Informed Search Strategies

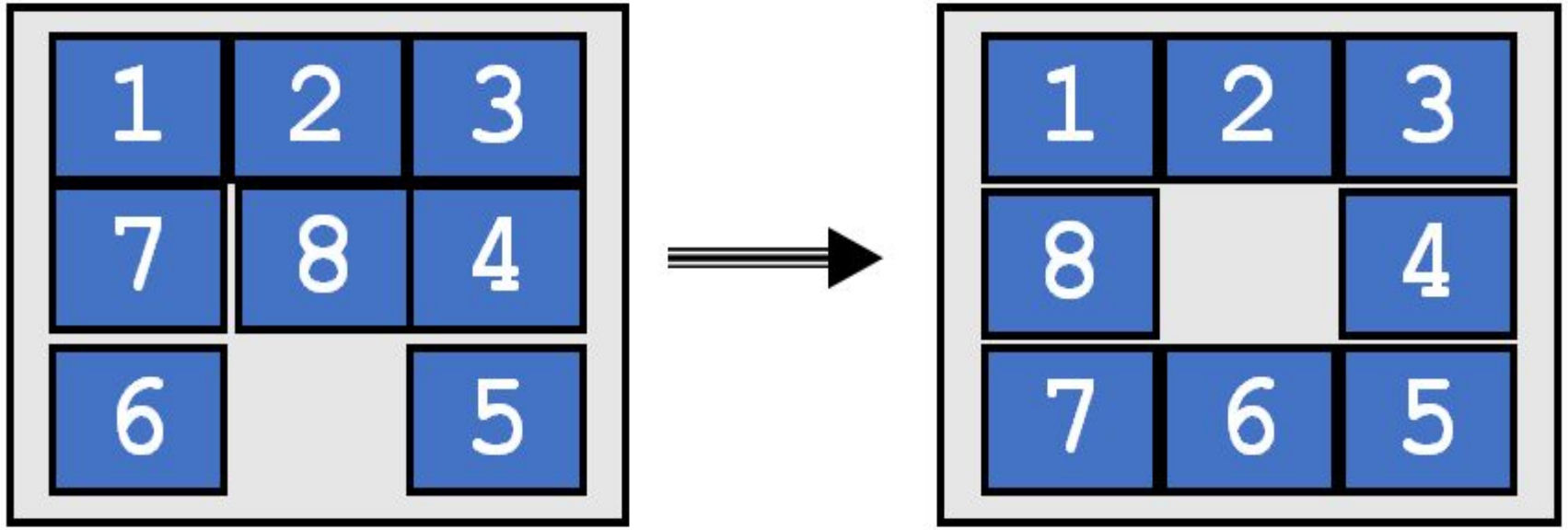
Learning objectives

- After reading this chapter, the students will be able to:
- To understand the concept of informed search (Heuristic Search).
- To apply and evaluate the various problems using the fundamental of search.

Introduction

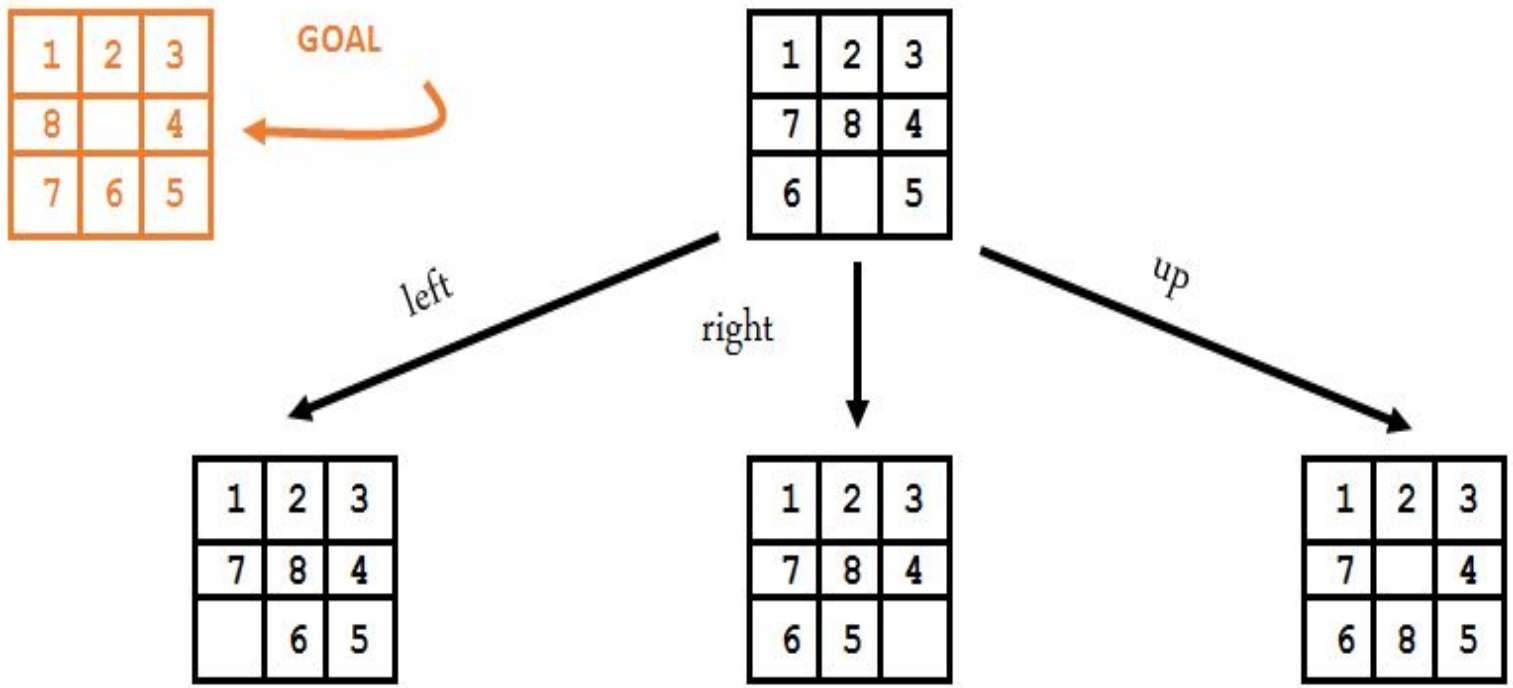
- Strategies that know whether one goal state is more promising than the other are called *heuristic/informed search techniques*. *These techniques work on the concept of heuristic and hence are also known as heuristic search techniques.*
- *Direct* techniques (blind search) are not always possible (they require too much time or memory).
- *Weak* techniques can be effective if applied correctly on the right kinds of tasks.
    - Typically require domain specific information.

8 Puzzle Heuristics



The 8-puzzle is a small board game for a single player; it consists of eight square tiles numbered 1 through 8 and one blank space on a 3 × 3 board moves of the puzzle are made by sliding an adjacent tile into the position occupied by the blank space, which has the effect of exchanging the positions of the tile and blank space. Only tiles that are horizontally or vertically adjacent (not diagonally adjacent) may be moved into the blank space.

Which move is best?



8 Puzzle Heuristics

- Blind search techniques used an arbitrary ordering (priority) of operations.
- Heuristic search techniques make use of domain specific information - a heuristic.
- What heurisitic(s) can we use to decide which 8-puzzle move is “best” (worth considering first).

8 Puzzle Heuristics- Approach 1

Number of tiles in the *Correct* position.

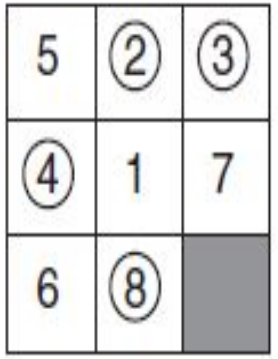
- The **heuristic function of the game may be as follows:** Count the number of tiles that are in place with respect to the goal state. Hence,

h(Initial State) = 4

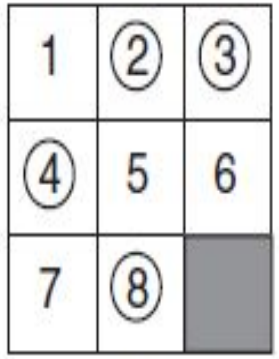
- As seen in the initial state, the number of tiles that are in the correct place with respect to the goal (Tiles 2, 3, 4 and 8 are in the same place as required in the goal state) state are four. Hence, the heuristic value for the initial state is 4 (Note, as a convention we do not consider the position of blank in the nongoal state).

h(Goal State) = 8

- Heuristic value for the goal state will always be eight as all the tiles are in correct place in the goal state.



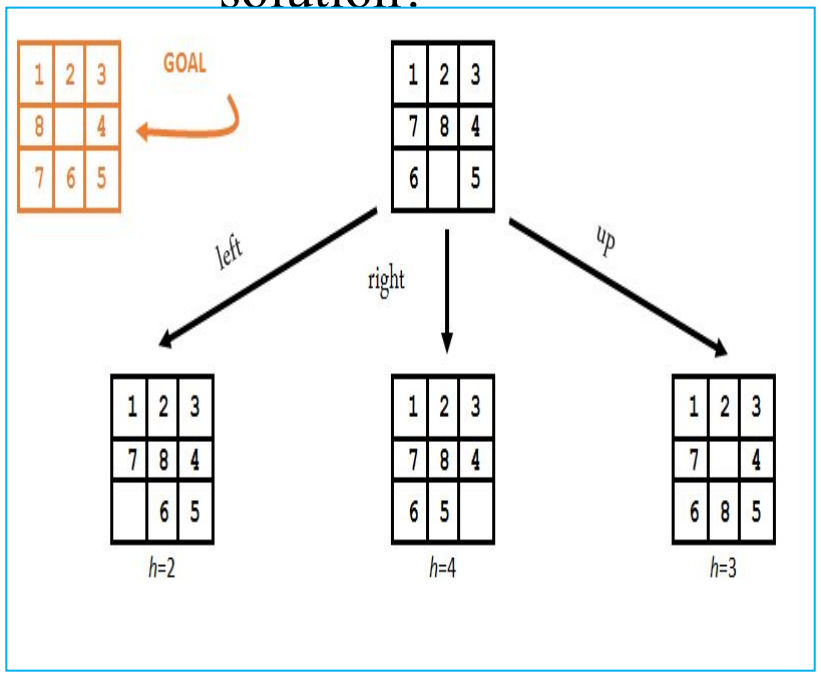
I.S.

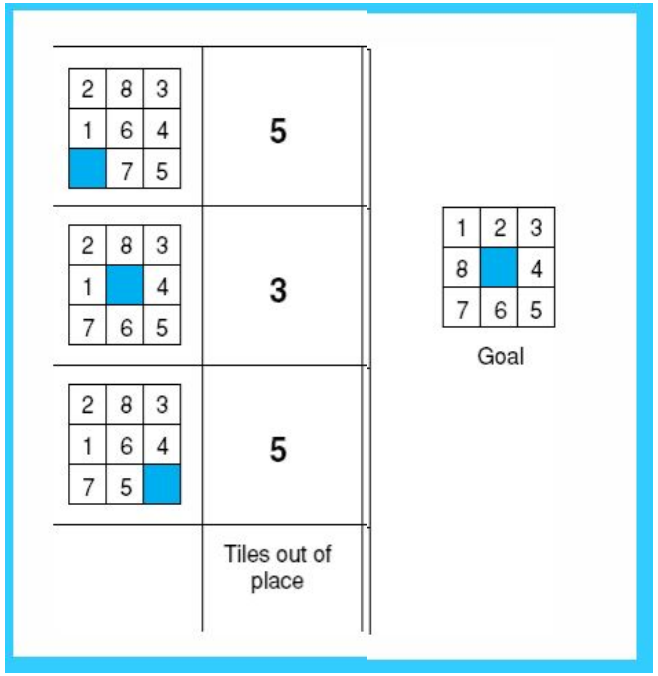


G.S.

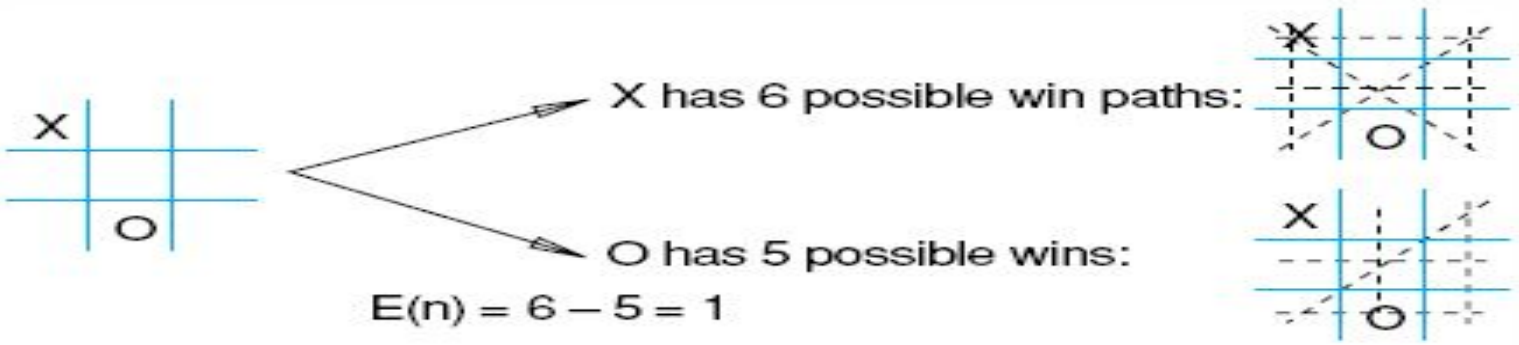
8 Puzzle Heuristics- Approach 2

- Number of tiles in the *incorrect* position.
  - This can also be considered a lower bound on the number of moves from a solution!
  - The "best" move is the one with the lowest number returned by the heuristic.
  - Is this heuristic more than a heuristic (is it always correct?).
    - Given any 2 states, does it always order them properly with respect to the minimum number of moves away from a solution?



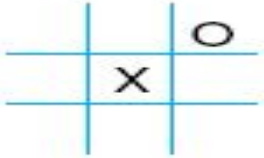


Heuristic for Tic-Tac-Toe Problem



X has 4 possible win paths;
O has 6 possible wins

E(n) = 4 – 6 = –2



X has 5 possible win paths;
O has 4 possible wins

E(n) = 5 – 4 = 1

Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

Role of Heuristic in Informed Search

Recall that the ordering of FRINGE defines the search strategy

Search Algorithm #2

SEARCH#2
1. INSERT(initial-node,FRINGE)
2. Repeat:
   a. If empty(FRINGE) then return failure
   b. N ← REMOVE(FRINGE)
   c. s ← STATE(N)
   d. If GOAL?(s) then return path or goal state
   e. For every state s' in SUCCESSORS(s)
      i. Create a node N' as a successor of N
      ii. INSERT(N',FRINGE)

Best-First Search

- It exploits state description to estimate how "good" each search node is
- An evaluation function f maps each node N of the search tree to a real number
  $f(N) \geq 0$
  [Traditionally, f(N) is an estimated cost; so, the smaller f(N), the more promising N]
- Best-first search sorts the FRINGE in increasing f
  [Arbitrary order is assumed among nodes with equal f]

Best-First Search

- It exploits state description to estimate how "good" each search node is
- An evaluation function the search tree to a re
  f(N) ≥ 0
  [Traditionally, f(N) is an e
  smaller f(N), the more pro
- Best-first search sorts the FRINGE in increasing f
  [Arbitrary order is assumed among nodes with equal f]

"Best" does not refer to the quality of the generated path
Best-first search does not generate optimal paths in general

How to construct f?

- Typically, f(N) estimates:
  - either the cost of a solution path through N

    Then f(N) = g(N) + h(N), where
    - g(N) is the cost of the path from the initial node to N
    - h(N) is an estimate of the cost of a path from N to a goal node
  - or the cost of a path from N to a goal node

    Then f(N) = h(N) □ Greedy best-search

- But there are no limitations on f. Any function of your choice is acceptable.
  But will it help the search algorithm?

How to construct f?

- Typically, f(N) estimates:
  - either the cost of a solution path through N

    Then f(N) = g(N) + h(N), where
    - g(N) is the cost of the path from the initial node to N
    - h(N) is an estimate of the cost of a path from N to a goal node
  - or the cost of a path from N to a goal node

    Then f(N) = h(N)

Heuristic function

- But there are no limitations on f. Any function of your choice is acceptable.
  But will it help the search algorithm?

Heuristic Function

- The heuristic function $h(N) \geq 0$ estimates the cost to go from STATE(N) to a goal state

  Its value is **independent of the current search tree**; it depends only on STATE(N) and the goal test GOAL?

- Example:

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

$h_1(N)$ = number of misplaced numbered tiles = 6

[Why is it an estimate of the distance to the goal?]

Other Examples

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

- $h_1(N)$ = number of misplaced numbered tiles = 6
- $h_2(N)$ = sum of the (Manhattan) distance of every numbered tile to its goal position
  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13
- $h_3(N)$ = sum of permutation inversions
  $= n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$
  $= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0$
  = 16

Admissible Heuristic

- Let h*(N) be the cost of the optimal path from N to a goal node
- The heuristic function h(N) is admissible if:

$$0 \leq h(N) \leq h^*(N)$$

- An admissible heuristic function is always optimistic !

Admissible Heuristic

- Let h*(N) be the cost of the optimal path from N to a goal node
- The heuristic function h(N) is admissible if:

$$0 \leq h(N) \leq h^*(N)$$

- An admissible heuristic function is always optimistic !

G is a goal node ⇒ h(G) = 0

8-Puzzle Heuristics

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  $= 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13$
  is admissible
- $h_3(N)$ = sum of permutation inversions
  $= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16$
  is not admissible

8-Puzzle Heuristics

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

- $h_1(N)$ = number of misplaced tiles = 6
  is admissible
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  $= 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13$
  is ???

8-Puzzle Heuristics

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

- $h_1(N)$ = number of misplaced tiles = 6
  is admissible
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  $= 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13$
  is admissible
- $h_3(N)$ = sum of permutation inversions
  $= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16$
  is ???

8-Puzzle Heuristics

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

- $h_1(N)$ = number of misplaced tiles = 6
  is admissible
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  $= 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13$
  is admissible
- $h_3(N)$ = sum of permutation inversions
  $= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16$
  is not admissible

- Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value.
- It stops when it reaches a "peak" where no neighbour has higher value. This algorithm is considered to be one of the simplest procedures for implementing heuristic search. The hill climbing comes from the idea that if you are trying to find the top of the hill and you go up direction from wherever you are.
- This heuristic combines the advantages of both depth-first and breadth-first searches into a single method.

Types of Hill Climbing

- *Simple Hill Climbing*

It examines the neighbouring nodes one-by-one and selects the first neighbouring node which optimises the current cost as

- *Steepest-ascent Hill Climbing*

At first examines all the neighbouring nodes and then selects the node closest to the solution state as the next nodenext node.

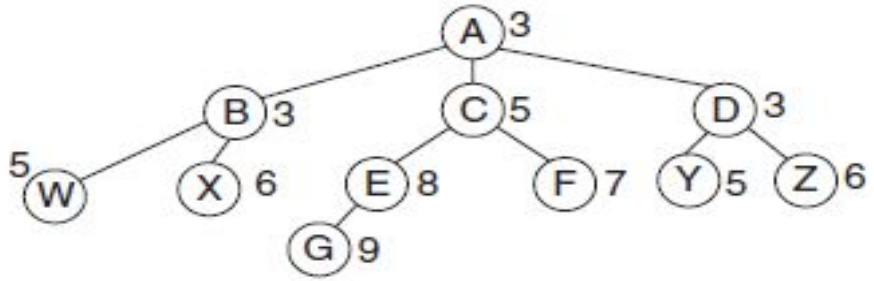
- *Stochastic Hill Climbing*

It does not examine all the neighbouring nodes before deciding which node to select. It just selects a neighbouring node at random, and decides (based on the amount of improvement in that neighbour) whether to move to that neighbour or to examine another.

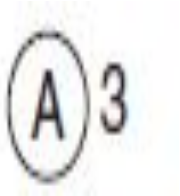
Begin
/*Initially OPEN contains the root node and CLOSE is EMPTY*/
OPEN=[start]
CLOSE=[]
/*We Continue the loop till OPEN list is not EMPTY*/
While OPEN≠[] do
Begin
Remove the left most state from OPEN and call it X
If X=GOAL **then**
Return SUCCESS
Else
Begin
1. Generate children of X
2. Put X on close
3. Discard the children of X if already on OPEN or CLOSE
4. Sort the remaining Children according to the heuristic value of each state
5. Put the children on left side of OPEN in sorted order.
/*Note that the children are sorted first according to the heuristic merit and are then put to the left sideof OPEN*/
End
End
Return Fail
End

The algorithm described is called as *steepest ascent hill climbing.*

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



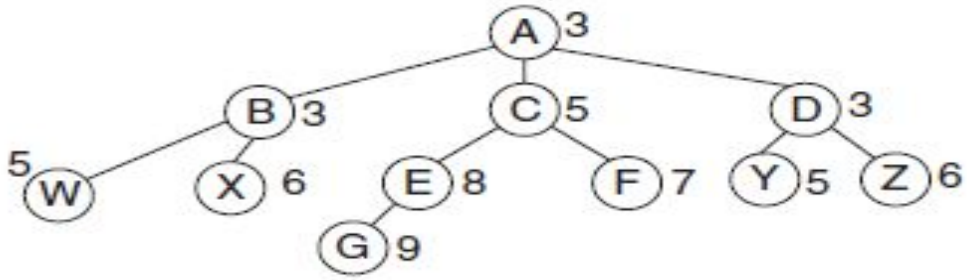
Step 1: The hill climbing algorithm starts with the initial state.



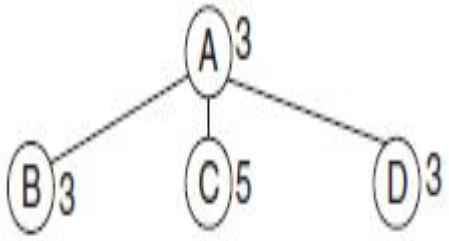
OPEN = [A3]
CLOSE = []

Note that leftmost node in the OPEN list is Node A, and hence, we expand Node A in the next step.

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



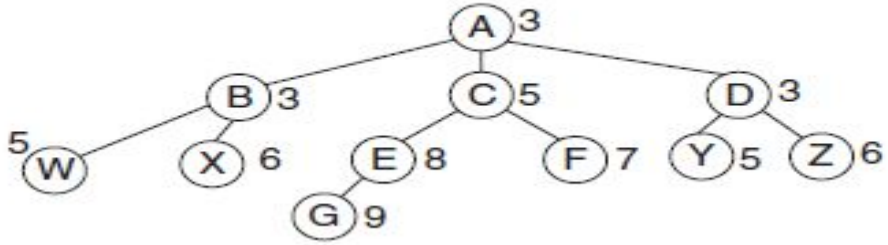
Step 2: All the children of Node A are now generated.



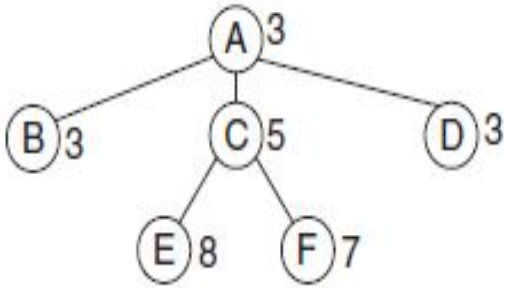
OPEN = [C5, B3, D3]
CLOSE = [A]

Note that the leftmost node in the OPEN list is Node C. Hence, we expand Node C in Step 3.

