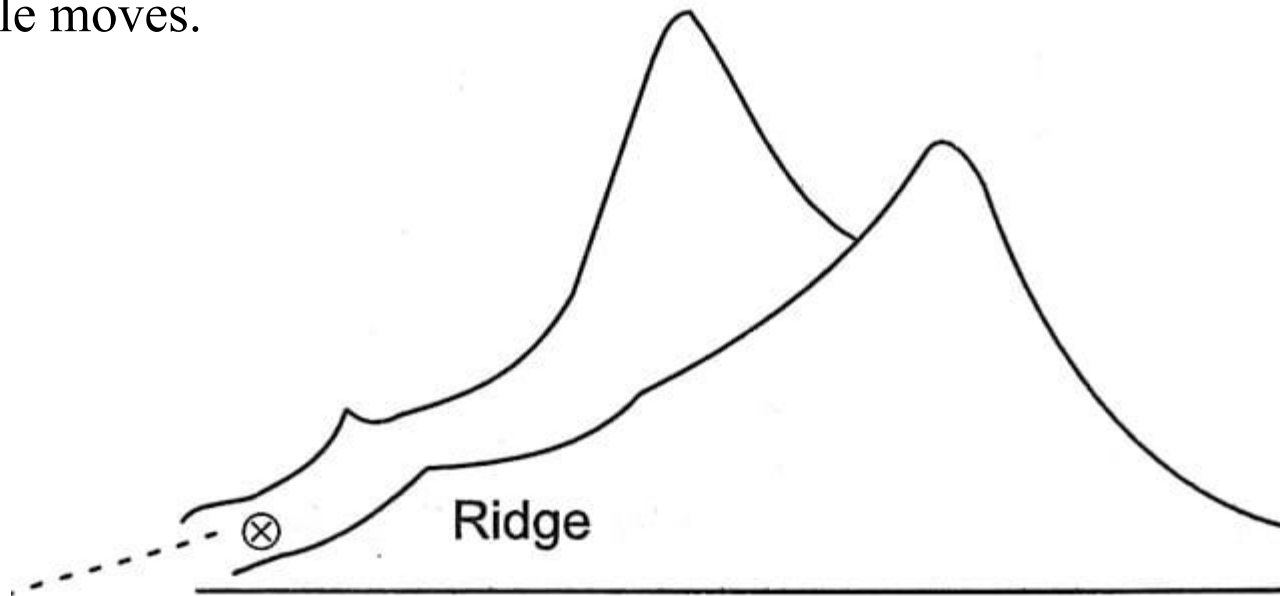
The orientation of the high region, compared to the set of available moves, makes it impossible to climb up. However, two moves executed serially may increase the height.





- (a) Local Maximum
- (b) Plateau
- (c) Ridge

- **Local maximum:** It is a state that is better than all its neighbors but is not better than some other states farther away .
- **Plateau:** It is a flat area of the search space in which a whole set of neighboring states have the same value. on a plateau it is not possible to determine the best direction in which to move by making local comparisons.
- **Ridge:** It is special case of local maximum. It is an are a of the search space that is higher than surrounding areas and that itself has a slope. But the orientation of the high region, compared to the set of available moves and directions in which they move, makes it impossible to traverse a ridge by single moves.



# Solution to problems

**Local Maxima :** Backtrack to some earlier node and try going in a different direction.

**Plateaus:** Make a big jump to try to get in a new section of search space.

**Ridges:** Moving in several directions at once (apply two or more rules before doing test).



### (3) Simulated Annealing :

- Simulated Annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.

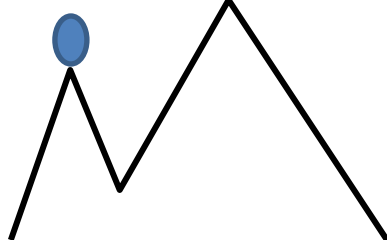
## Physical Annealing

- Physical substances are melted and then **gradually cooled** until some solid state is reached.
- The goal is to produce a **minimal-energy** state.
- **Annealing schedule**: if the temperature is lowered sufficiently slowly, then the goal will be attained.
- Nevertheless, there is some probability for a transition to a higher energy state:  $e^{-\Delta E/T}$ .

