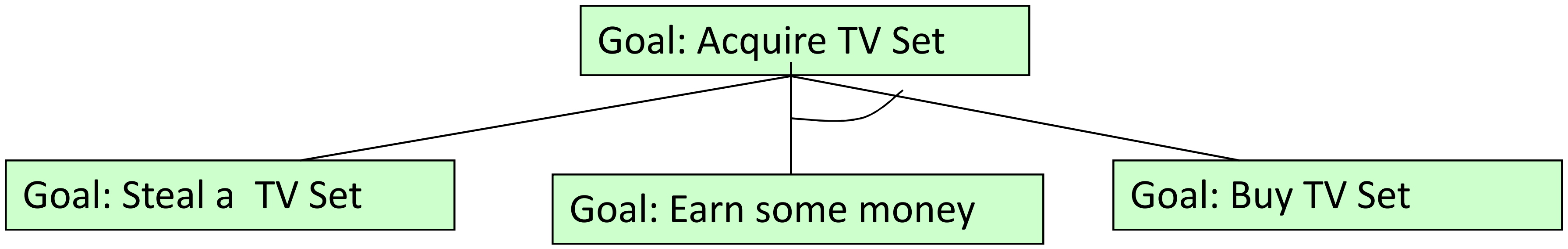
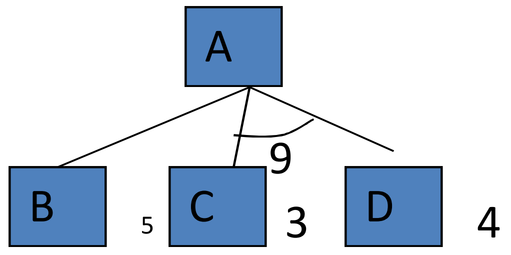
Problem Reduction
Branch and Bound

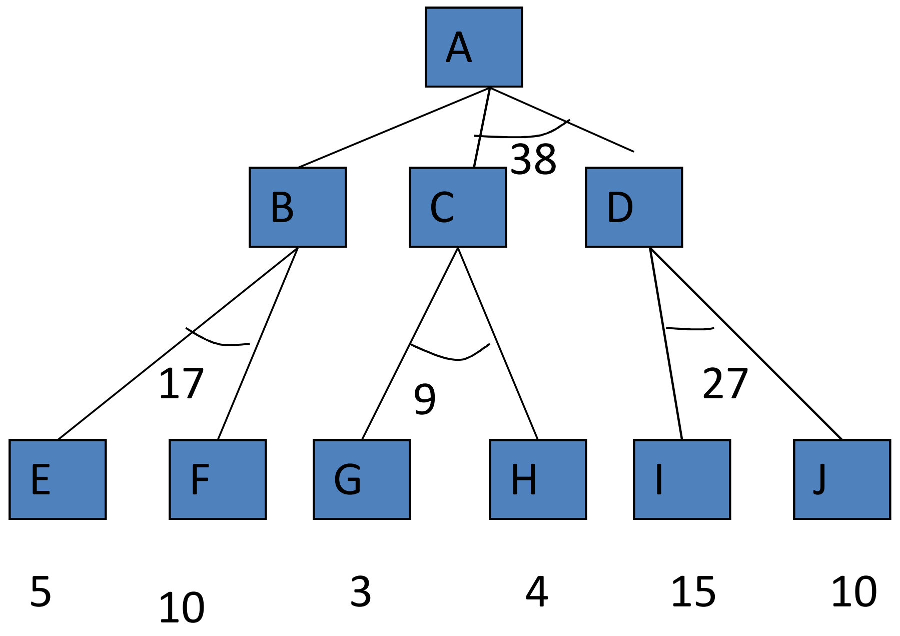
AND-OR graphs

- AND-OR graph (or tree) is useful for representing the solution of problems that can be solved by **decomposing them into a set of smaller problems**, all of which must then be solved.
- One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution.
- OR arc represent the successor nodes , one with minimum cost is solved



AND-OR graph examples





Problem Reduction

- FUTILITY is chosen to correspond to a threshold such than any solution with a cost above it is too expensive to be practical, even if it could ever be found.