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



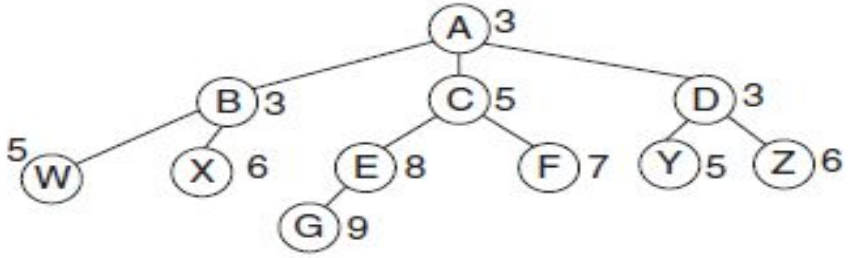
Step 3: Amongst all the children of Node root Node A Node C has the highest heuristic. Hence, all the children of Node C are now generated.



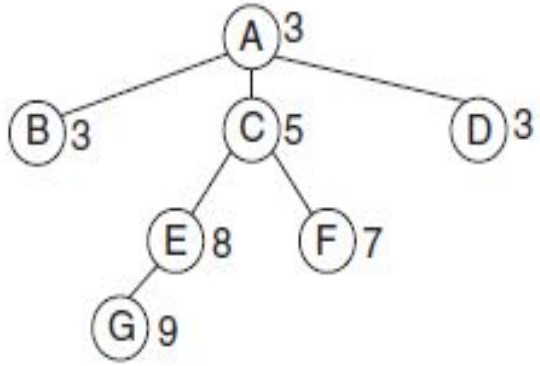
OPEN = [E8, F7, B3, D3]
CLOSE = [A, C]

Note that leftmost node in OPEN list is Node E. Hence, we expand Node E in Step 4.

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



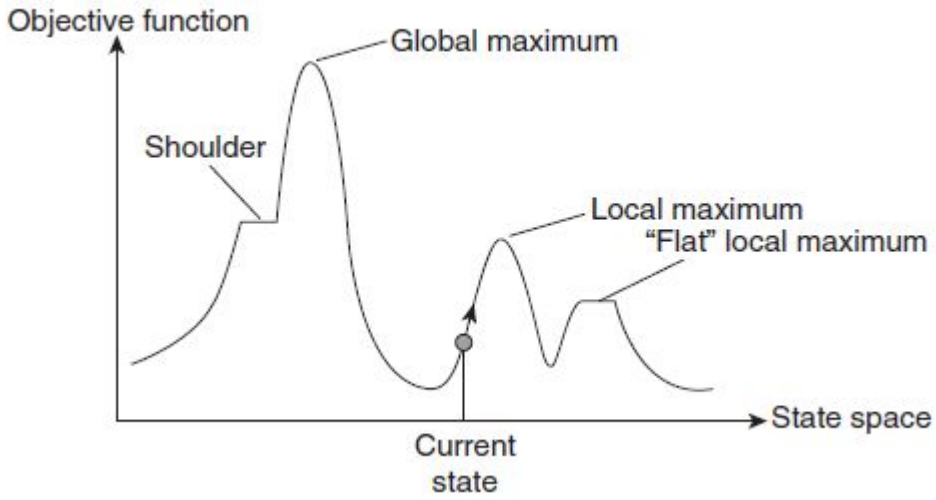
Step 4: Now the algorithm proceeds towards Node E and among all the children of Node E (only G is the child of Node C). Node G has a highest Heuristic merit; and hence, Node G is generated.



OPEN = [G9, F7, B3, D3]
CLOSE = [A, C, E]

Note that the leftmost node in OPEN list is Node G and Node G is a GOAL and hence we STOP

Demerits of Hill Climbing



State–space diagram is a graphical representation of the set of states our search algorithm can reach versus
the value of our objective function (the function which we wish to maximise).

X-axis: It denotes the state space, that is, states or configuration our algorithm may reach.

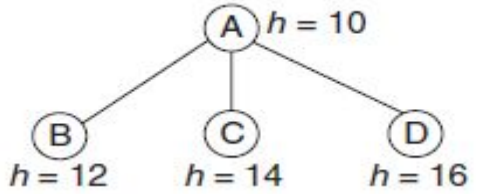
Y-axis: It denotes the values of objective function corresponding to a particular state. The best solution will be that state space where objective function has maximum value (global maximum).

Different Regions in the State–Space Diagram

1. Local maximum: It is a state which is better than its neighbouring state. However, there exists a state which is better than it (global maximum). This state is better because here value of objective function is higher than its neighbours.
2. Global maximum: It is the best possible state in the state–space diagram. This because at this state, objective function has the highest value.
3. Plateau/Flat local maximum: It is a flat region of state space where neighbouring states have the same value.
4. Ridge: It is region which is higher than its neighbours but itself has a slope. It is a special kind of local maximum.
5. Current state: The region of state–space diagram, where we are currently present during the search.
6. Shoulder: It is a plateau that has an uphill edge.

Problems in Different Regions in Hill climbing

1. Local maximum: At a local maximum all neighbouring states having a values, which is worse than the current state. Since, hill climbing uses greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.



To overcome local maximum problem Solution to the problem are:

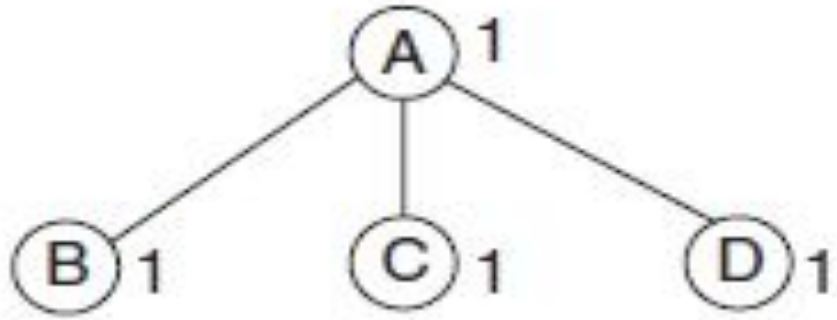
(a) One possible solution is backtracking.

We can backtrack to some earlier node and try to go in a different direction to attain the global peak. We can maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end.

(b) Another solution can be a list of promising plan.

Problems in Different Regions in Hill climbing

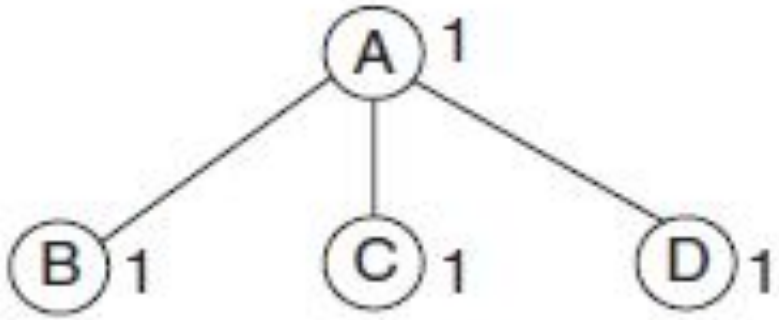
2. Plateau: On plateau all neighbours have same value. Hence, it is not possible to select the best direction. Plateau is a flat area of the search space in which a whole set of neighbouring states has the value. On a plateau, it is not possible to determine the best direction in which to move the local comparisons.



To overcome plateaus:
(a) A big jump in some direction can be done in order to get to a new section of search space. This method is recommended as in a plateau all neighbouring points have the same value.
(b) Another solution is to apply small steps several times in the same direction. This depends on the rules available.

Problems in Different Regions in Hill climbing

3. Ridge: Any point on a ridge can look like peak because movement in all possible directions is downwards. Hence, the algorithm stops when it reaches this state.

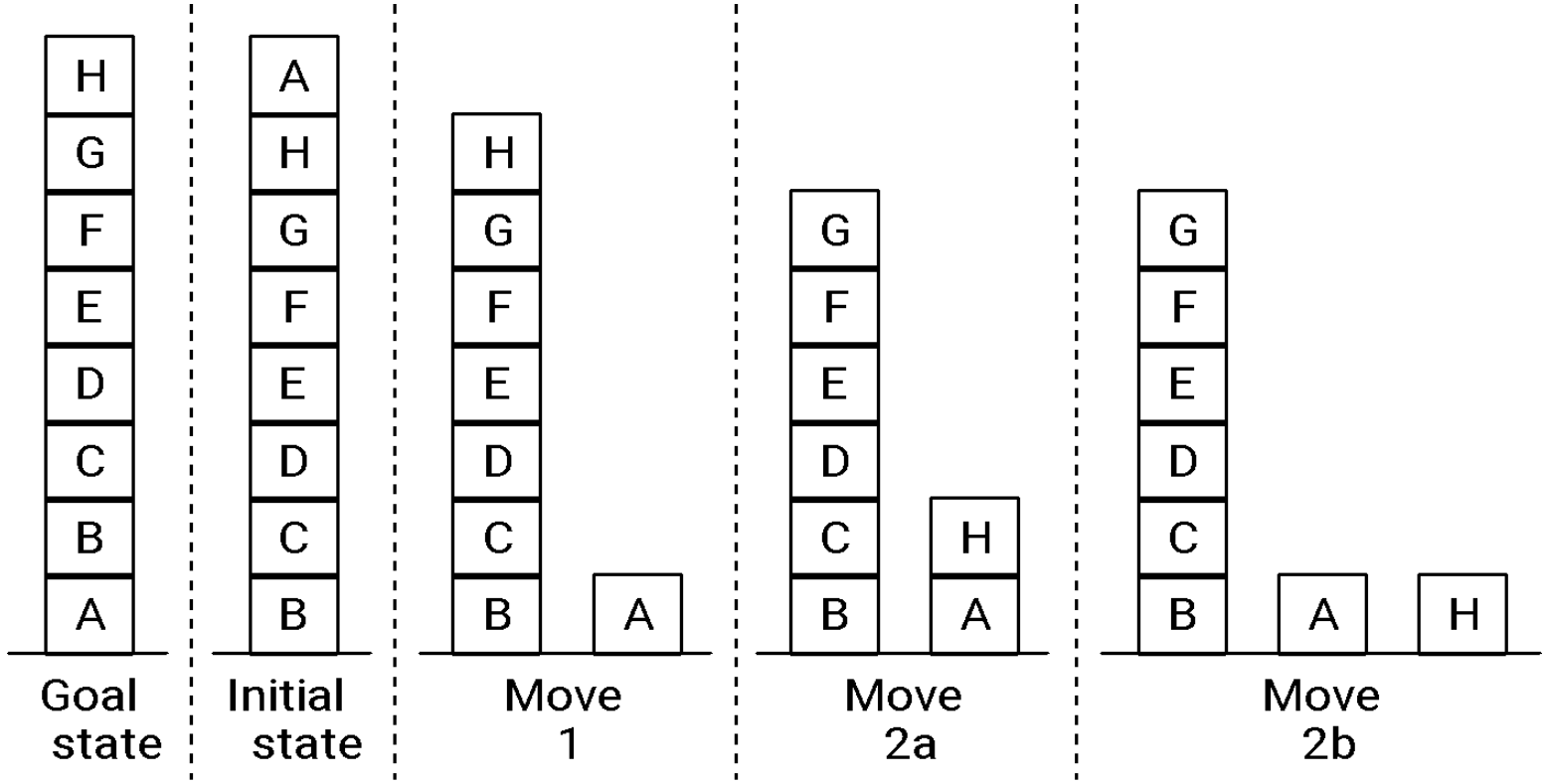


To overcome plateaus:

(a) A big jump in some direction can be done in order to get to a new section of search space. This method is recommended as in a plateau all neighbouring points have the same value.

(b) Another solution is to apply small steps several times in the same direction. This depends on the rules available.

A hill climbing example



A hill climbing example

A local heuristic function

Count +1 for every block that sits on the correct thing. The goal state has the value +8.

Count -1 for every block that sits on an incorrect thing. In the initial state blocks C, D, E, F, G, H count +1 each. Blocks A, B count -1 each , for the total of +4.

Move 1 gives the value +6 (A is now on the correct support). Moves 2a and 2b both give +4 (B and H are wrongly situated). This means we have a local maximum of +6.

A hill climbing example

A global heuristic function

Count +N for every block that sits on a correct stack of N things. The goal state has the value +28.

Count -N for every block that sits on an incorrect stack of N things. That is, there is a large penalty for blocks tied up in a wrong structure.

In the initial state C, D, E, F, G, H count -1, -2, -3, -4, -5, -6. A counts -7 , for the total of -28.

continued

A hill climbing example

Move 1 gives the value -21 (A is now on the correct support).

Move 2a gives -16, because C, D, E, F, G, H count -1, -2, -3, -4, -5, -1.

Move 2b gives -15, because C, D, E, F, G count -1, -2, -3, -4, -5.

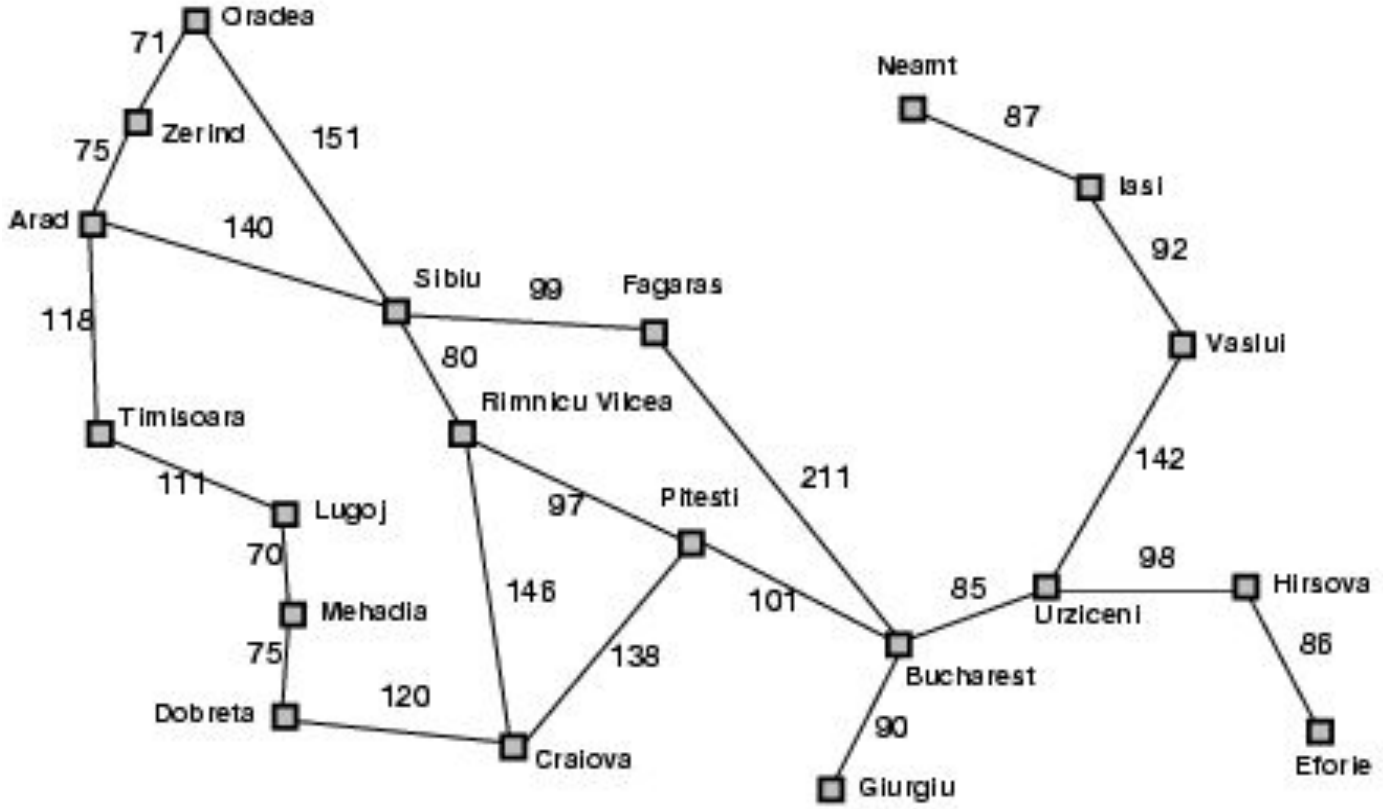
There is no local maximum!

Moral: sometimes changing the heuristic function is all we need.

Greedy best-first search

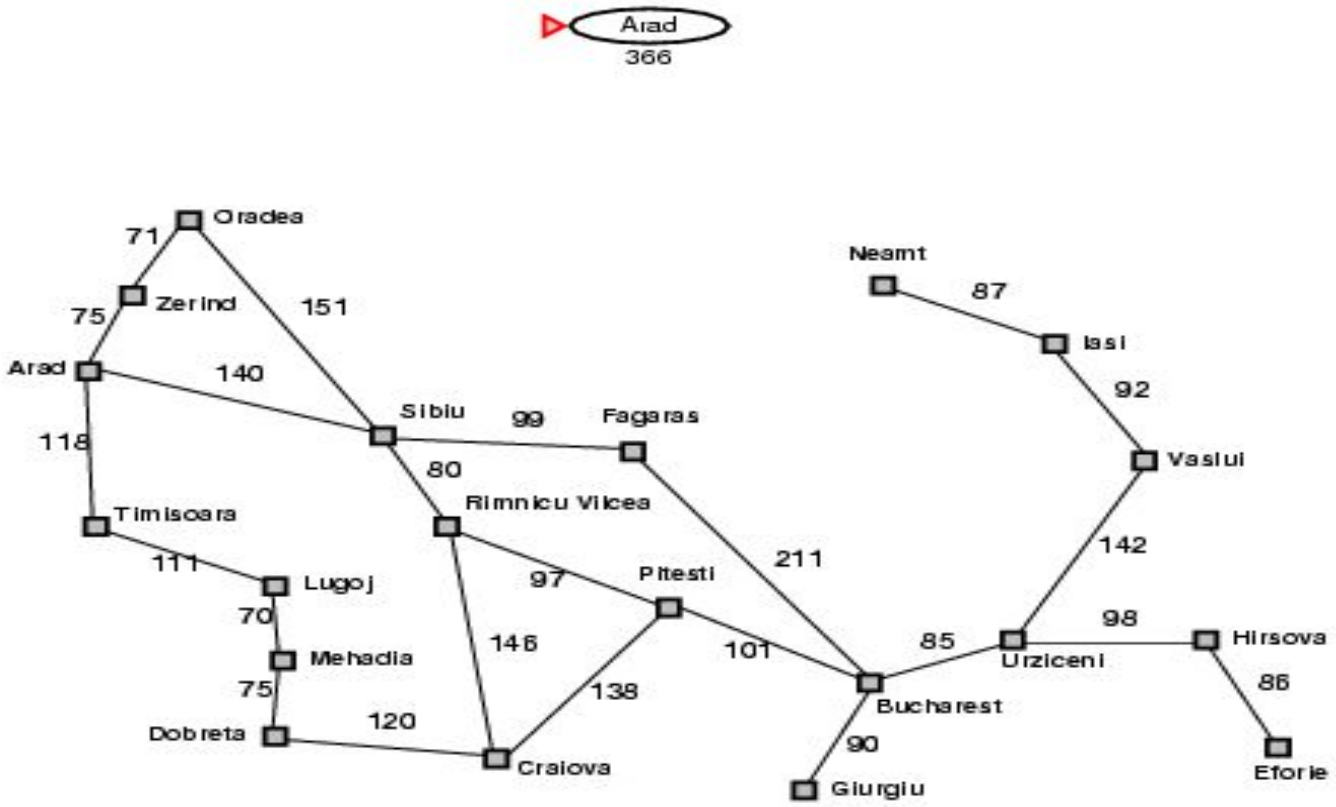
- f(n) = estimate of cost from *n* to *goal*
- e.g., $f_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal.

Romania with straight-line dist.



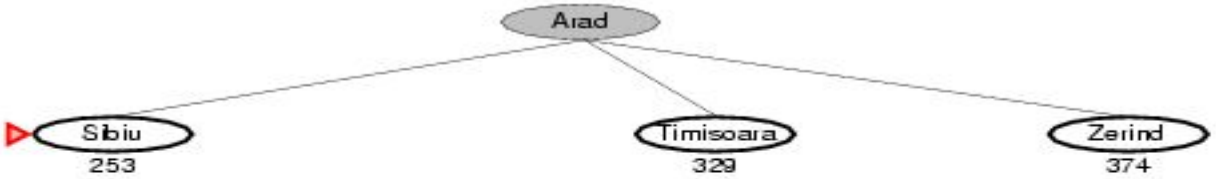
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