# Algorithm: Simulated Annealing

1. Energy  $E$ , Change in Energy  $\Delta E$   
Temperature  $T$

Local Maxima/Minima



2.

3. Current Configuration to New Configuration

$C$  ☐  $N$  <sup>Move</sup>

$C = C_{\text{init}}$   
for  $T = T_{\text{max}}$  to  $T_{\text{min}}$   
 $E_C = E(C)$   
 $N = \text{Next}(C)$   
 $E_N = E(N)$   
 $\Delta E = E_N - E_C$

if  $(\Delta E > 0)$  (positive – good)  
 $C = N$   
elseif  $(E^{(\Delta E/T)} > \text{rand}(0,1))$   
 $C = N$

**Probability Function:**  $E^{(\Delta E/T)}$

For high temperature we accept **more bad** moves,  
as we progress, with low temperature, the probability to accept bad move is also **low**

## Algorithm: Simulated Annealing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize BEST-SO-FAR to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
  - (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state. Compute

$$\Delta E = \text{Val}(\text{current state}) - \text{Val}(\text{new state})$$

- If the new state is a goal state then return it and quit.
- If it is not a goal state but it is better than the current state then make it the current state. Also set BEST-SO-FAR to this new state.
- If it is not better than the current state, then make it the current state with probability  $p'$ . This step is usually implemented by invoking a random number generator to produce a number in the range  $[0,1]$ . If the number is less than  $p'$ , then the move is accepted. Otherwise do nothing.

(c) Revise  $T$  as necessary according to the annealing schedule.

(5) Return BEST-SO-FAR, as the answer.

# Ways Out

- **Backtrack** to some earlier node and try going in a different direction.
- Make a **big jump** to try to get in a new section.
- Moving in **several directions** at once.

# Hill Climbing: Disadvantages

- Hill climbing is a **local method**:  
Decides what to do next by looking only at the “immediate” consequences of its choices.
- **Global information** might be encoded in heuristic functions.





# Hill Climbing: Conclusion

- Can be **very inefficient** in a large, rough problem space.
- Global heuristic may have to pay for **computational complexity**.
- **Often useful** when combined with other methods, getting it started right in the right general neighbourhood.



# 4-Queen Problem using Hill Climbing

Step 0

	A	B	C	D
0				
1				
2				
3				

Score -6

Step 1

Step 2

Step 3

# Local Beam Search

- Beam search is a heuristic search algorithm that explores a graph by expanding the most optimistic node in a limited set. Beam search is an *optimization* of best-first search that reduces its memory requirements.
- Best-first search is a graph search that orders all partial solutions according to some heuristic. But in beam search, only a predetermined number of best partial solutions are kept as candidates. Therefore, it is a *greedy* algorithm.
- Beam search uses *breadth-first search* to build its search tree. At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost. However, it only stores a predetermined number ( $\beta$ ), of best states at each level called the *beamwidth*. Only those states are expanded next.

- The greater the beam width, the fewer states are pruned. No states are pruned with infinite beam width, and beam search is identical to breadth-first search. The beamwidth bounds the memory required to perform the search.
- Since a *goal state* could potentially be pruned, beam search sacrifices completeness (the guarantee that an algorithm will terminate with a solution if one exists). Beam search is not optimal, which means there is no guarantee that it will find the best solution.
- In general, beam search returns the *first* solution found. Once reaching the configured maximum search depth (i.e., translation length), the algorithm will evaluate the solutions found during a search at various depths and return the best one that has the highest probability.
- The beam width can either be *fixed* or *variable*. One approach that uses a variable beam width starts with the width at a minimum. If no solution is found, the beam is widened, and the procedure is repeated.

# Example: Beam Search

- For example, let's take the value of  $\beta = 2$  for the tree shown below. So, follow the following steps to find the goal node.