Algorithm : Problem Reduction

1. Initialize the graph to the starting node.

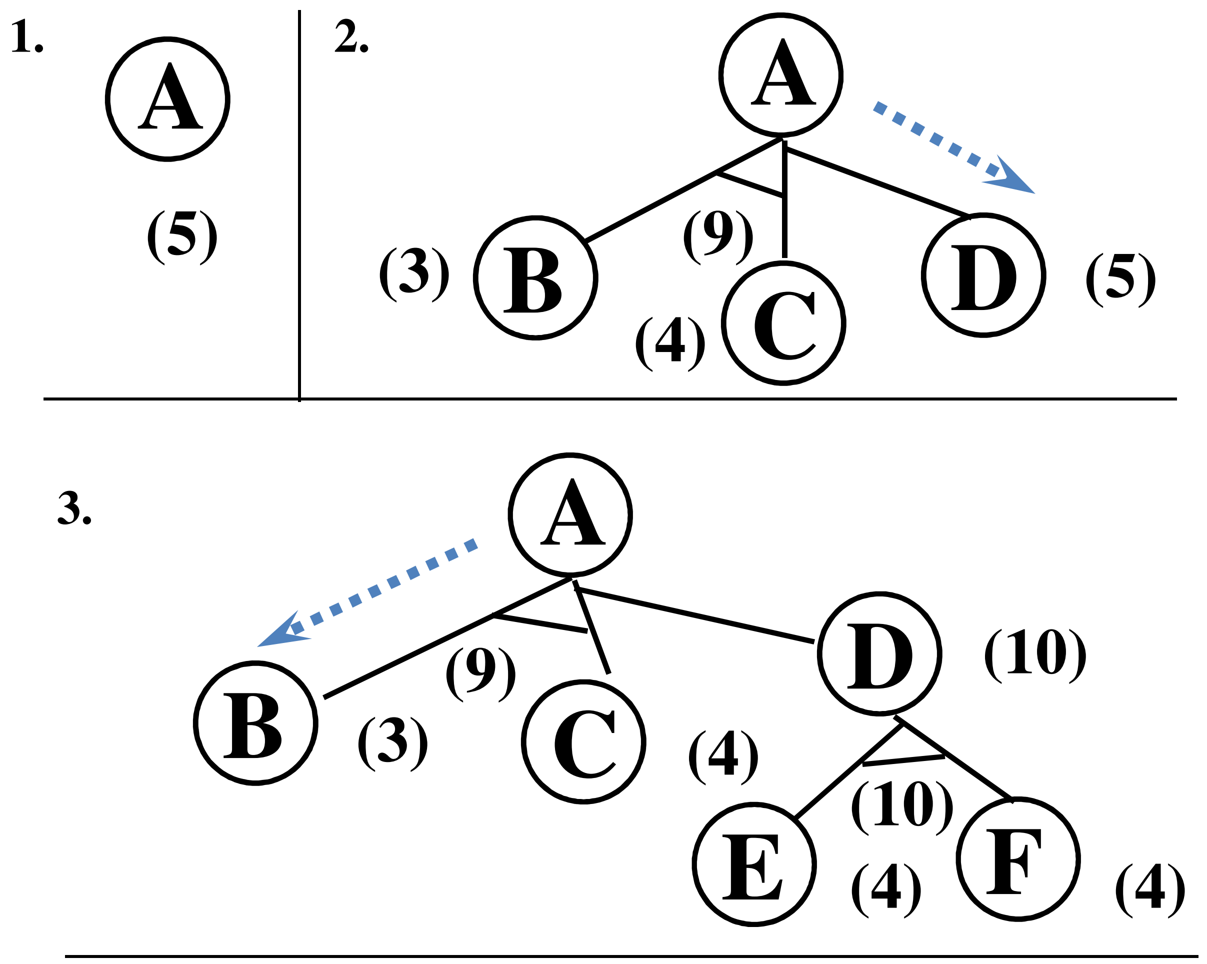
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY
   a. Traverse the graph,
      - starting at the initial node
      - following the current best path,
      - accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
   b. Pick one of these nodes
      - expand it.
      - If there are no successors, assign FUTILITY as the value of this node.
      - Otherwise, add its successors to the graph
      - for each of them compute f’.
      - if f’ of any node is 0, mark that node as SOLVED.