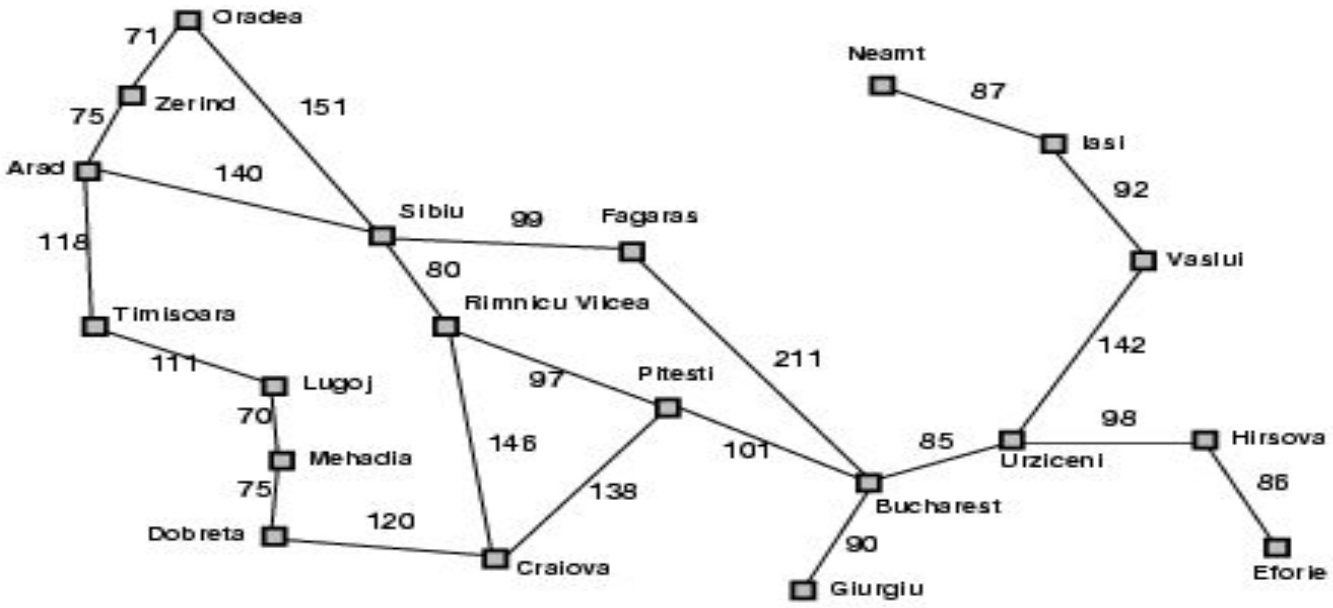
Greedy best-first search example



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

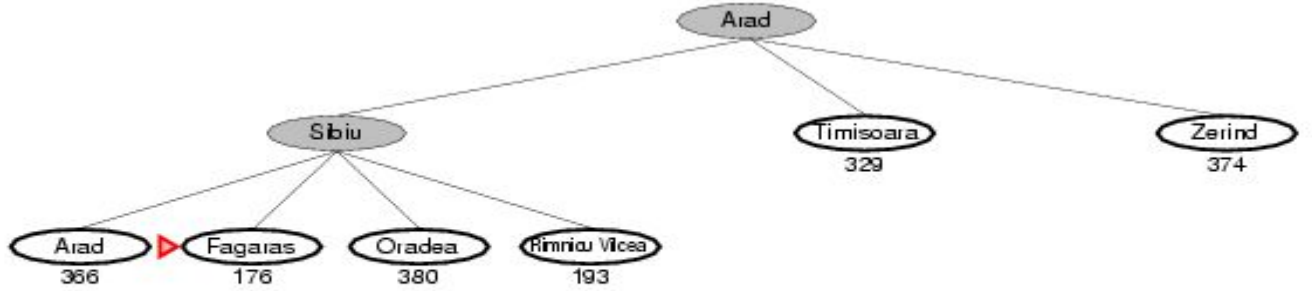
Greedy best-first search example

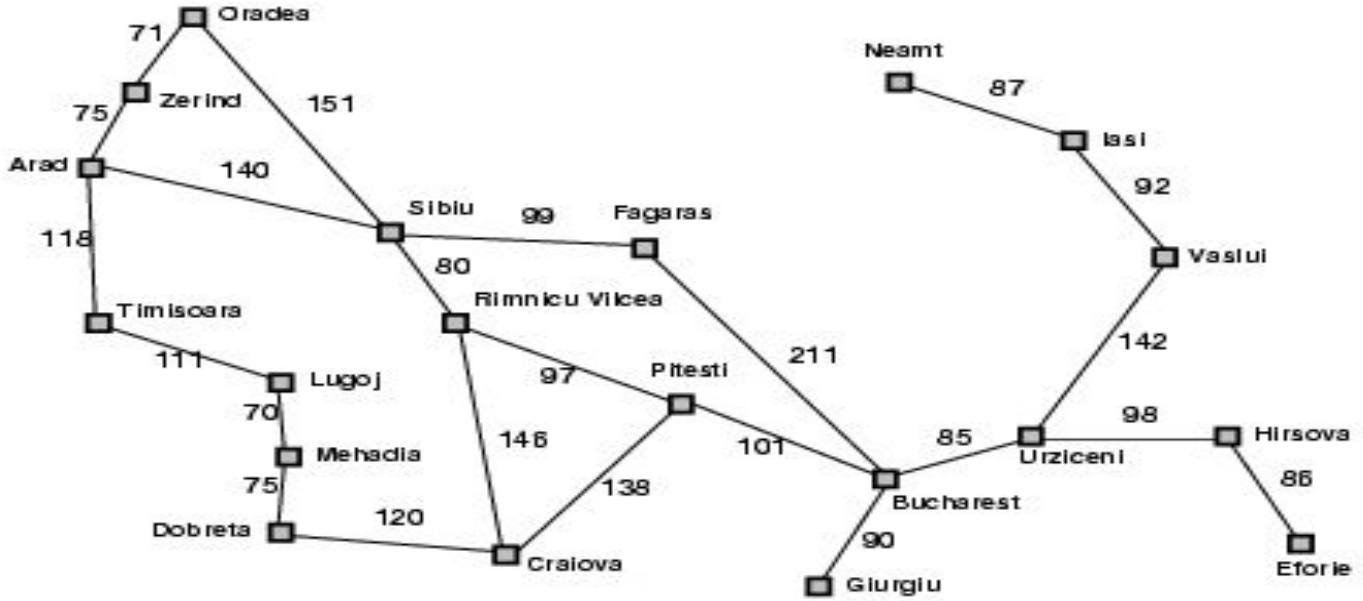




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

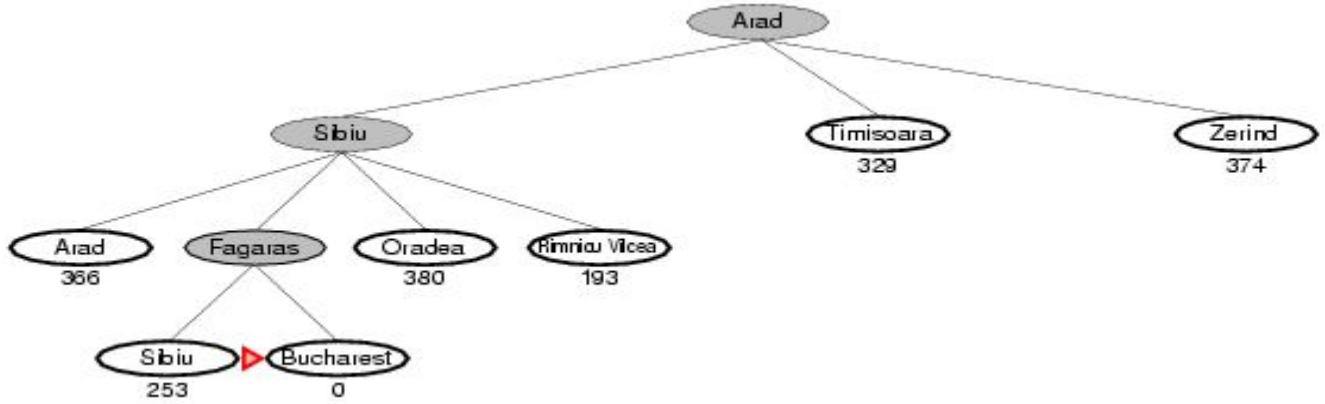


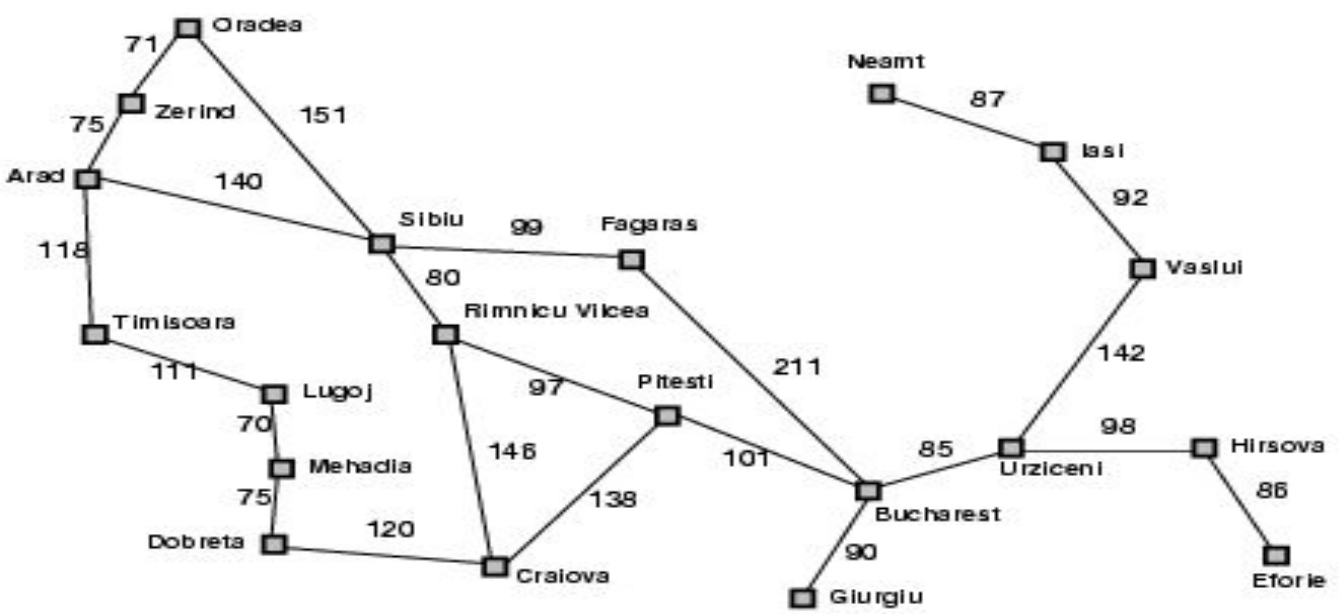


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example





Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of greedy best-first search

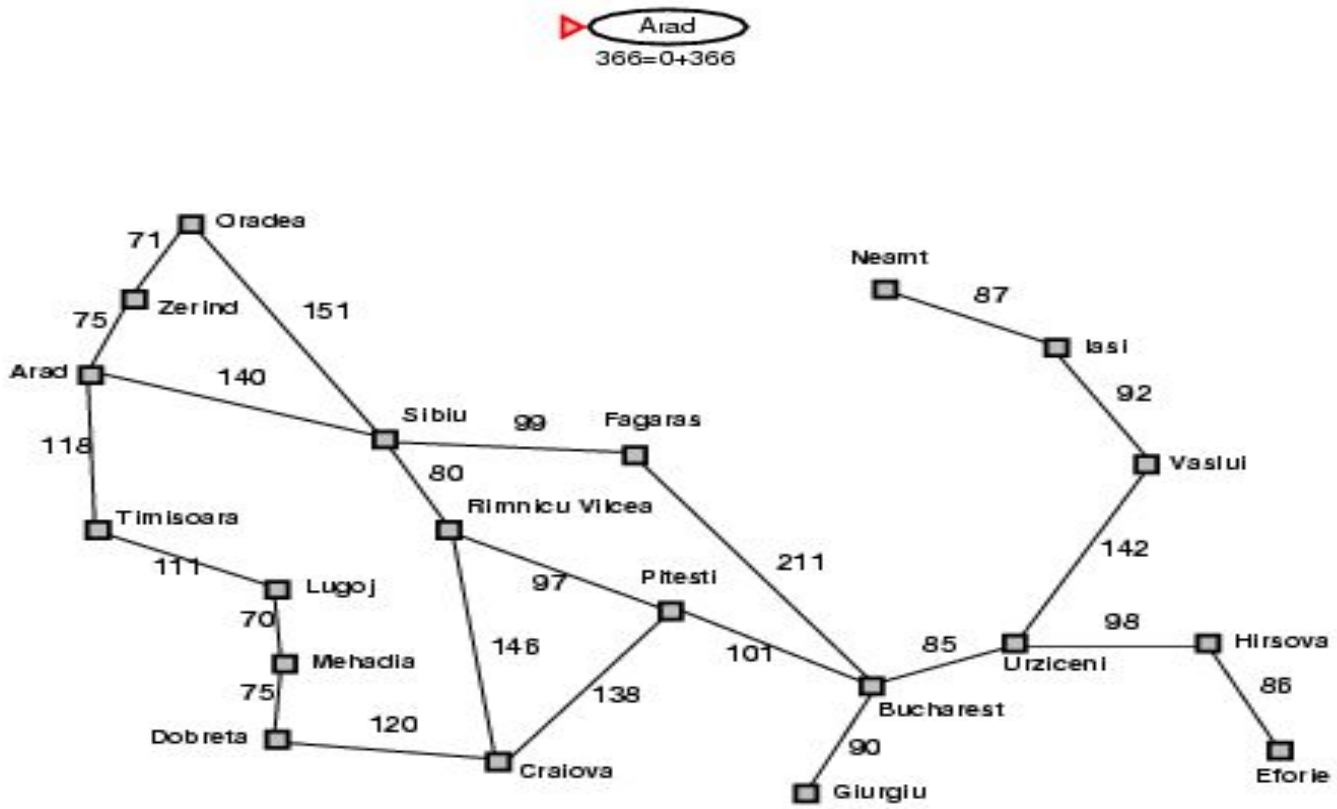
- Complete? No – can get stuck in loops.
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ - keeps all nodes in memory
- Optimal? No

  e.g. Arad□Sibiu□Rimnicu Virea□Pitesti□Bucharest is shorter!

A* search

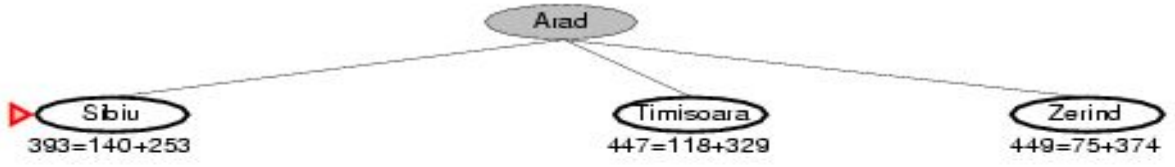
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal
- Best First search has $f(n)=h(n)$