**Step 1:** OPEN = {A}

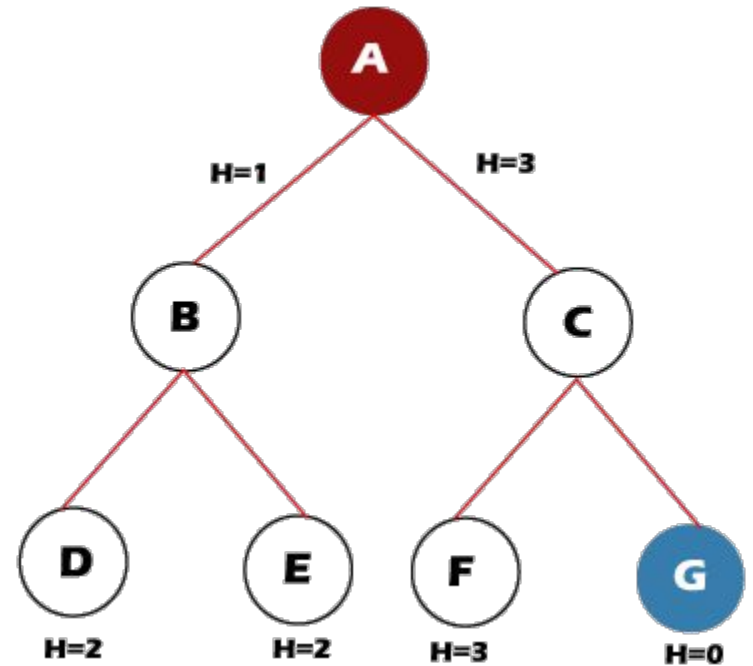
**Step 2:** OPEN = {B, C}

**Step 3:** OPEN = {D, E}

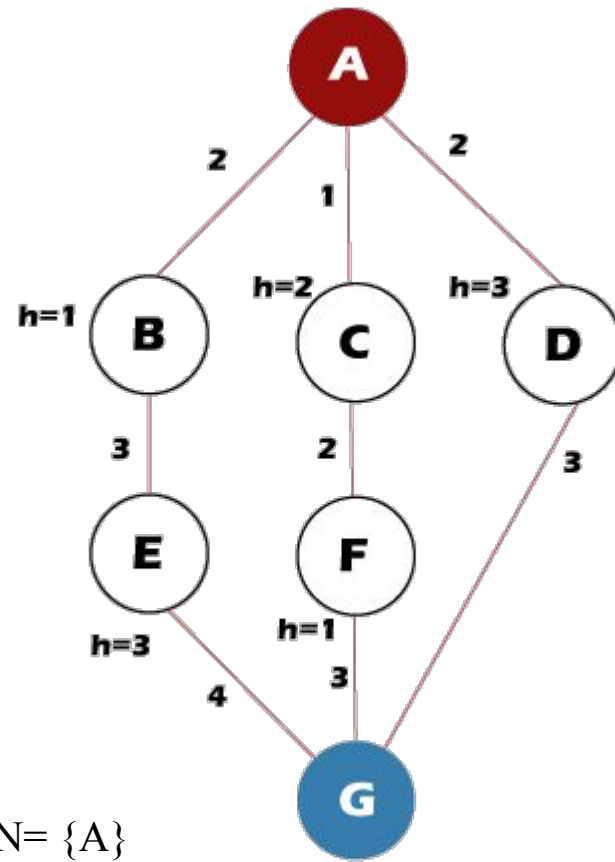
**Step 4:** OPEN = {E}

**Step 5:** OPEN = { }

The open set becomes empty **without** finding the goal node.



- The beam width and an inaccurate heuristic function may cause the algorithm to *miss* expanding the shortest path.
- A more *precise* heuristic function and a *larger* beam width can make Beam Search more likely to find the optimal path to the goal.
- For example, we have a tree with heuristic values as shown here:



Step 1: OPEN= {A}  
 Step 2: OPEN= {B, C}  
 Step 3: OPEN= {C, E}  
 Step 4: OPEN= {F, E}  
 Step 5: OPEN= {G, E}  
 Step 6: Found the goal node {G}, now stop.  
**Path: A-> C-> F-> G**

But the *Optimal Path* is: A-> D-> G