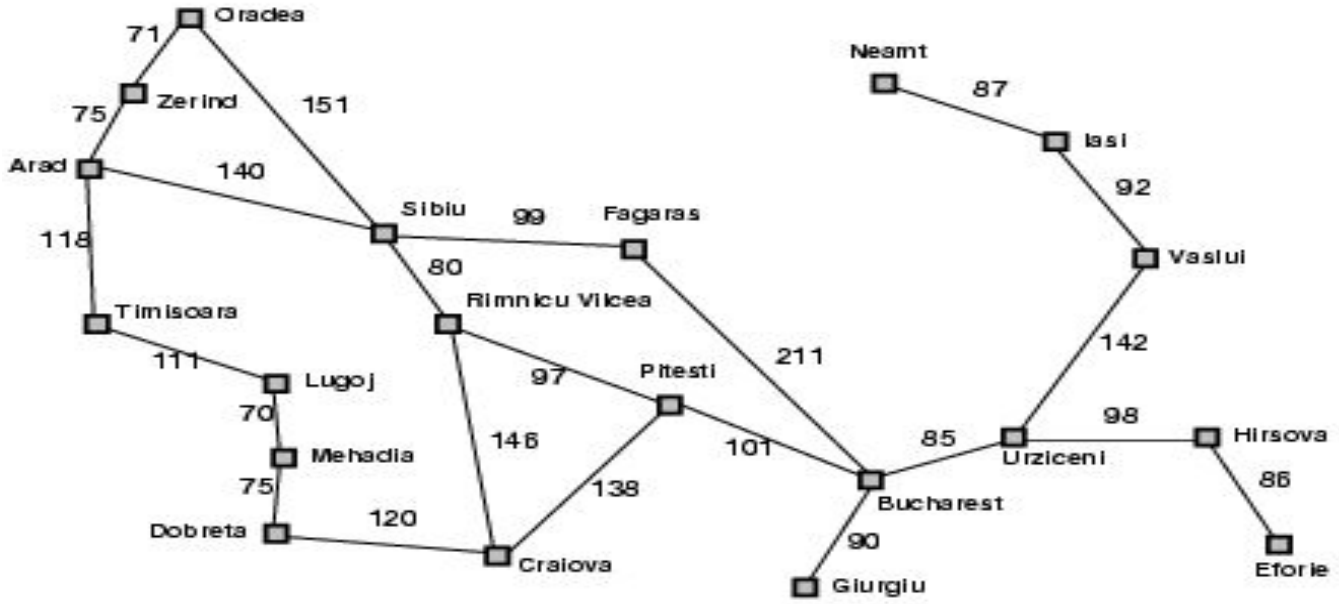
A* search example



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

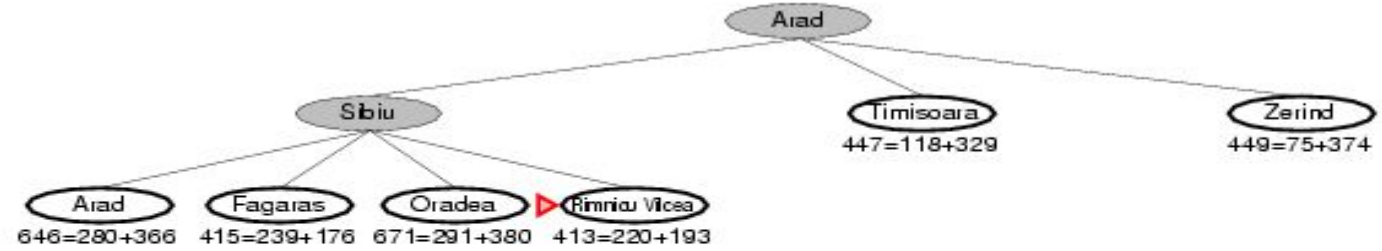
A* search example

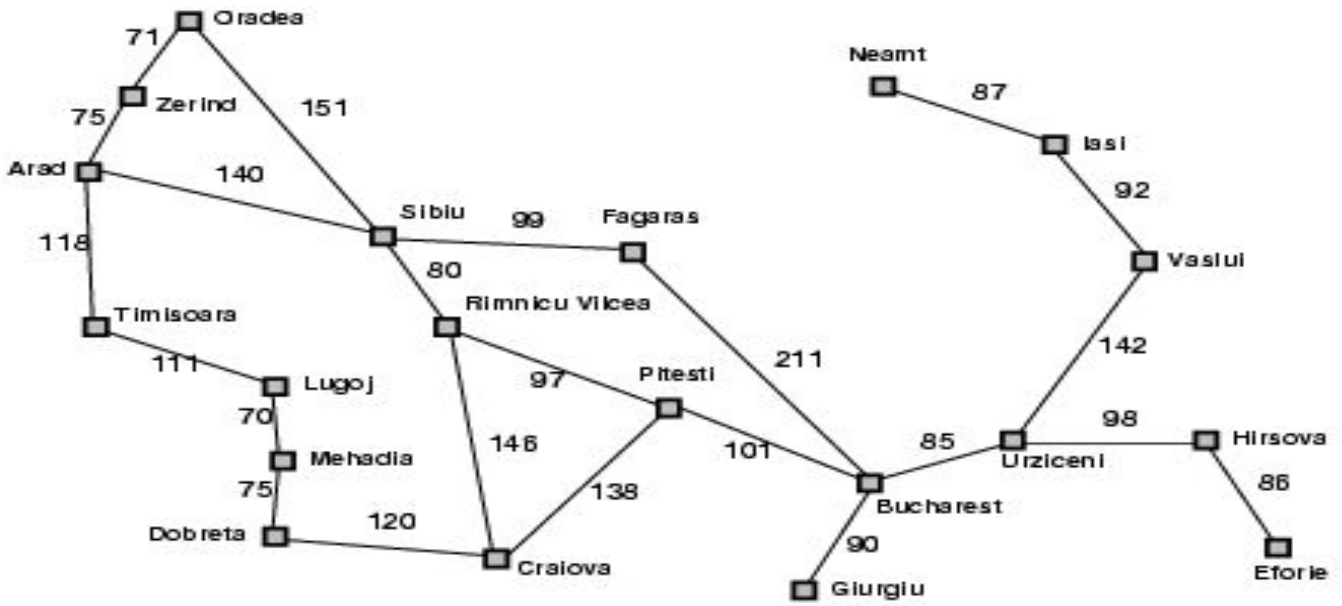




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

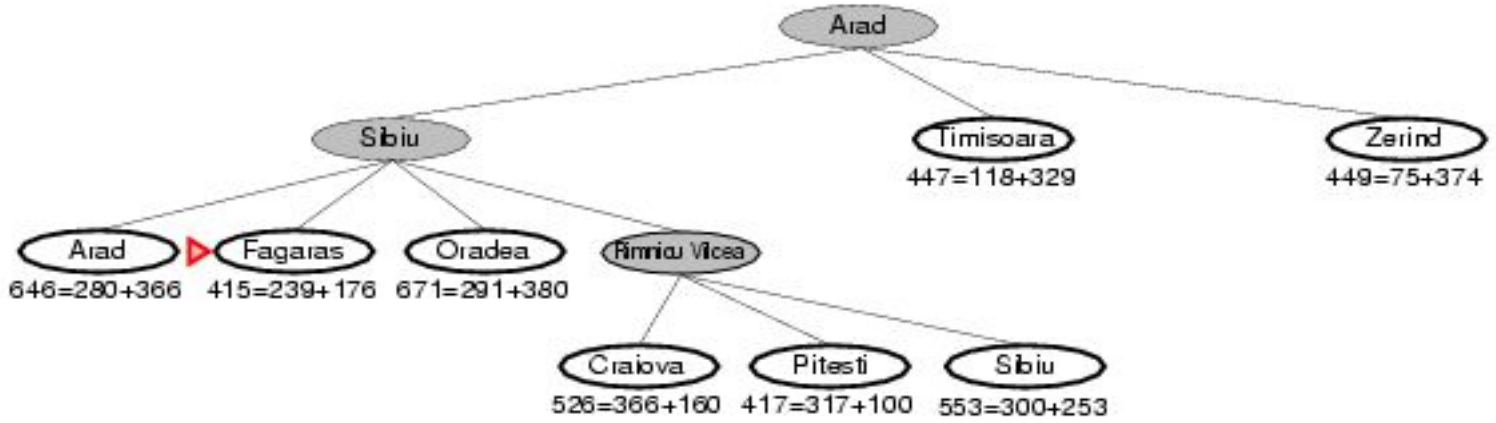


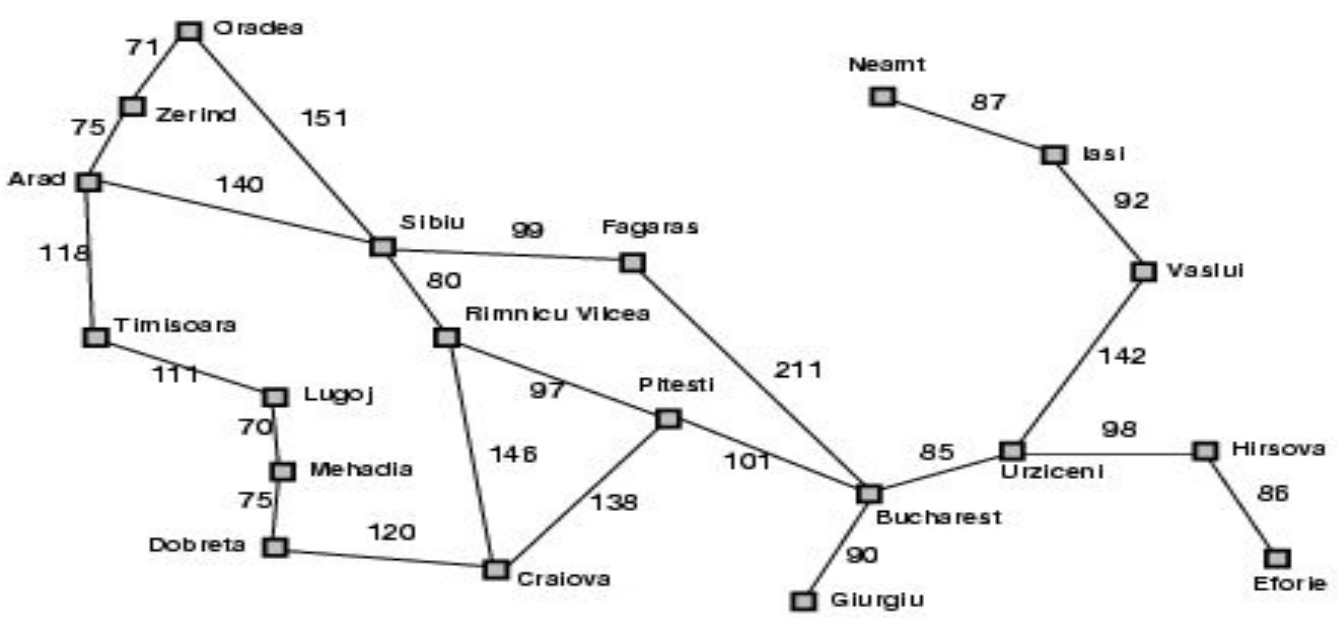


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

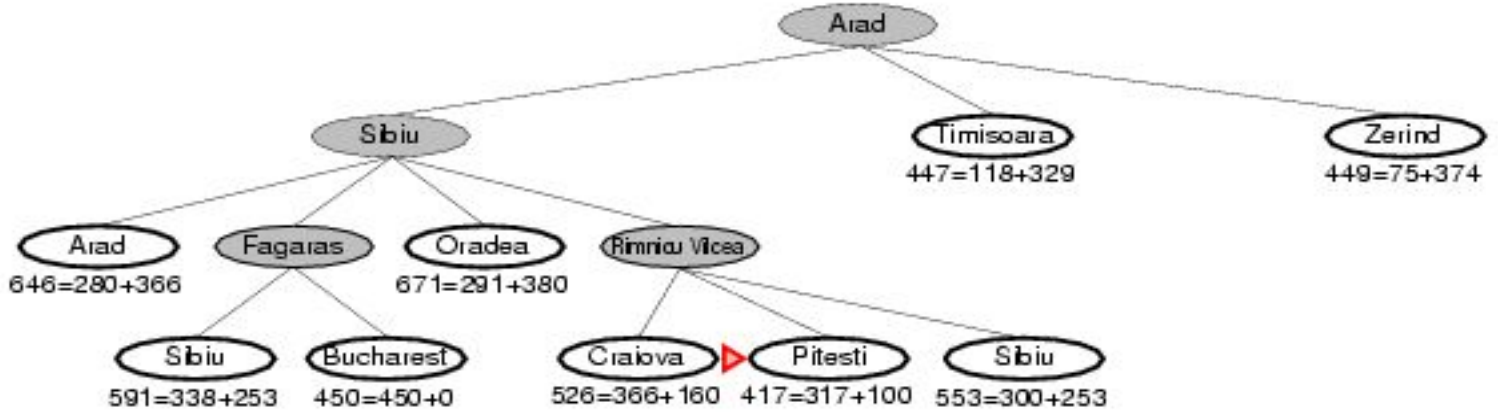
A* search example

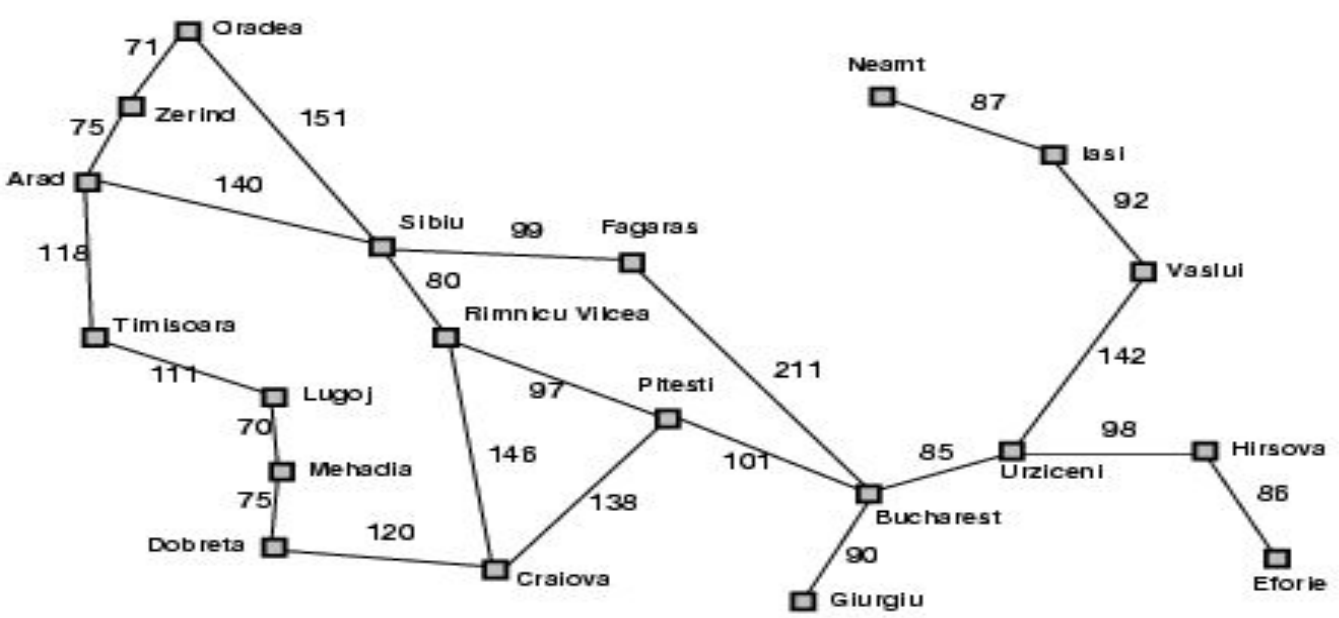




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

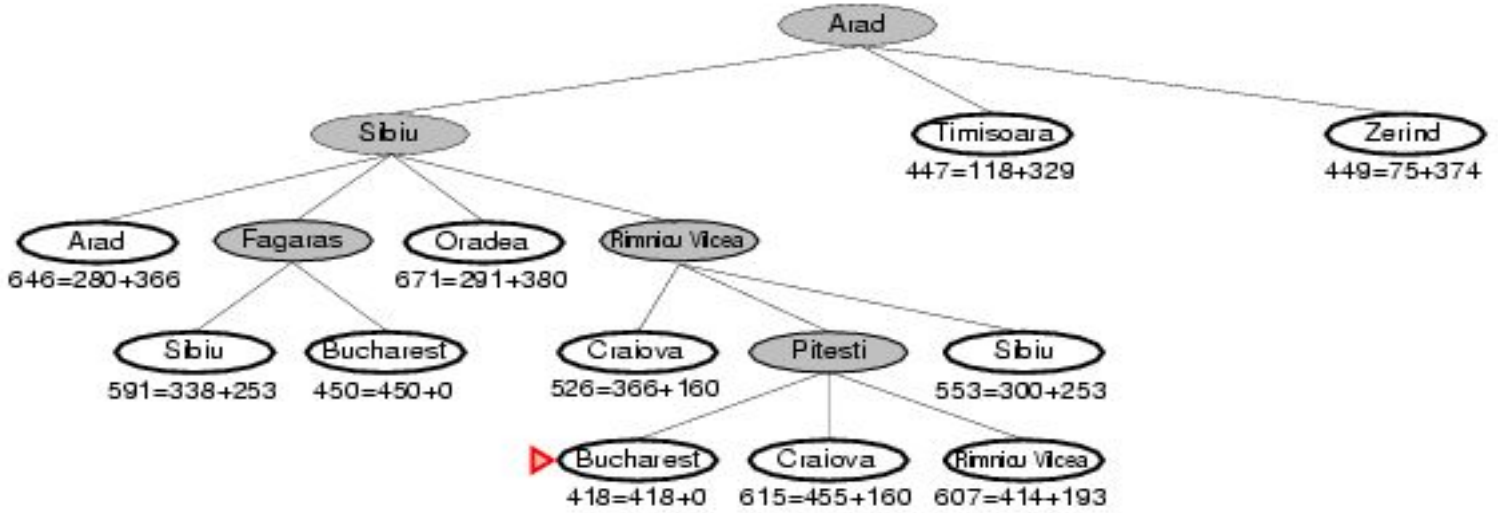
A* search example

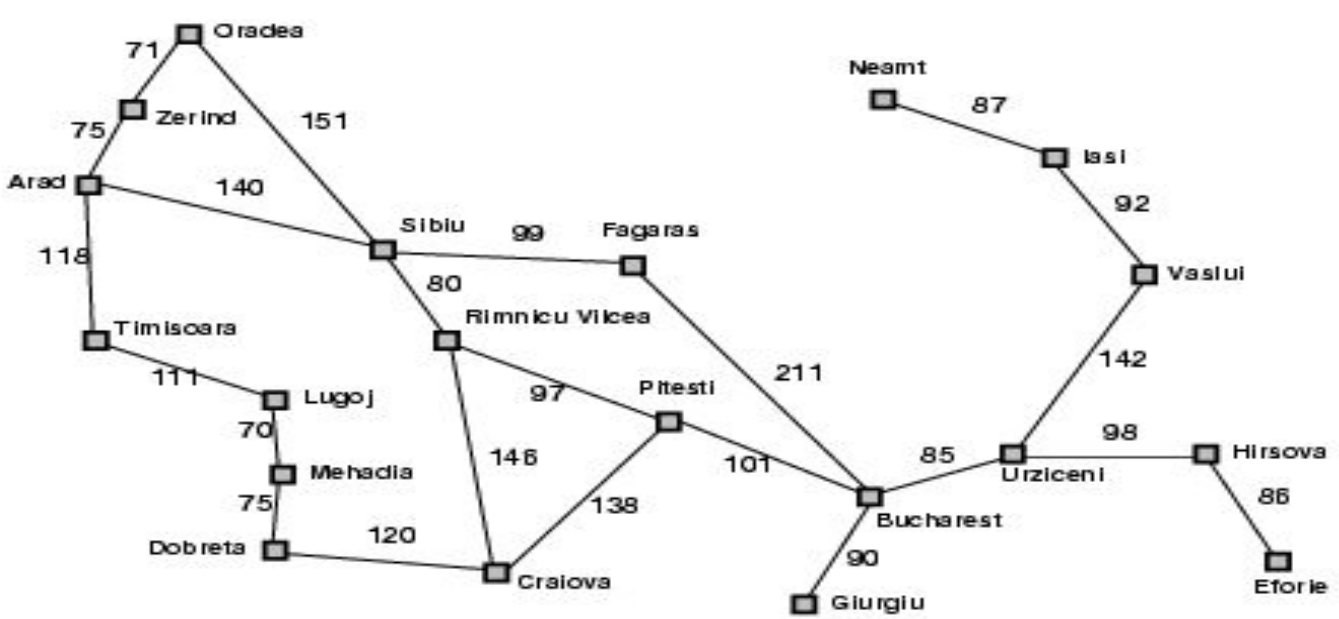




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

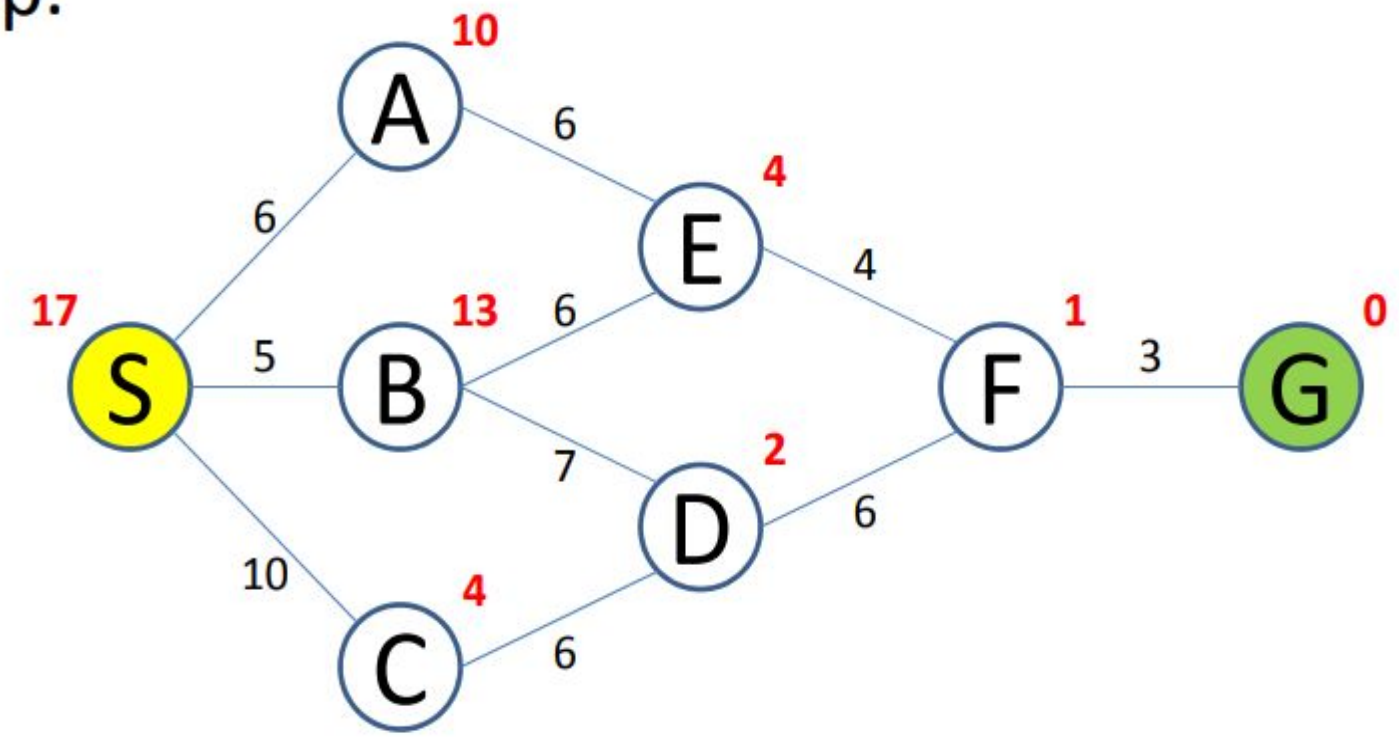




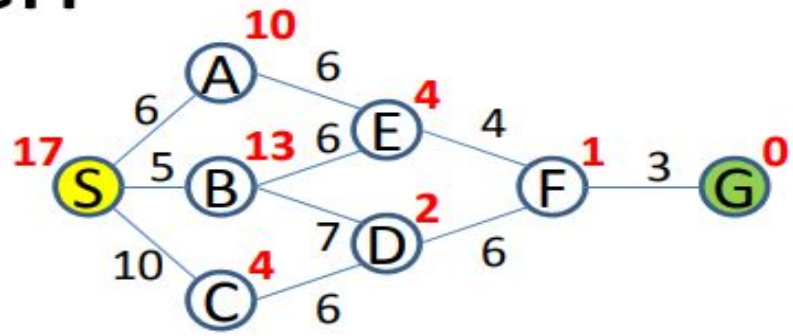
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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

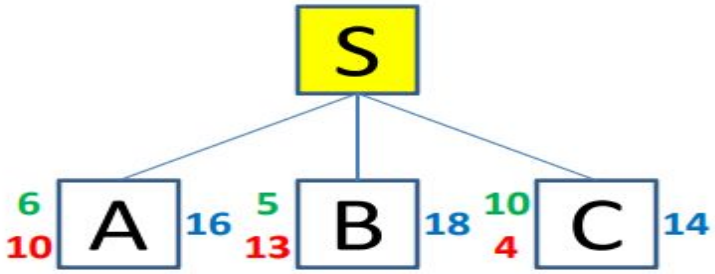


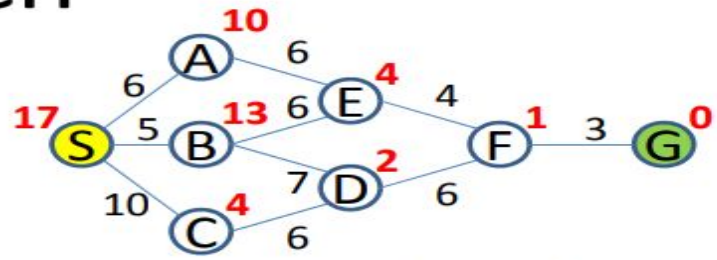
A* Search



QUEUE:
S

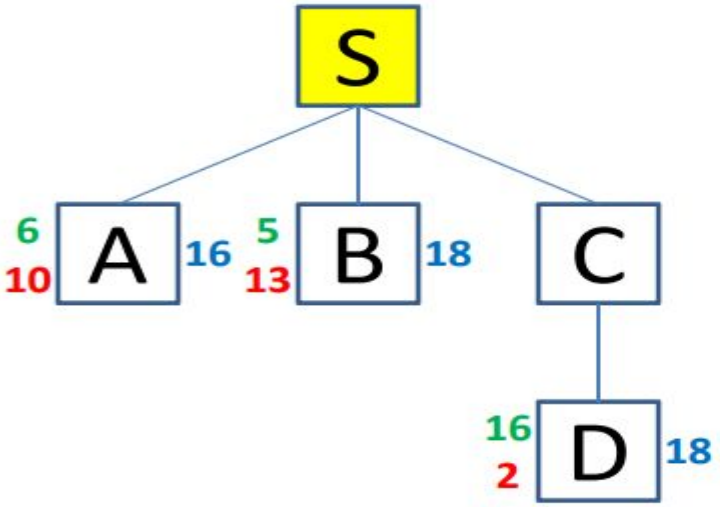
A* Search

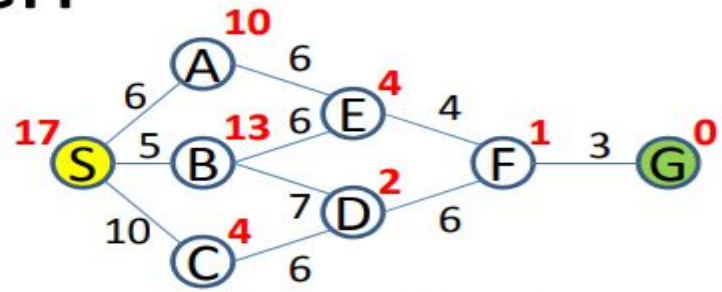




QUEUE:
SC
SA
SB

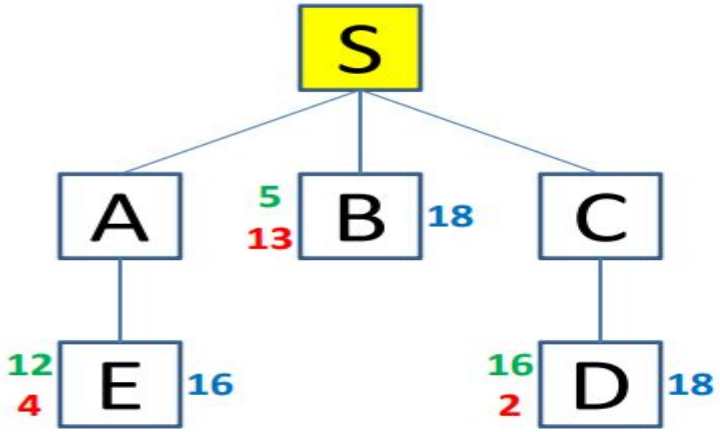
A* Search

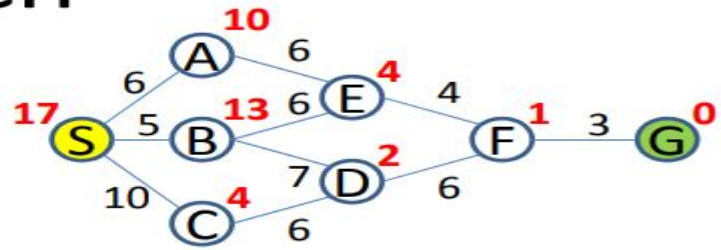




QUEUE:
SA
SCD
SB

A* Search

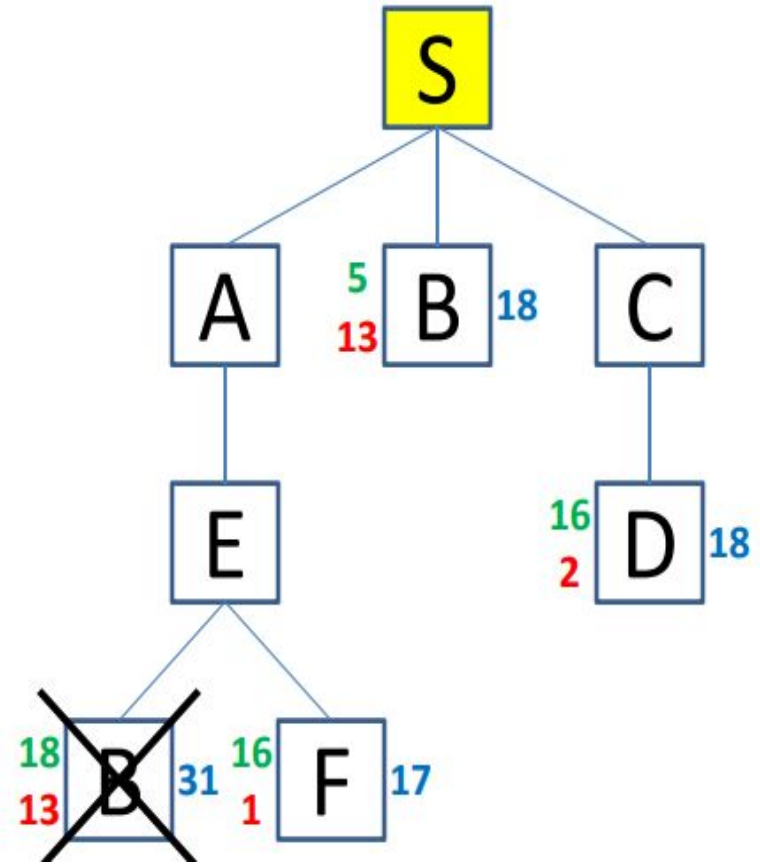


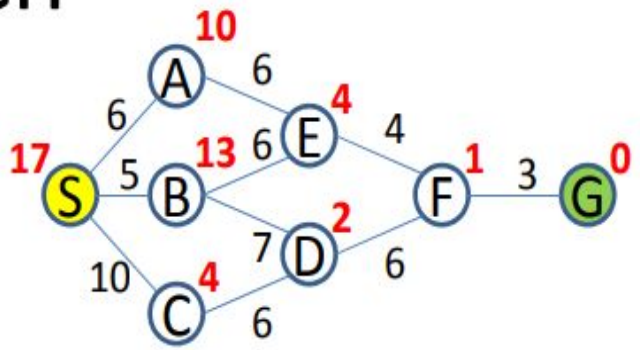


QUEUE:
SAE
SCD
SB

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

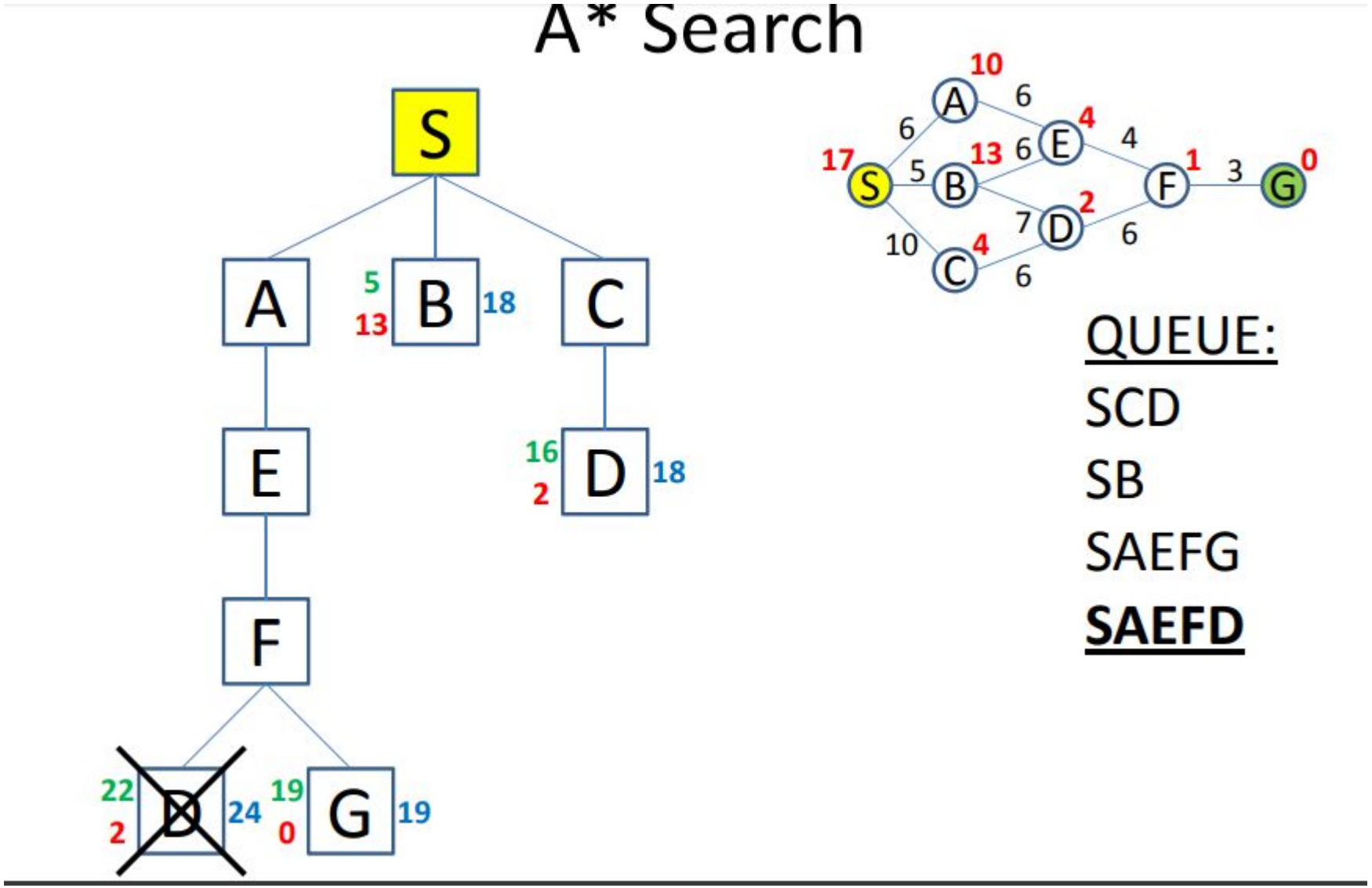
A* Search





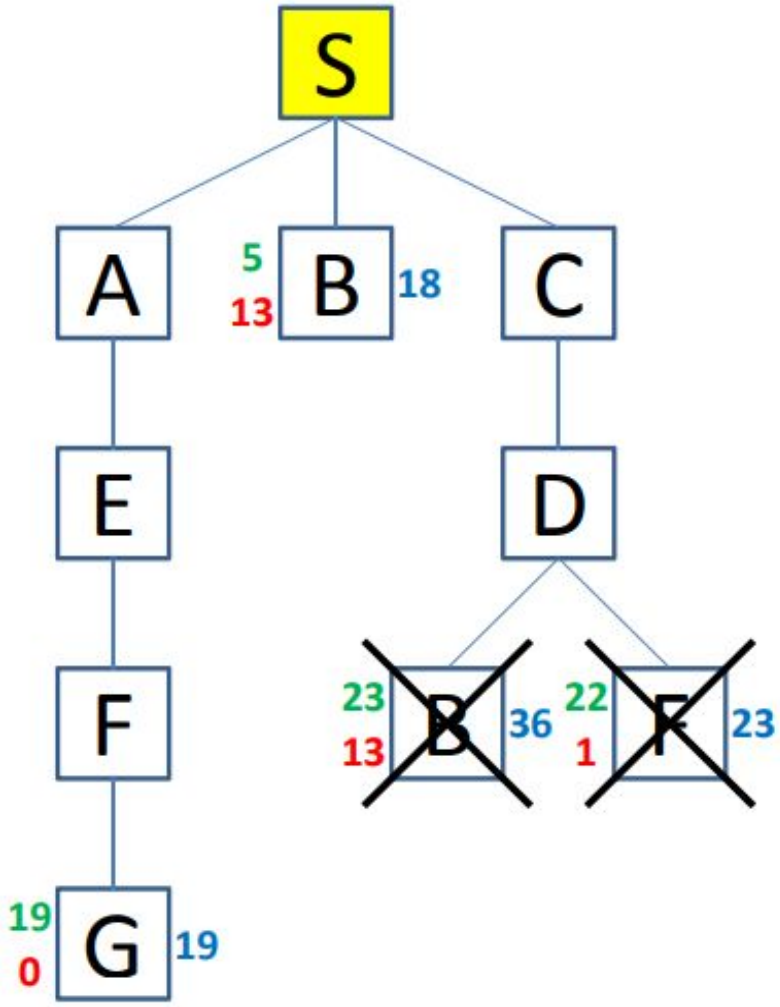
QUEUE:
SAEF
SCD
SB
SAEB

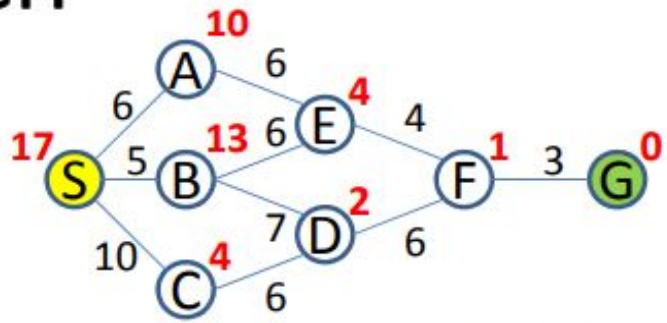
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search



5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

A* Search

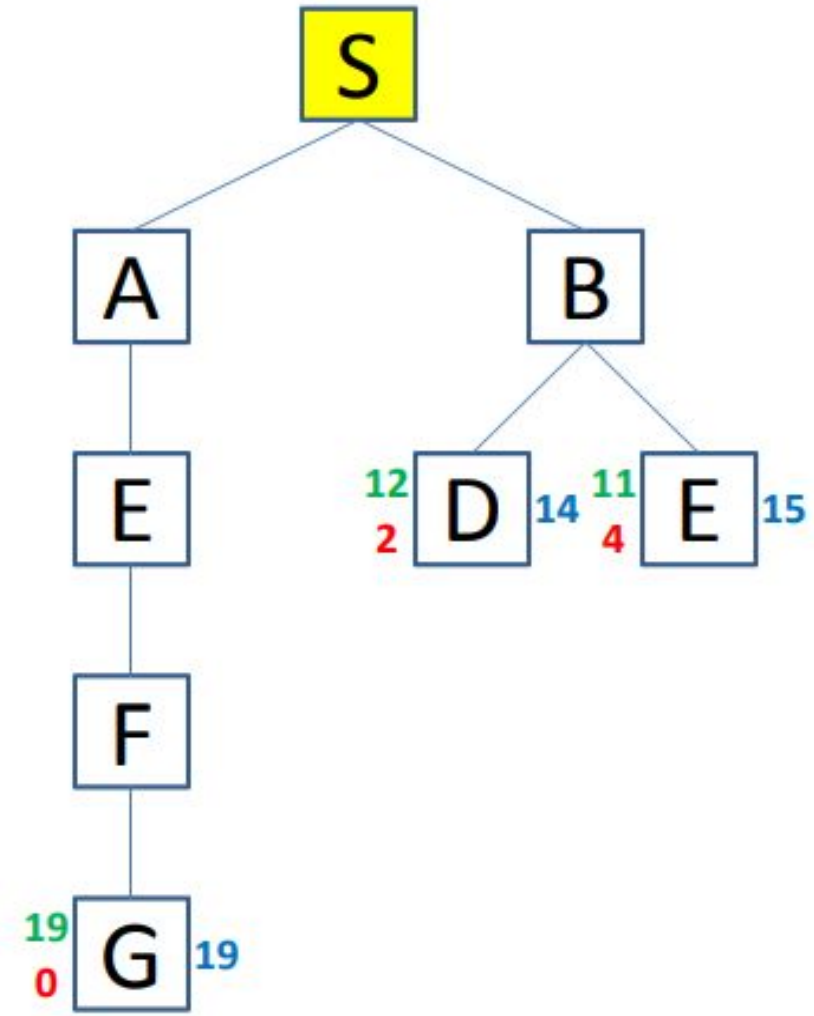


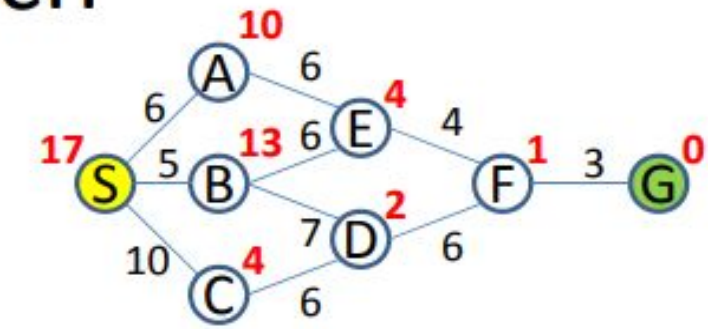


QUEUE:
SB
SAEFG
SCDF
SCDB

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

A* Search

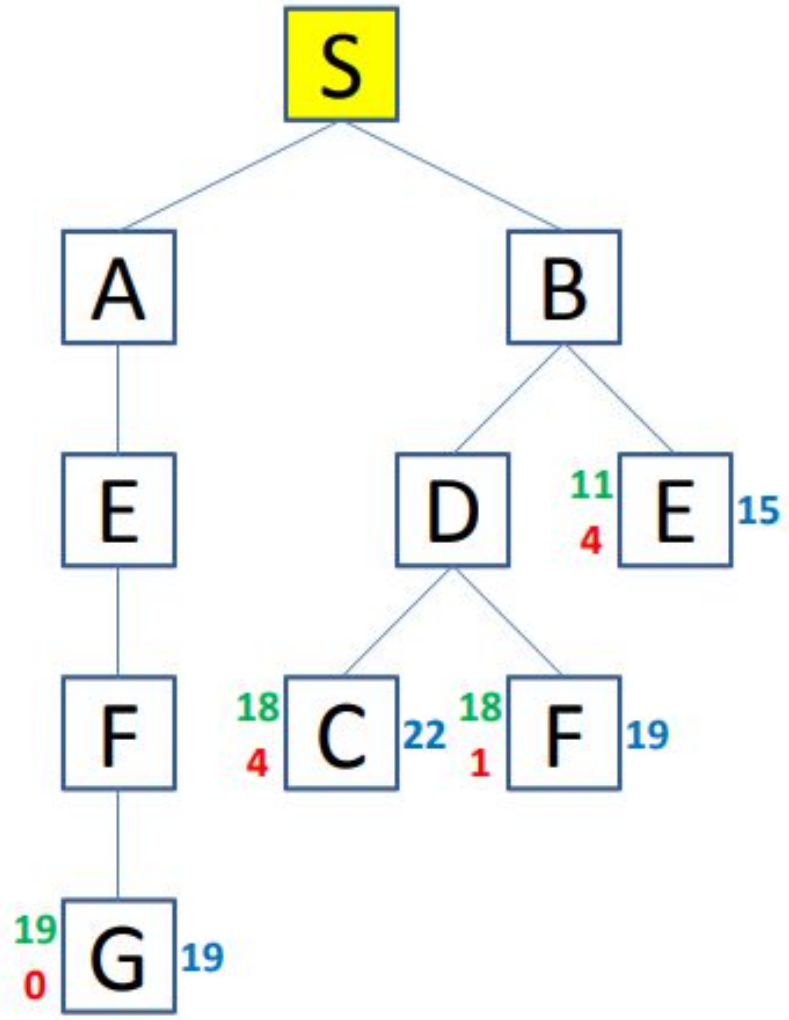


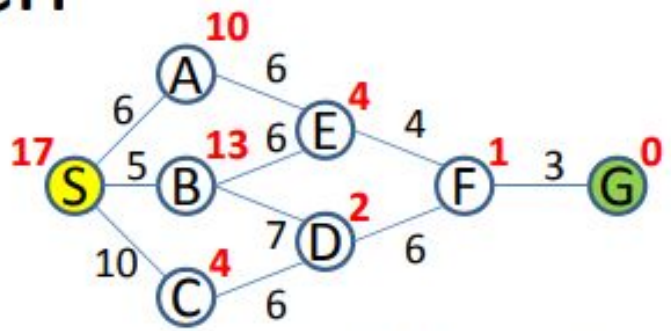


QUEUE:
SBD
SBE
SAEFG

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

A* Search

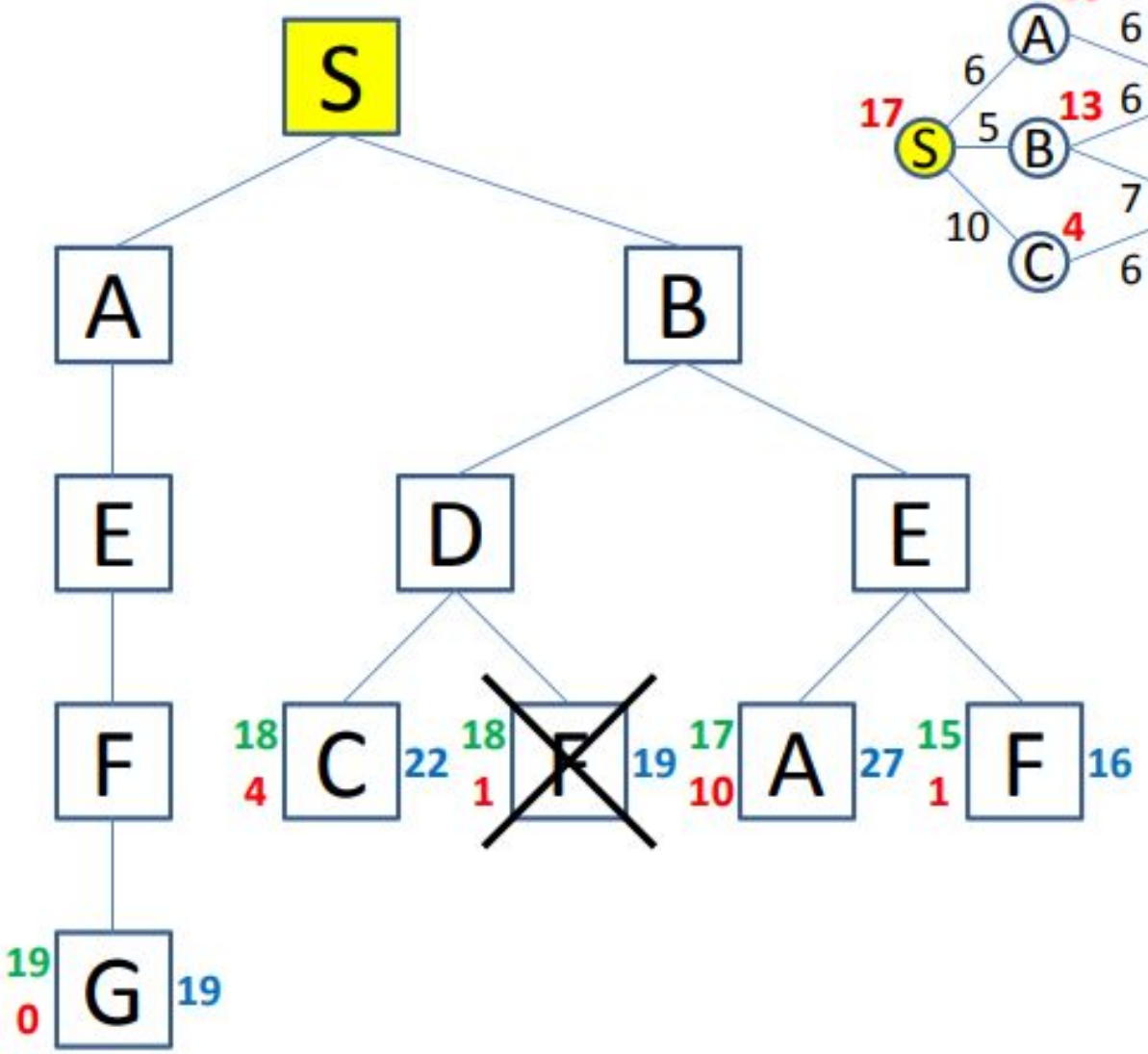


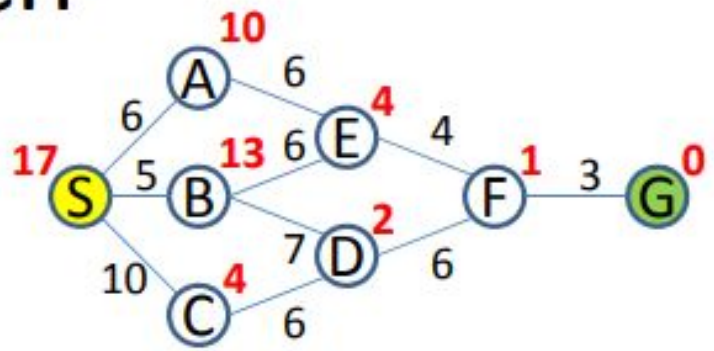


QUEUE:
SBE
SBDF
SAEFG
SBDC

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

A* Search

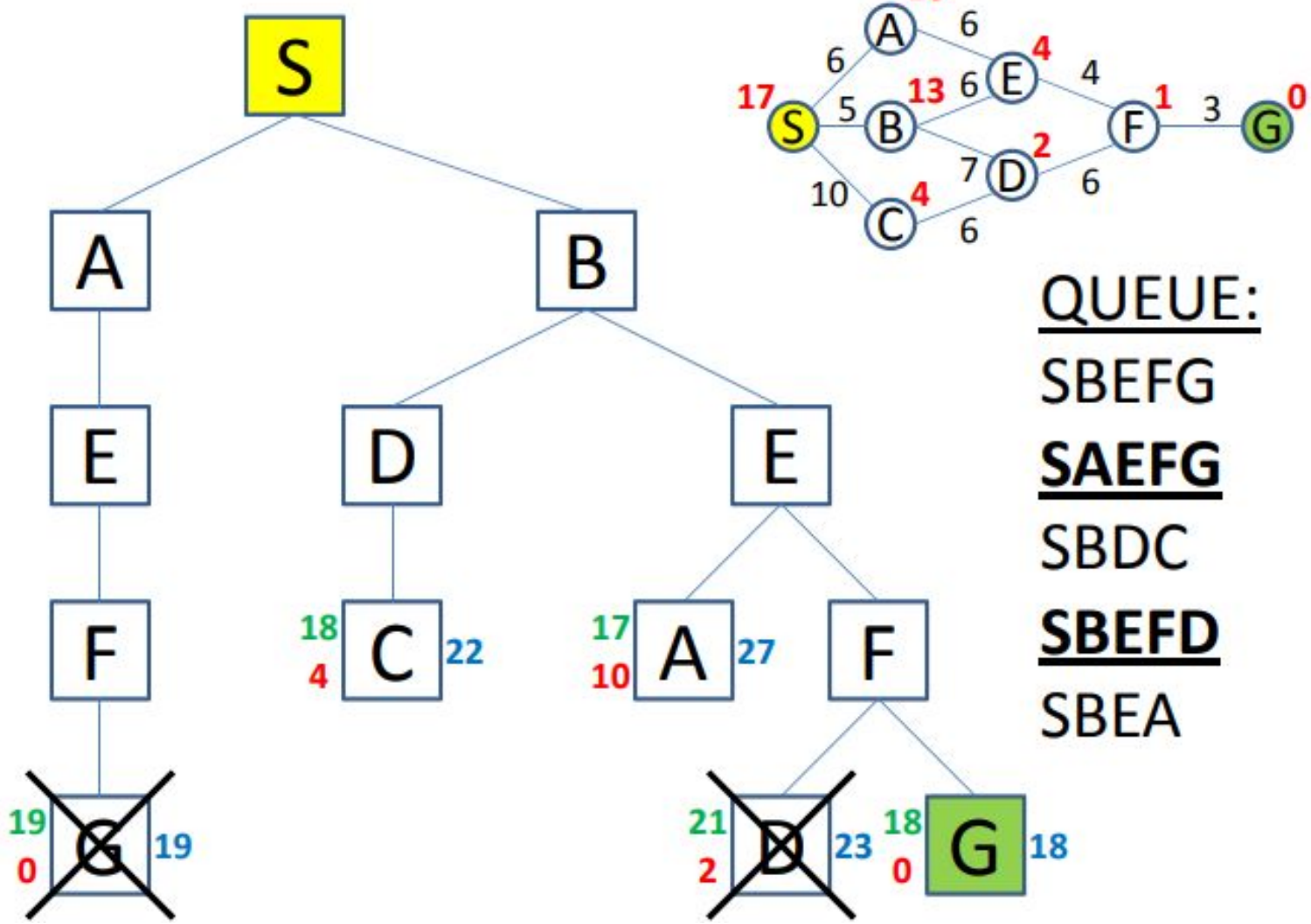




QUEUE:
SBEF
SAEFG
SBDF
SBDC
SBEA

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 O* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

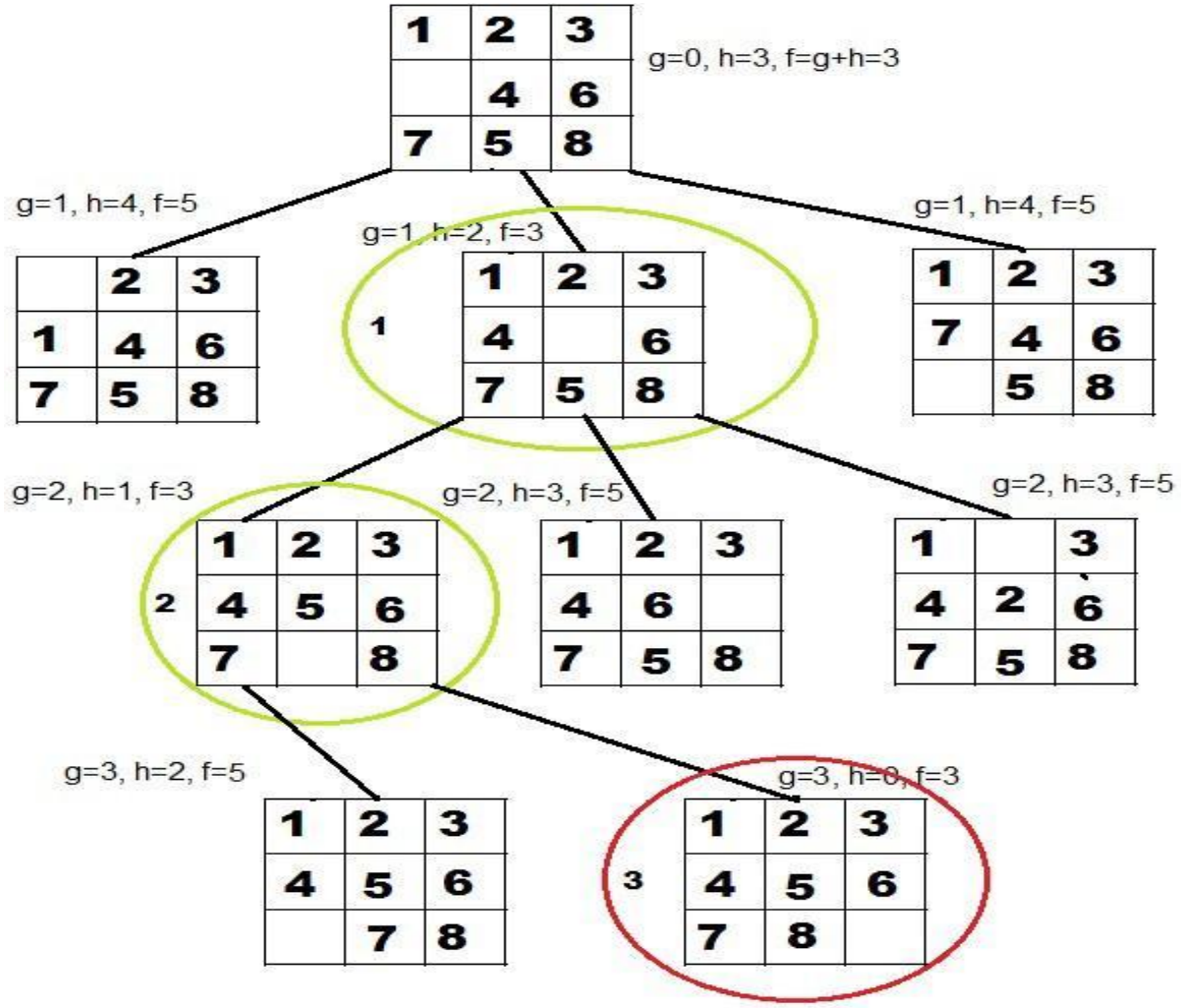
A* Search



Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal

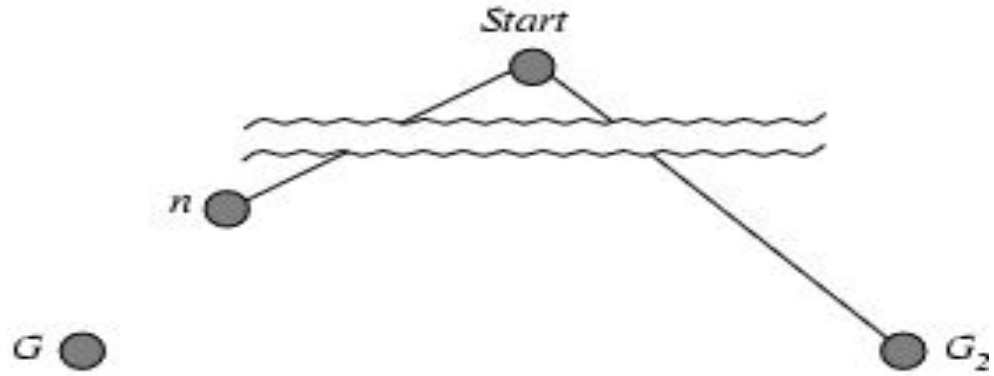
Puzzle problem using A*



Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

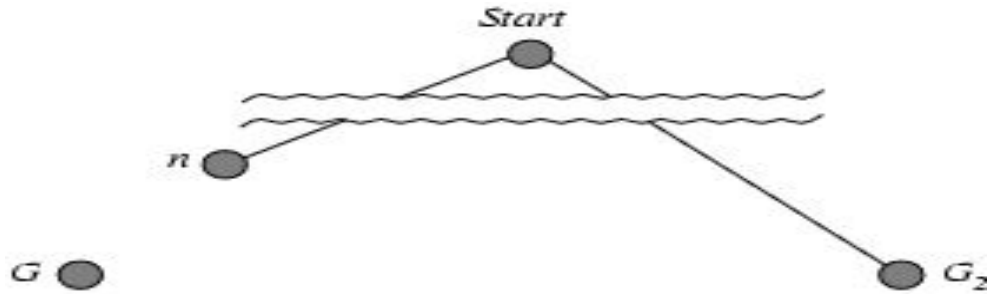
We want to prove:
f(n) < f(G2)
(then A* will prefer n over G2)



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



- $f(G_2) > f(G)$ copied from last slide
- $h(n) \leq h^*(n)$ since h is admissible (*under*-estimate)
- $g(n) + h(n) \leq g(n) + h^*(n)$ from above
- $f(n) \leq f(G)$ since $g(n)+h(n)=f(n)$ & $g(n)+h^*(n)=f(G)$
- $f(n) < f(G2)$ from top line.

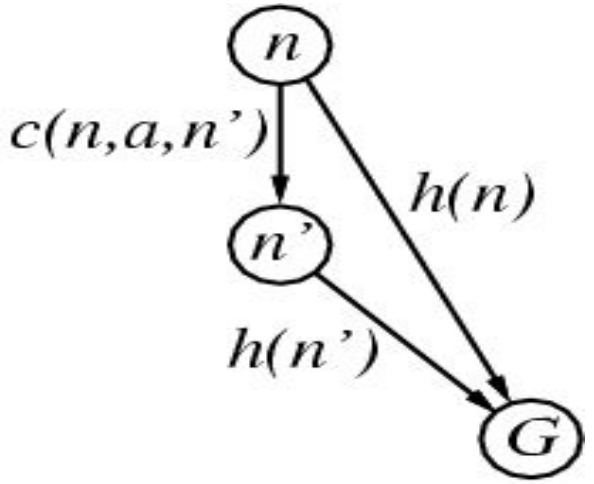
Hence: n is preferred over G2

Consistent heuristics

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

$$h(n) \leq c(n,a,n') + h(n')$$

It's the triangle inequality !



- If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) = f(n) \\ f(n') &\geq f(n) \end{aligned}$$

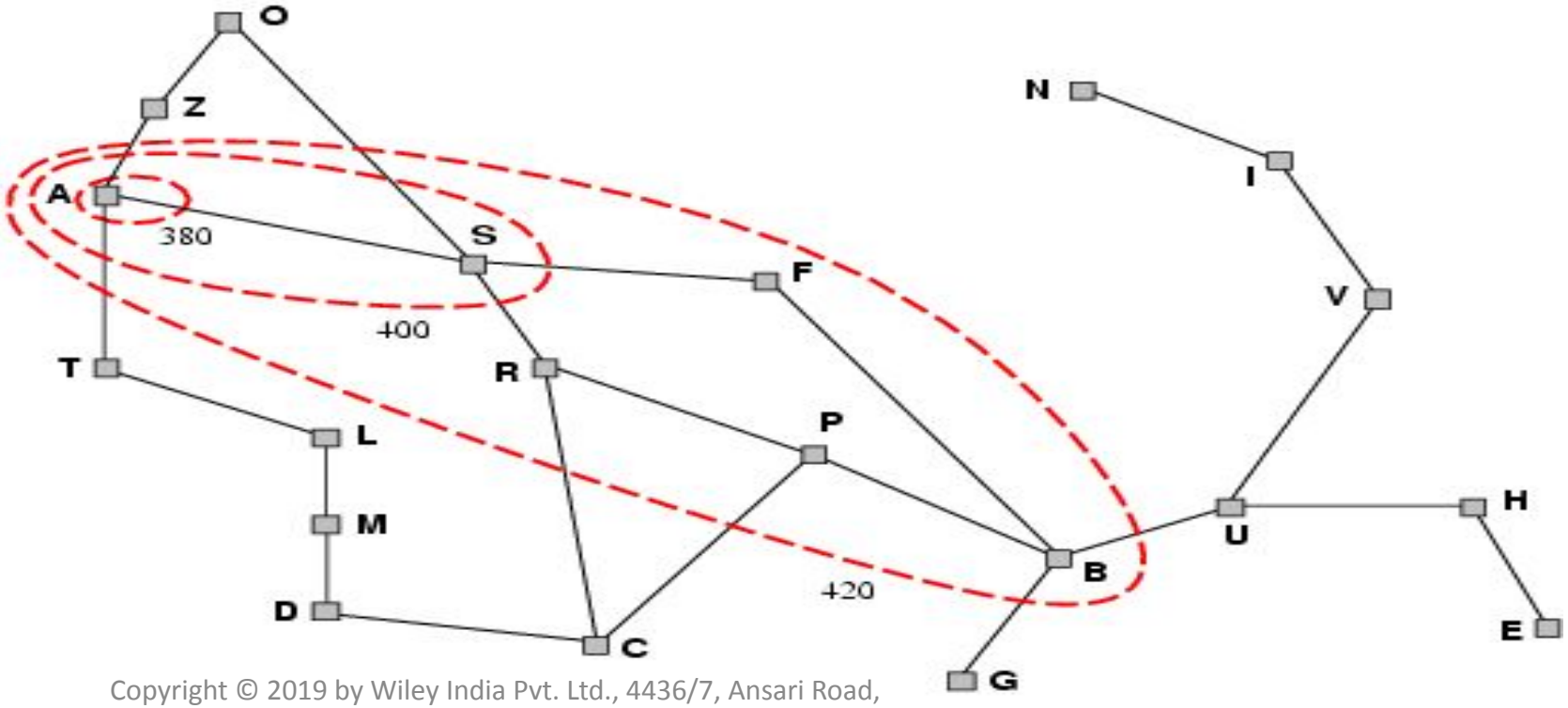
- i.e., $f(n)$ is non-decreasing along any path.

- Theorem:
  If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal

keeps all checked nodes in memory to avoid repeated states

Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* contains all nodes with $f \leq f_i$ where $f_i < f_{i+1}$



Properties of A*

- Complete? Yes (unless there are infinitely many nodes with f $\leq f(G)$, i.e. path-cost > ε)
- Time/Space? Exponential

  except if: b^d

  $$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

- Optimal? Yes

- *Optimally Efficient*: Yes (no algorithm with the same heuristic is guaranteed to expand fewer nodes)

And-Or Graph

- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

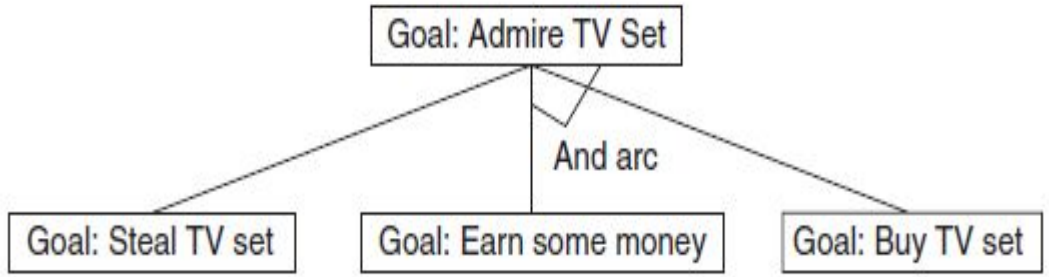


Figure shows AND–OR graph – an example

AO Algorithm*

1. Let G consists only to the node representing the initial state call this node INTT. Compute *h' (INIT).*

2. Until INIT is labelled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(a) Trace the marked arcs from INIT and select an unbounded node NODE.

(b) Generate the successors of NODE. if there are no successors then assign FUTILITY as *h'*

(NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancestor of NODE do the following:

• Add SUCCESSOR to graph G.

• If successor is not a terminal node, mark it solved and assign zero to its h' value.

• If successor is not a terminal node, compute it h' value.

(c) Propagate the newly discovered information up the graph by doing the following:

– Let S be a set of nodes that have been marked SOLVED. Initialise S to NODE. Until S is empty repeat the following procedure;

– Select a node from S call if CURRENT and remove it from S.

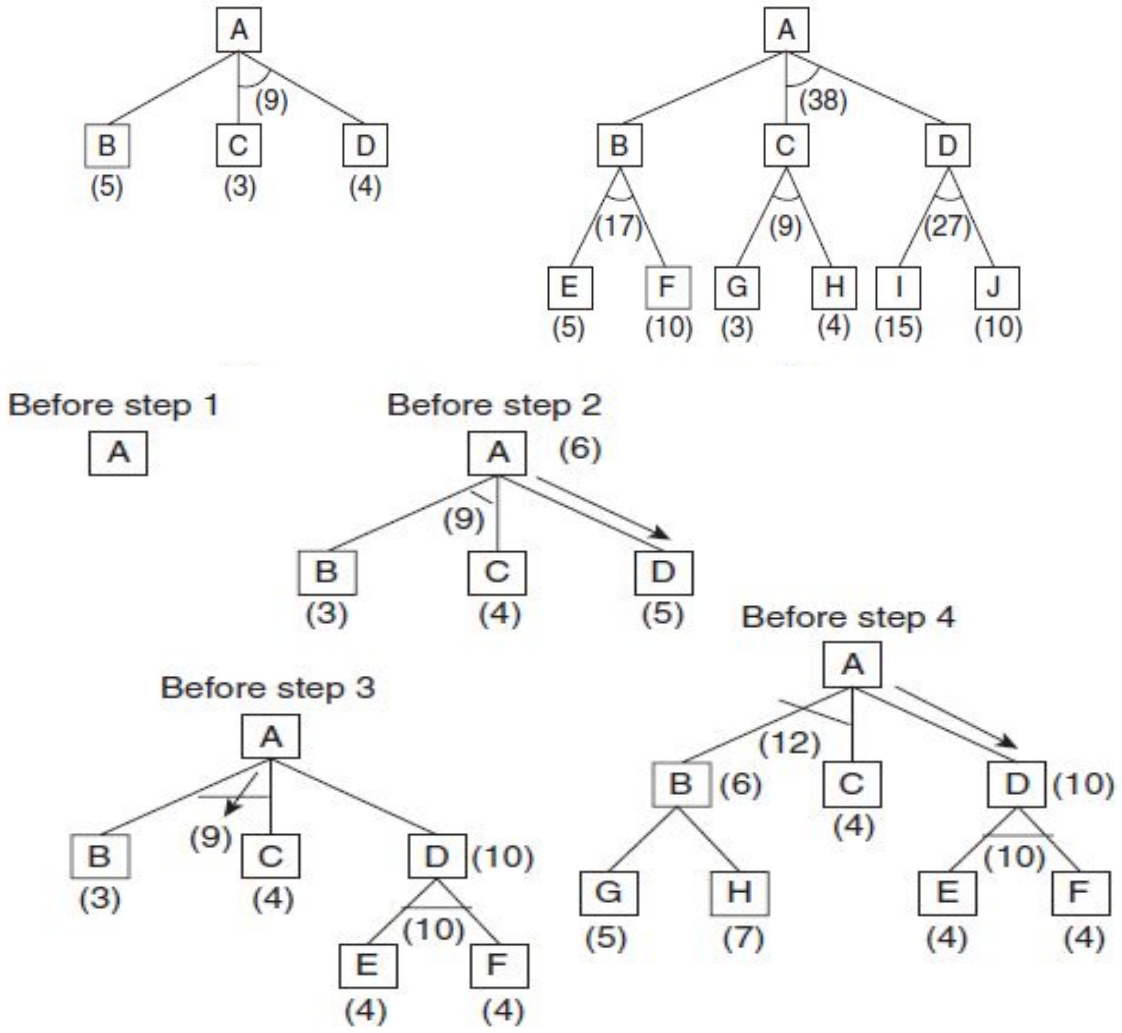
– Compute *h' of each of the arcs emerging from CURRENT. Assign minimum h' t CURRENT.*

– Mark the minimum cost path as the best out of CURRENT.

-Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labelled SOLVED.

- If CURRENT has been marked SOLVED or its *h' has just* changed, its new status must be propagating backwards up the graph. Hence, all the ancestors of CURRENT are added to S.

*Example of AO**



Thus, we can see that to search an AND–OR graph, the following three things must be done.

1. Traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.

2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer *f '* (cost of the remaining distance) for each of them.

3. Change the *f ' estimate of the newly expanded node to reflect the new information produced by* its successors. Propagate this change backward through the graph. Decide which of the current best path.

Difference between A* and AO* Algorithm

A* algorithm

1. a* is a computer algorithm which is used in pathfinding and graph traversal. It is used in the process of plotting an efficiently directed path between a number of points called nodes.
2. In a* algorithm you traverse the tree in depth and keep moving and adding up the total cost of reaching the cost from the current state to the goal state and add it to the cost of reaching the current state.

AO* algorithm

1. In ao* algorithm you follow a similar procedure but there are constraints traversing specific paths.
2. When you traverse those paths, cost of all the paths which originate from the preceding node are added till that level, where you find the goal state regardless of the fact whether they take you to the goal state or not.

IDA* Algorithm

f-bound ⟵ f(S)

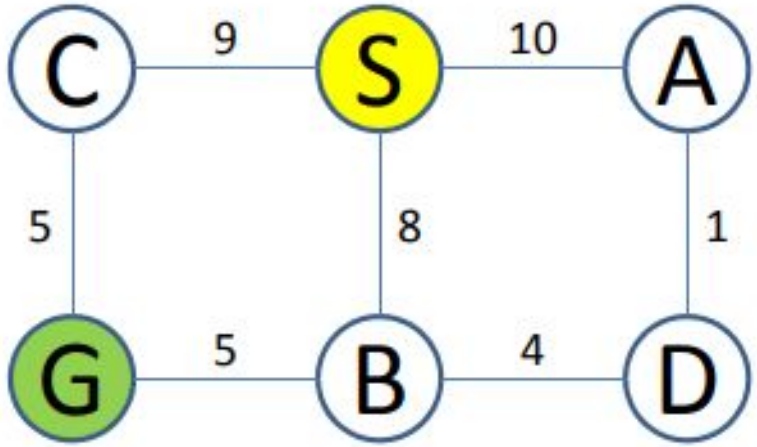
Algorithm:

- **WHILE** (goal is not reached) **DO**
  - ***f-bound ⟵ f-limitted_search(f-bound)***
    - Perform **f-limited search** with ***f-bound***

f-limitted Search Algorithm

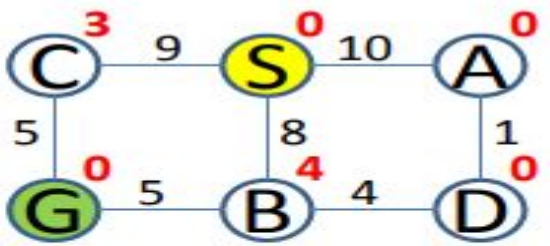
- ***Input:***
  - QUEUE ⟵ Path only containing root
  - f-bound ⟵ Natural number
  - f-new ⟵ ∞
- ***Algorithm:***
  - **WHILE** (QUEUE not empty && goal not reached) **DO**
    - Remove **first path** from QUEUE
    - Create paths to children
    - Reject paths with loops
    - Add paths with **f(path) ≤ f-bound** to **front** of QUEUE *(depth-first)*
    - f-new ⟵ minimum({f-new} ∪ {f(P) | P is rejected path})
  - **IF** goal reached **THEN** success **ELSE** report f-new

Perform the IDA* Algorithm on the following figure.

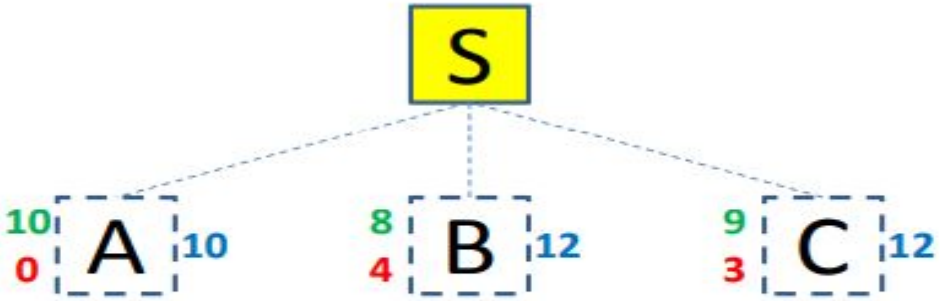


	S	A	B	C	D	G
heuristic	0	0	4	3	0	0

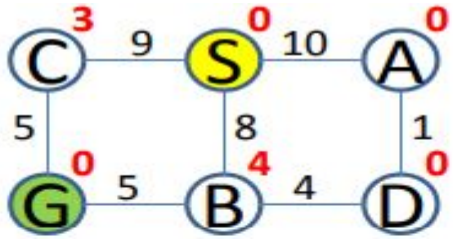
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search



f-bound = 0
f-new = ∞

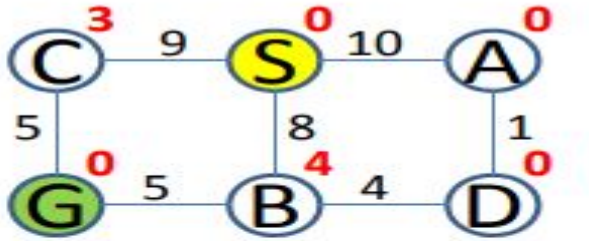


Children are explored depth-first!

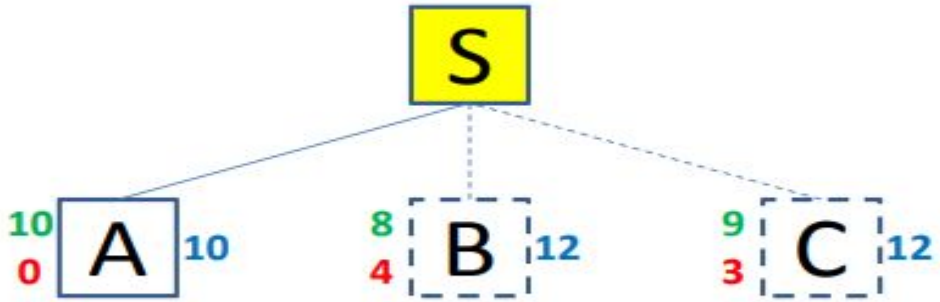


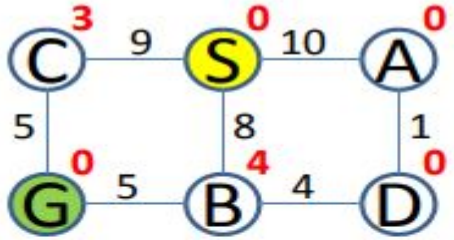
f-bound = 0
f-new = 10

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search



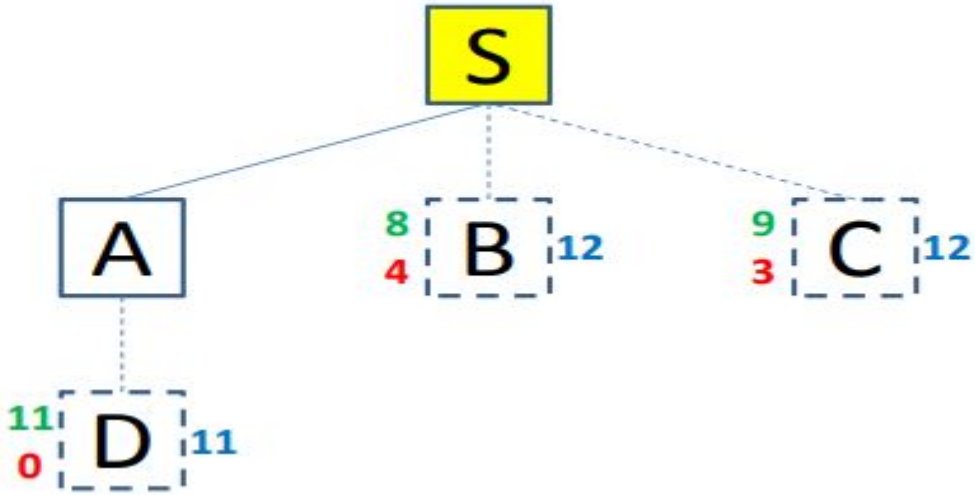
f-bound = 10
f-new = ∞

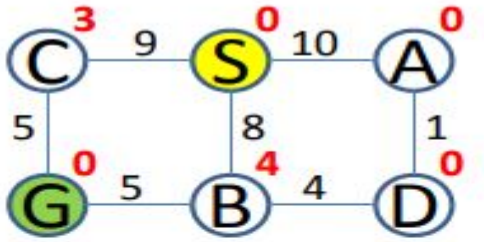




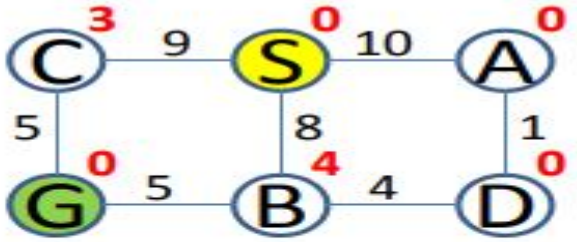
f-bound = 10
f-new = 12

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search



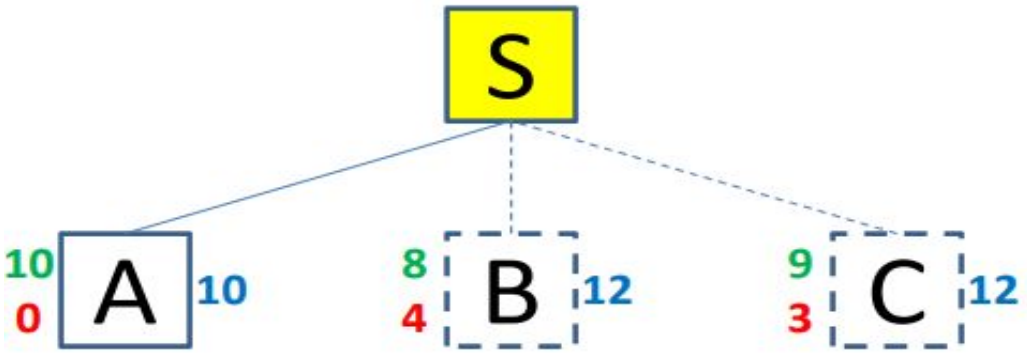


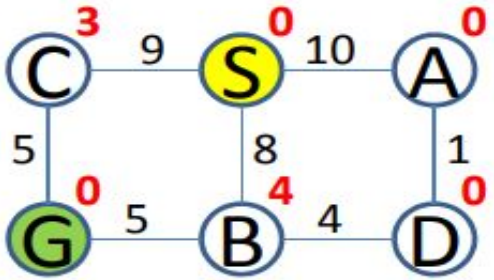
f-bound = 10
f-new = 11



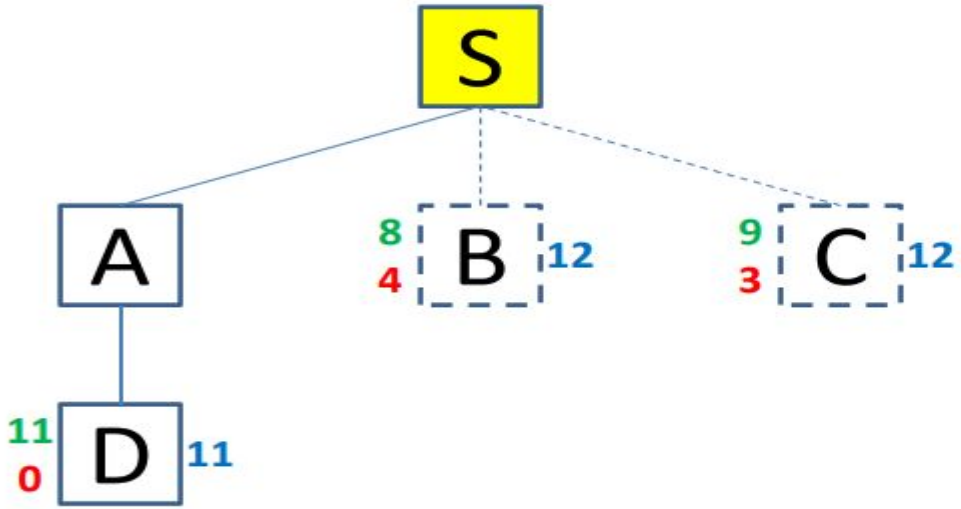
f-bound = 11
f-new = ∞

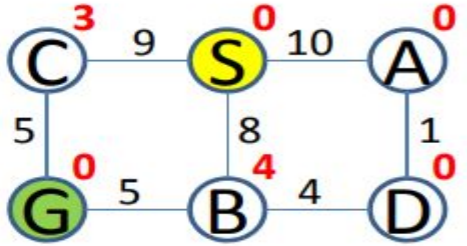
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search





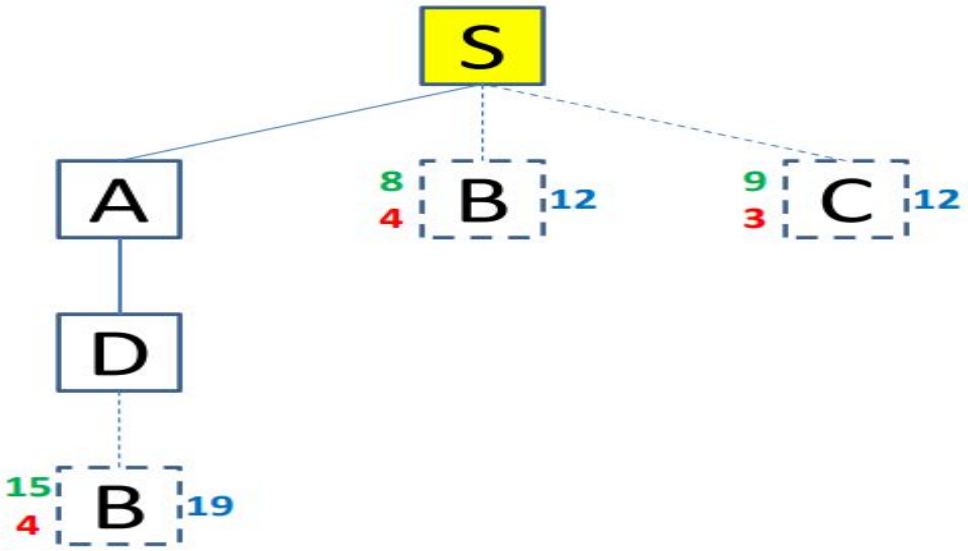
f-bound = 11
f-new = 12

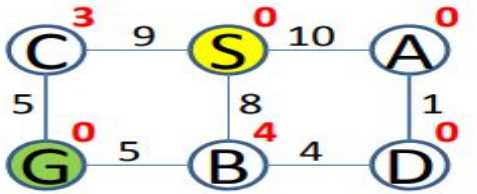




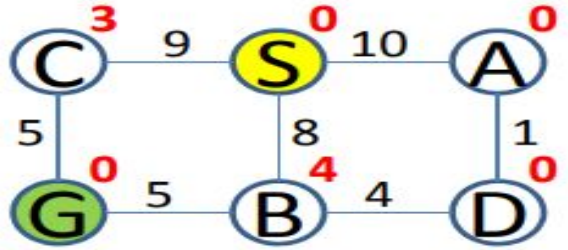
f-bound = 11
f-new = 12

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search



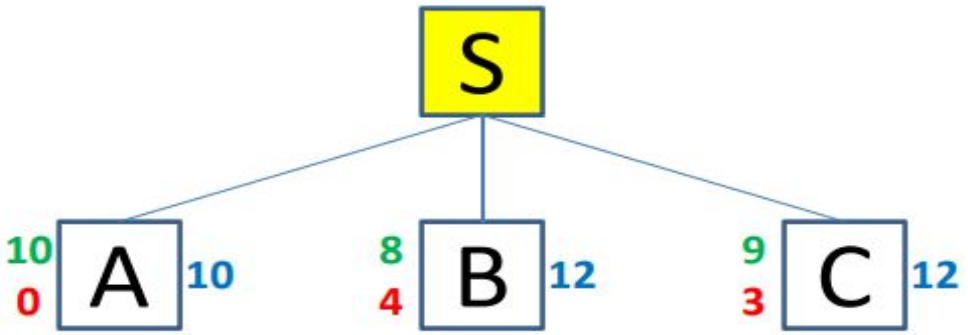


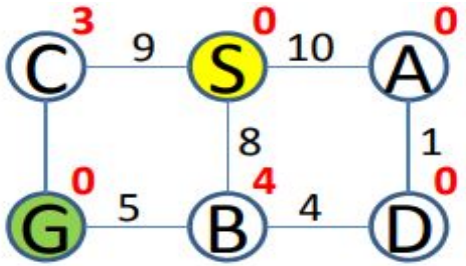
f-bound = 11
f-new = 12



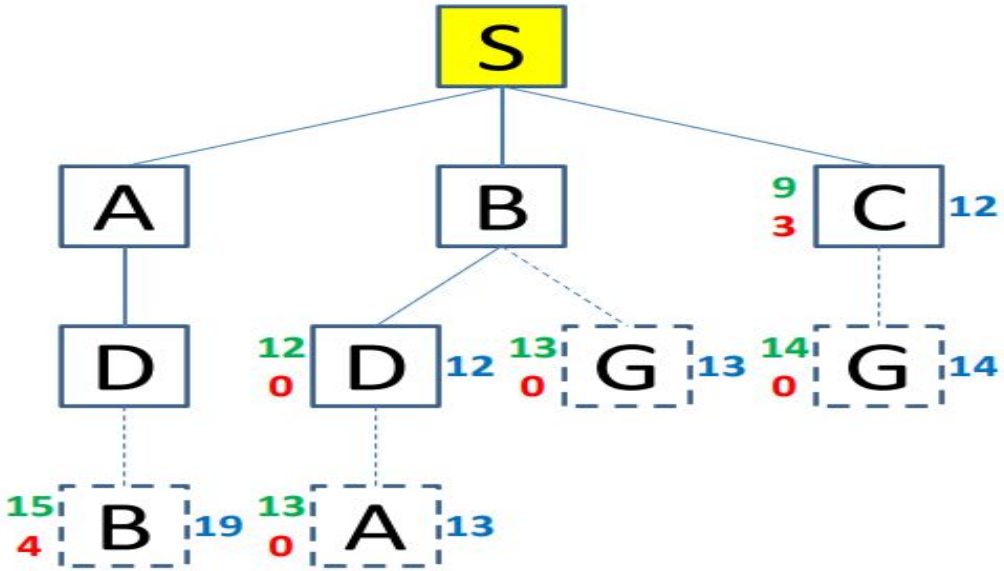
f-bound = 12
f-new = ∞

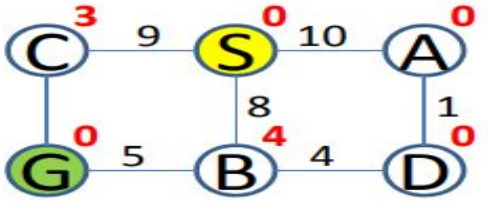
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search





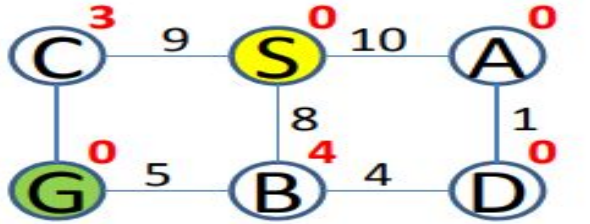
f-bound = 12
f-new = ∞



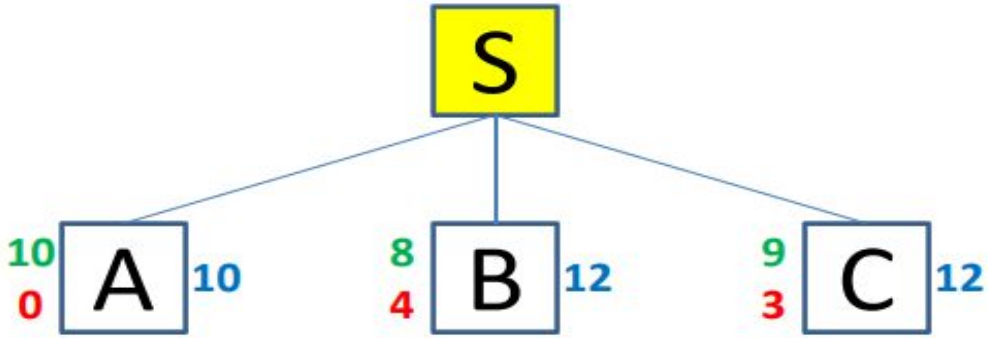


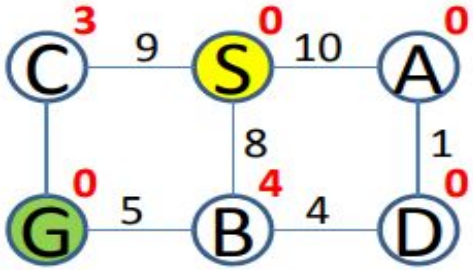
f-bound = 12
f-new = 13

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search



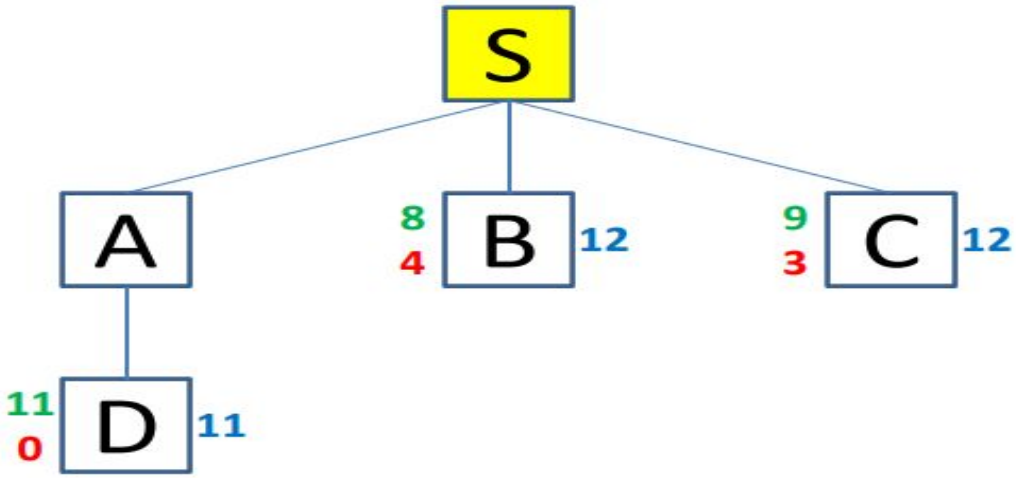
f-bound = 13
f-new = ∞

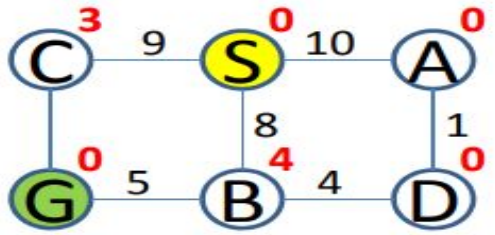




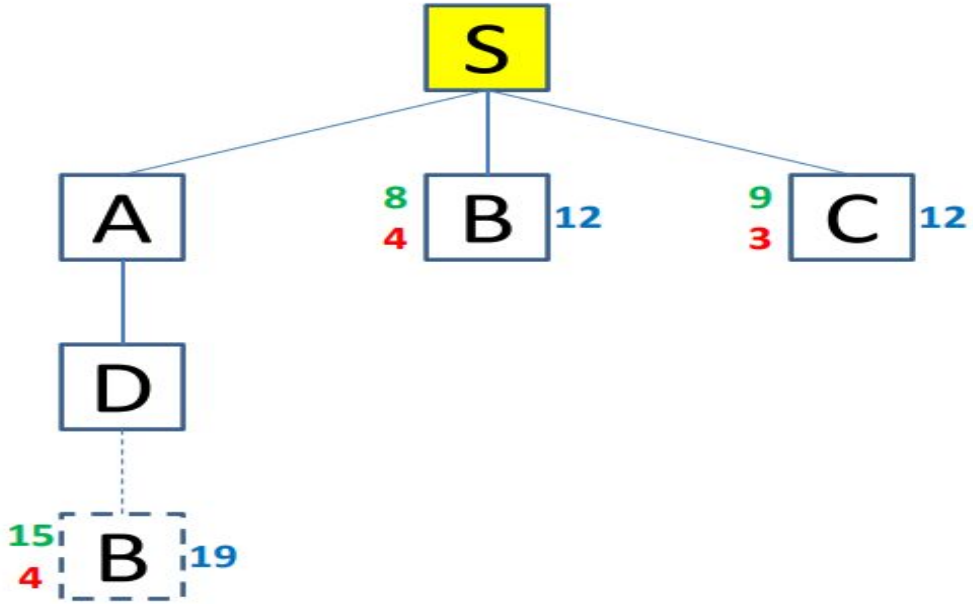
f-bound = 13
f-new = ∞

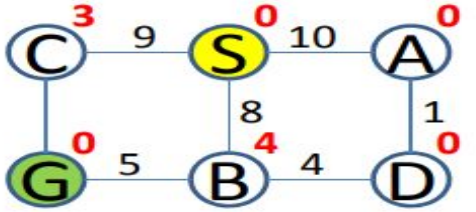
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search





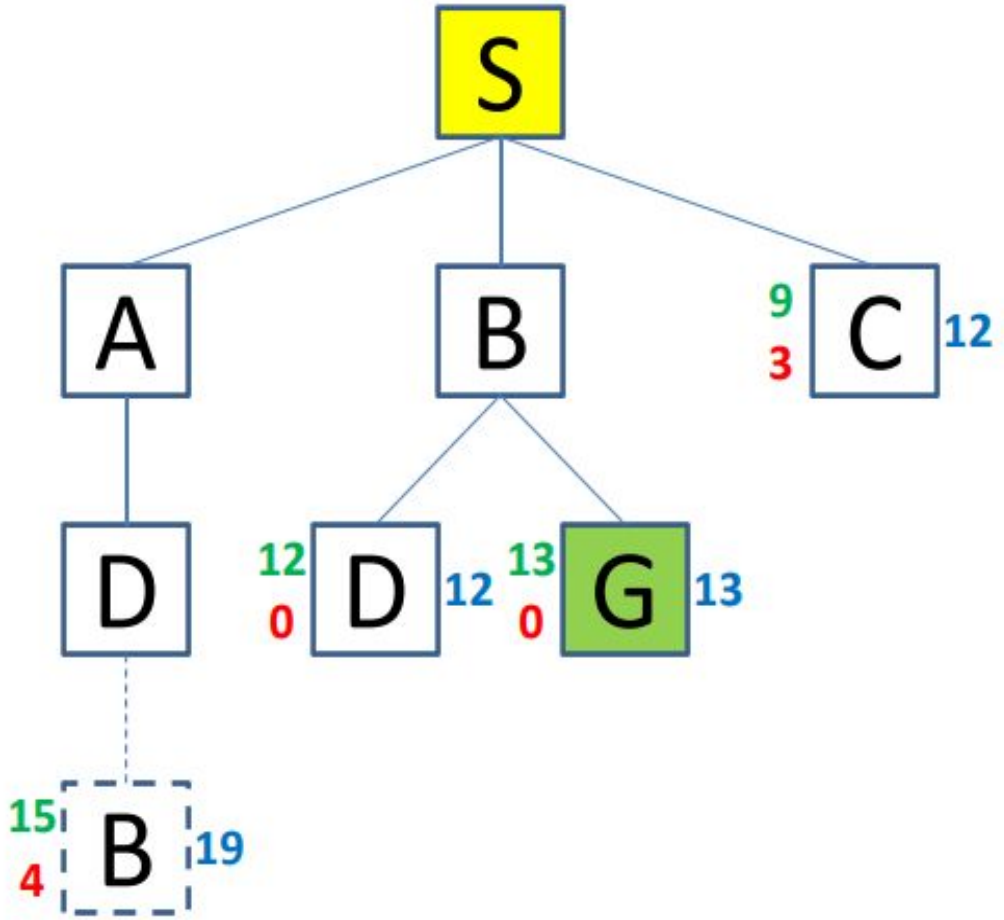
f-bound = 13
f-new = ∞

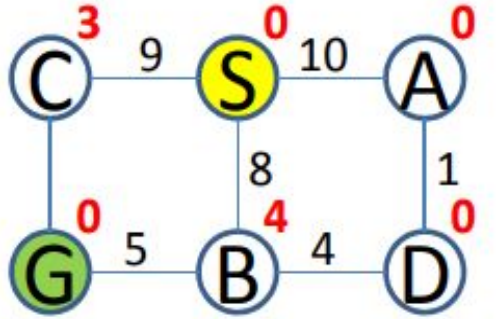




f-bound = 13
f-new = 19

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search





f-bound = 13

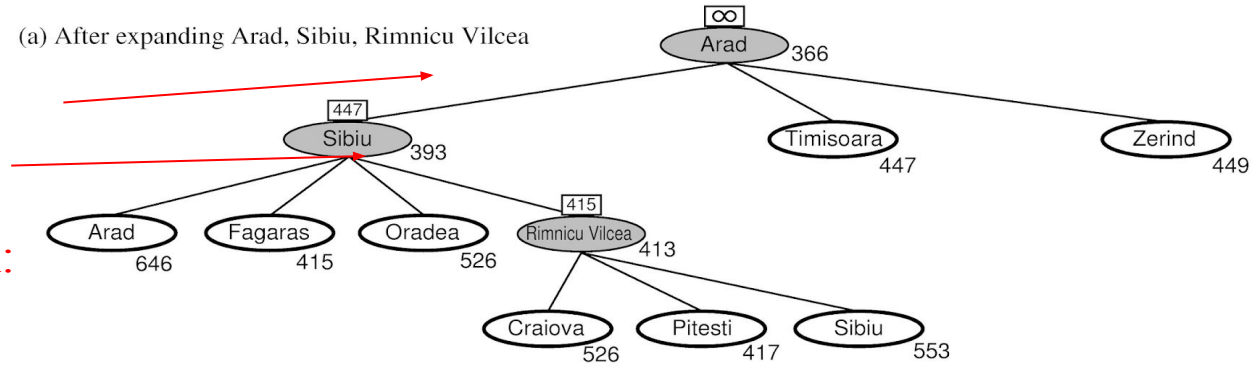
f-new = 19

Memory Bounded Heuristic Search: Recursive BFS

- How can we solve the memory problem for A* search?
- Idea: Try something like depth first search, but let's not forget everything about the branches we have partially explored.
- We remember the best f-value we have found so far in the branch we are deleting.

RBFS:

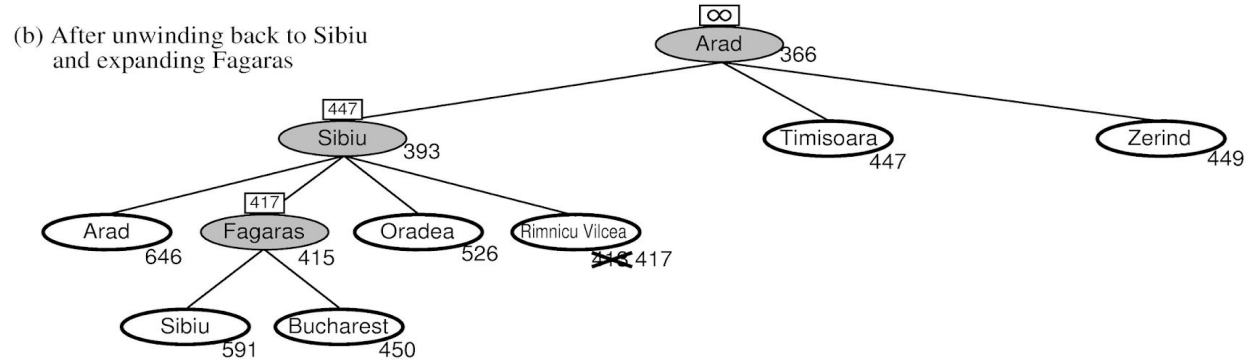
best alternative over fringe nodes, which are not children: *do I want to back up?*

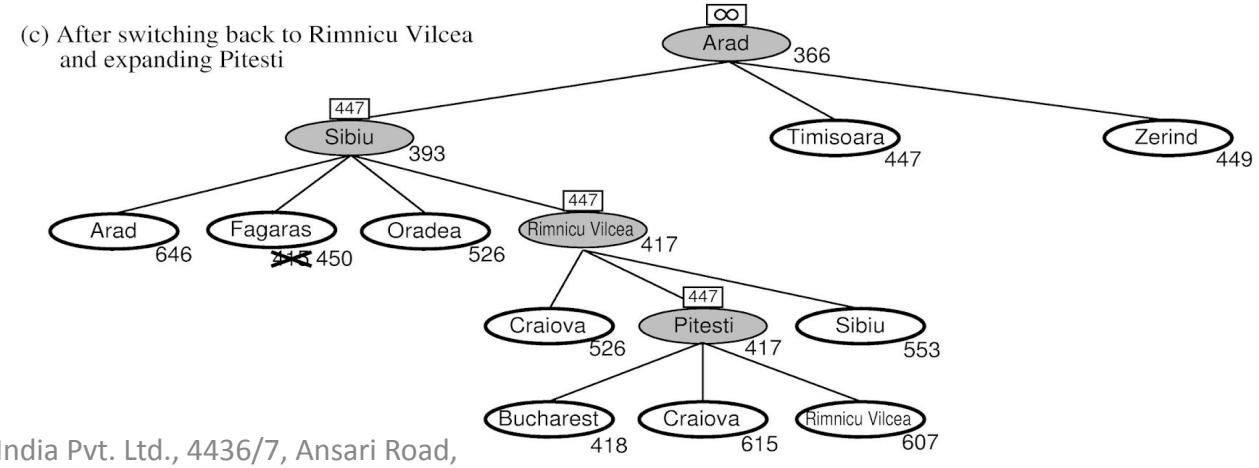


RBFS changes its mind very often in practice.

This is because the f=g+h become more accurate (less optimistic) as we approach the goal. Hence, higher level nodes have smaller f-values and will be explored first.

Problem: We should keep in memory whatever we can.

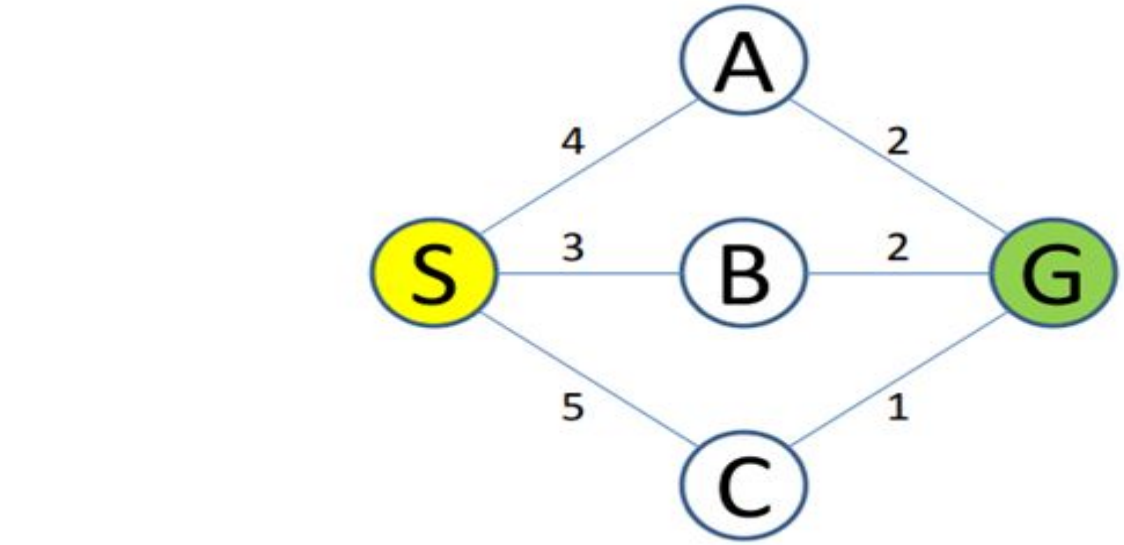




Simple Memory Bounded A*

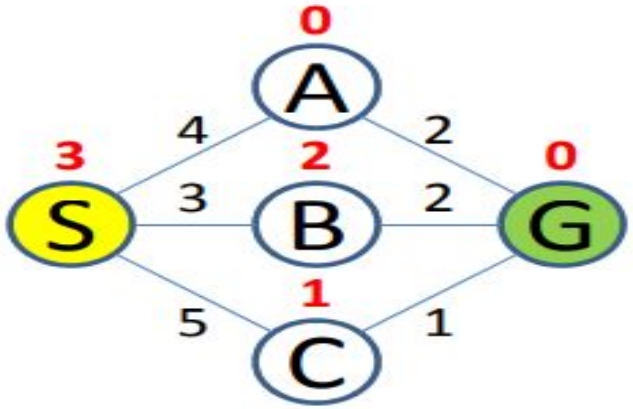
- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we first delete the oldest nodes first.
- simple-MBA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

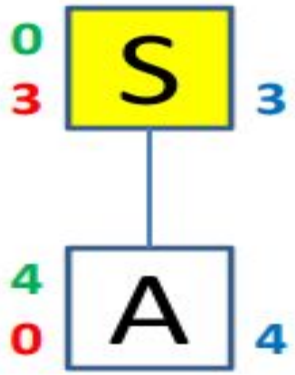
Perform SMA* (memory: 3 nodes) on the following figure.

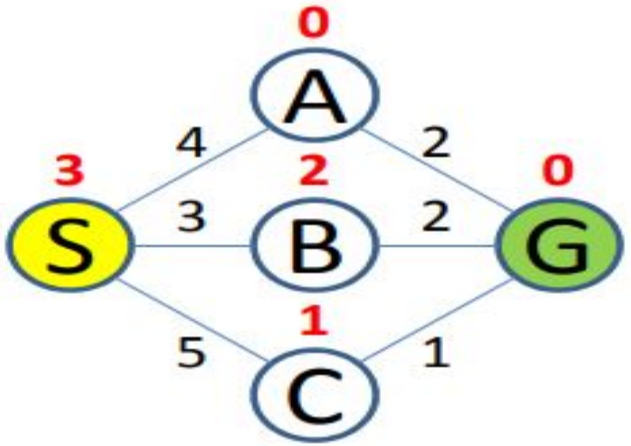


	S	A	B	C	G
heuristic	3	0	2	1	0

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

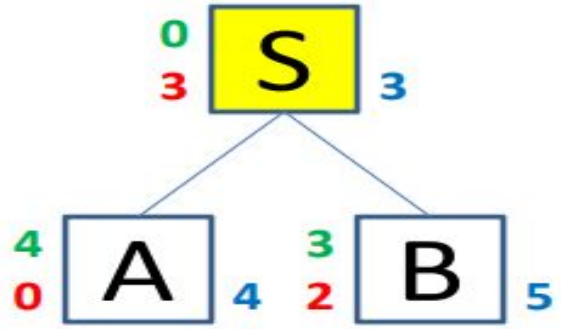


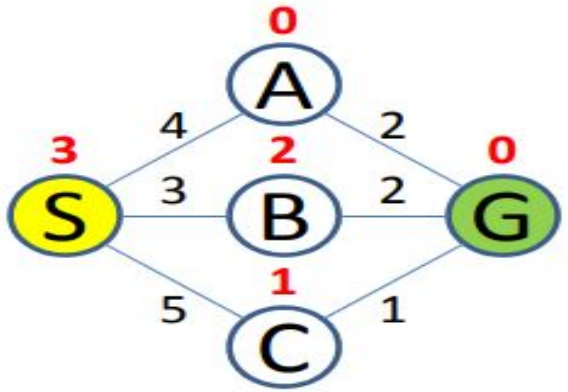




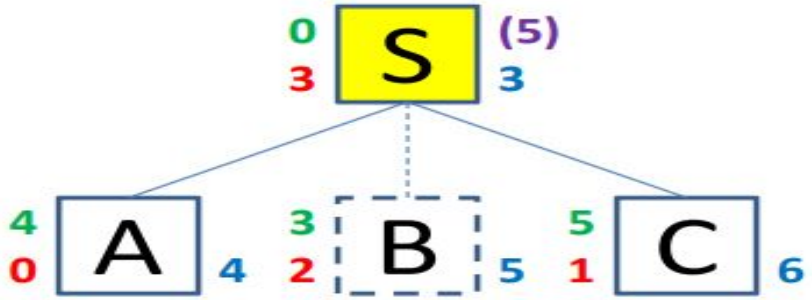
Generate children
(One by one)

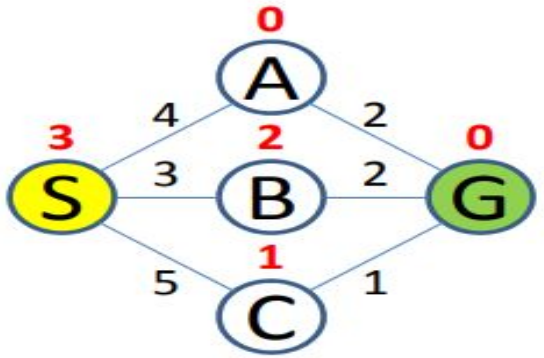
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search





Generate children
(One by one)

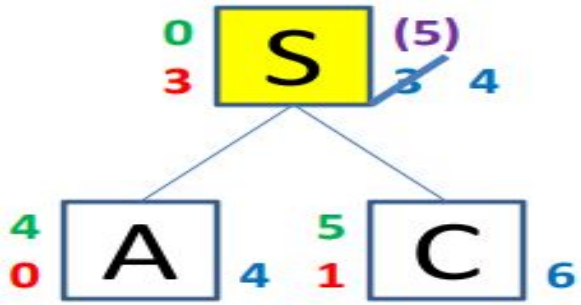


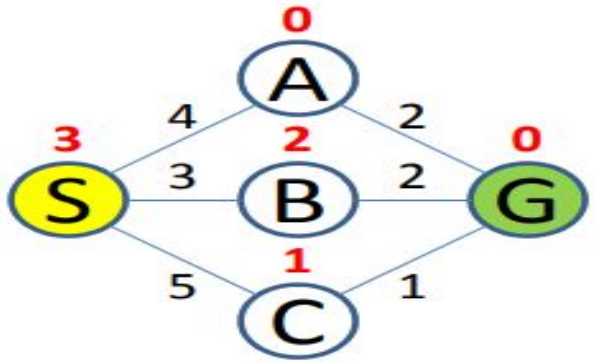


Generate children
(One by one)

Memory full

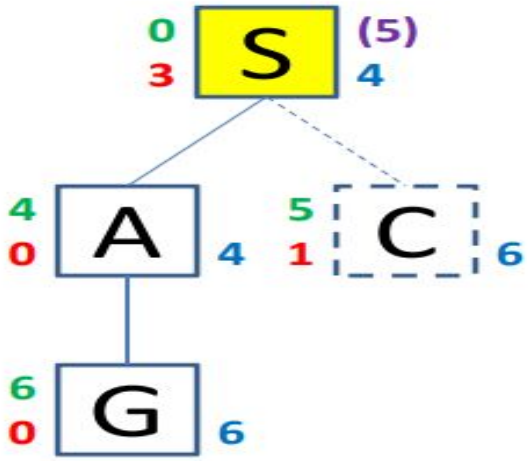
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

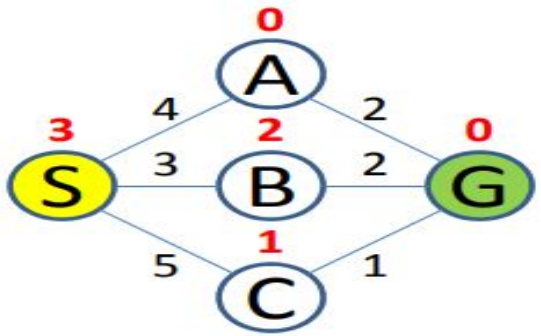




All children are explored

Adjust f-values

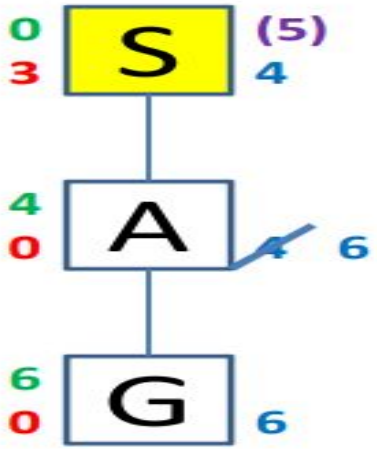


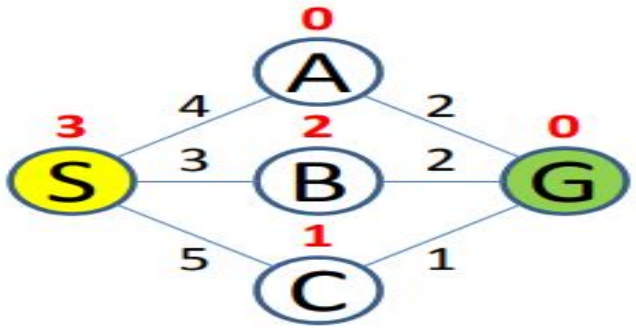


Generate children (One by one)

Memory full

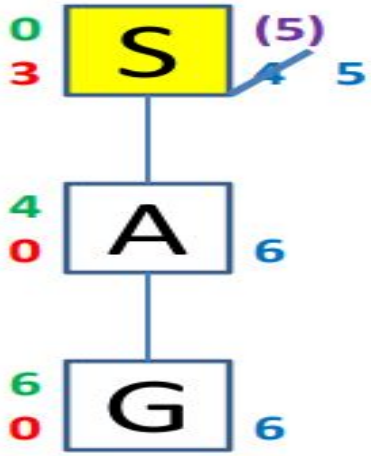
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

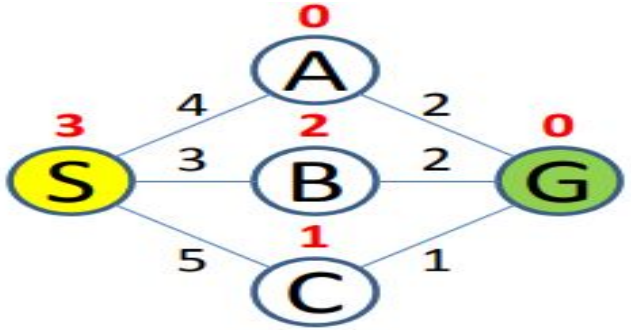




All children are explored

Adjust f-values

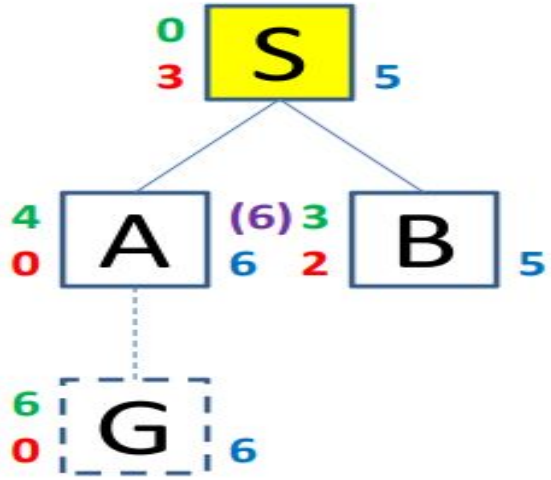


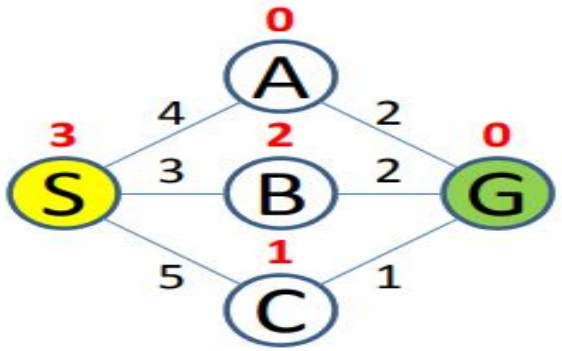


All children are explored (update)

Adjust f-values

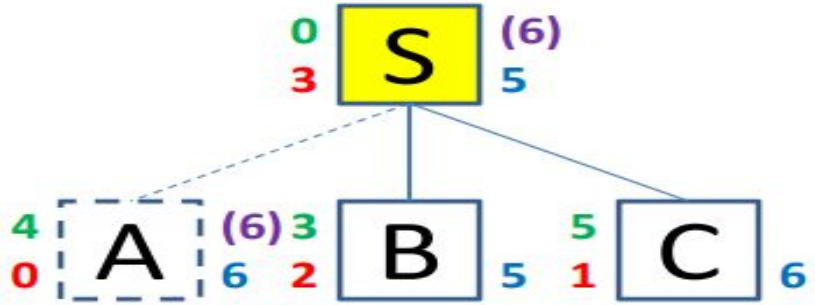
5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search

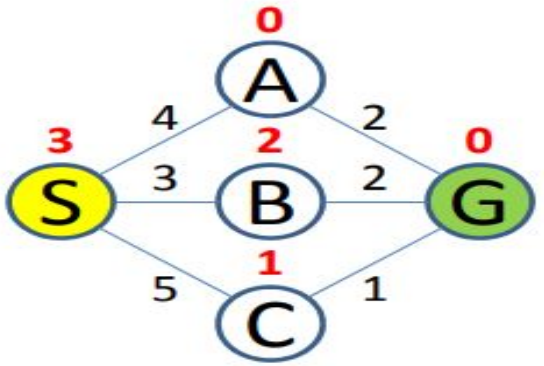




Generate children
(One by one)

Memory full

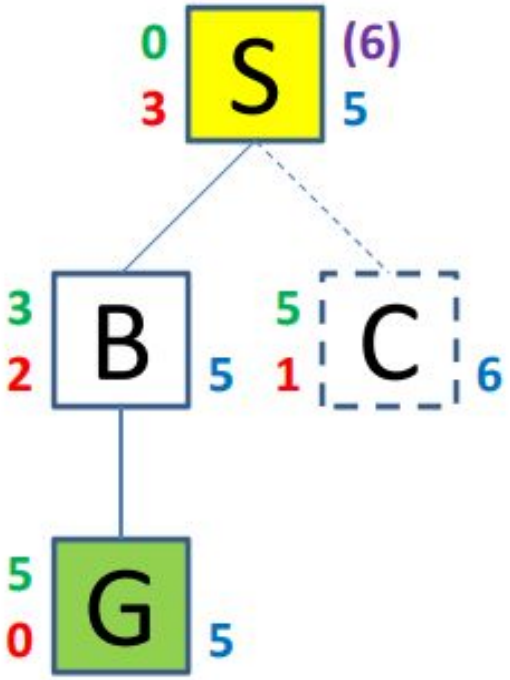


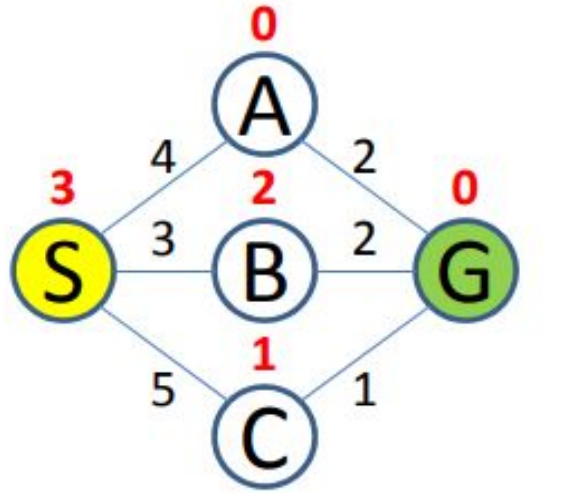


Generate children
(One by one)

Memory full

5.1 Introduction
5.2 Hill Climbing
5.3 Best-first Search (Greedy Search)
5.4 A* Search
5.5 AO* Search: (AND–OR) Graph
5.6 Memory Bounded Heuristic Search
5.6.1 Iterative Deepening A*
5.6.2 Recursive BFS
5.6.3 Simplified Memory Bounded A* (SMA*)
5.7 Simulated Annealing Search
5.8 Local Beam Search
5.9 Branch and Bound Search





Generate children
(One by one)

Memory full

Annealing

- Annealing is a thermal process for obtaining low energy states of a solid in a heat bath.
- The process contains two steps:
  - Increase the temperature of the heat bath to a maximum value at which the solid melts.
  - Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid. Ground state is a minimum energy state of the solid.
- The ground state of the solid is obtained only if the maximum temperature is high enough and the cooling is done slowly.

Simulated Annealing

- The process of annealing can be simulated with the Metropolis algorithm, which is based on Monte Carlo techniques.
- We can apply this algorithm to generate a solution to combinatorial optimization problems assuming an analogy between them and physical many-particle systems with the following equivalences:
  - Solutions in the problem are equivalent to states in a physical system.
  - The cost of a solution is equivalent to the "energy" of a state.

Simulated Annealing

- To apply simulated annealing with optimization purposes we require the following:
  - A successor function that returns a "close" neighboring solution given the actual one. This will work as the "disturbance" for the particles of the system.
  - A target function to optimize that depends on the current state of the system. This function will work as the energy of the system.
- The search is started with a randomized state. In a polling loop we will move to neighboring states always accepting the moves that decrease the energy while only accepting bad moves accordingly to a probability distribution dependent on the "temperature" of the system.

Simulated Annealing

- Decrease the temperature slowly, accepting less bad moves at each temperature level until at very low temperatures the algorithm becomes a greedy hill-climbing algorithm.
  - The distribution used to decide if we accept a bad movement is know as Boltzman distribution.

$$P(\gamma) = \frac{e^{-E_\gamma/T}}{Z(T)},$$

$$Z(T) = \sum_{\gamma'} e^{-E_{\gamma'}/T},$$

- This distribution is very well known is in solid physics and plays a central role in simulated annealing. Where γ is the current configuration of the system, E_γ is the energy related with it, and Z is a normalization constant.

Simulated Annealing: the code

function Simulated-Annealing (*problem, schedule)*
returns a solution state
inputs: *problem, a problem*
schedule, a mapping from time to "temperature"
local variables: *current, a node next, a node T, the temperature*
current ← *MakeNode (RandomState[problem])*
for *t* ← *1 to* ∞ *do*
T ← *schedule [t]*
if *T = 0 then return current*
next ← *a randomly selected successor of current*
ΔE ← *Value [next] - Value [current]*
if *ΔE > 0 then current* ← *next*
else *current* ← *next only with probability eΔE/T*

Practical Issues with simulated annealing

- The cost function should be fast it is going to be called "millions" of times.
- The best is if we just have to calculate the deltas produced by the modification instead of traversing through all the state.
- This is dependent on the application.

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)

Example: *n*-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Note that a state cannot be an incomplete configuration with m<n queens

Branch and Bound Search

Branch and Bound is an algorithmic technique which finds the optimal solution by keeping the best solution found so far. If partial solution can't improve on the best it is abandoned, by this method the number of nodes which are explored can also be reduced. It also deals with the optimization problems over a search that can be presented as the leaves of the search tree. The usual technique for eliminating the sub trees from the search tree is called pruning. For Branch and Bound algorithm we will use stack data structure.

- **Step 1:** Traverse the root node.
- **Step 2:** Traverse any neighbour of the root node that is maintaining least distance from the root node.
- **Step 3:** Traverse any neighbour of the neighbour of the root node that is maintaining least distance from the root node.