Problem reduction

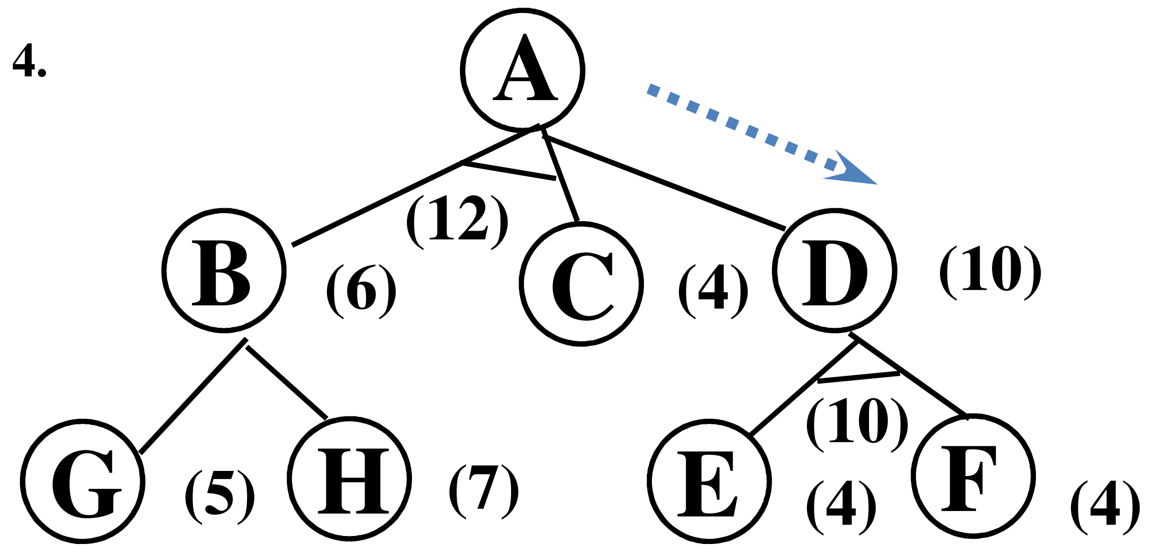
c. Change the f’ estimate
- of the newly expanded node to reflect the new information provided by its successors.
- Propagate this change backward through the graph.

Note: *This propagation of revised cost estimates back up the tree was not necessary in the BFS algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected.*

AND/OR Best-First-Search example



AND/OR Best-First-Search example



Main Features

- How this algorithm is different?
  - It is implemented whenever problem is reduced to smaller parts for solving
  - Hence uses AND OR graph
  - Cost calculated at a node or sub problem is propagated backwards to parent node where new cost is reconsidered for following the best path
  - If current path does not appear to be better then the path with minimum cost is considered

Cost calculation

- Calculation of the cost
  - Node n having **AND arc** with succ m1 and m2
    - h’(m1)+C(n,m1)+h’(m2)+C(n,m2)
    - If each of succ m1 and m2 is ***solved*** then n is marked as ***solved***
  - Node n having **OR arc** with succ m1 and m2
    - Min(h’(m1)+C(n,m1) , h’(m2)+C(n,m2))
    - If minimum succ is ***solved*** then n is also marked ***solved***

Main Features

- If cost of n changes after evaluation of m1 and m2 then this cost must be propagated backwards to all the ancestors

- What is the terminating condition of the algorithm ?
  - When root node is marked **solved** then algorithm will terminate

AO* algorithm

- GRAPH consist of node representing initial state – call this state as INIT
  - *Let G={s}*
- Until INIT is labelled SOLVED or h’ > FUTILITY repeat the following
  - **Select** for expansion one of unexpanded node from marked sub tree and call it NODE n
  - **Expand** For every successor m of n
    - Add succ m to graph
    - If succ m is SOLVED , mark it as SOLVED
    - If not SOLVED compute h’(m)

AO* algorithm contd..

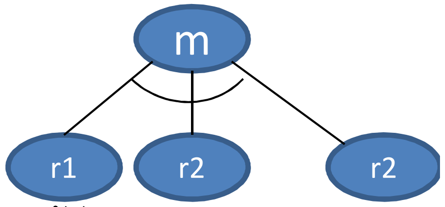
- Back propagation of the newly discovered information
  - Call cost revision(n)

Cost Revision(n) :Let S be set of nodes that have been SOLVED or whose value has been changed. Repeat following steps till S is empty

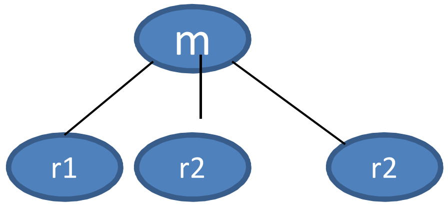
- Initialize S ={n}
- Select from S a node m (select a node possibly whose descendent occurs in G).Remove m from S
- If s={ } return

AO* algorithm contd..

- Compute the cost of each arc emerging from m
  - If m is AND node with succ r1, r2, r3 ……….. rk
    - Set h'(m)= [h(r1)+C(m,r1)] + [h(r2)+C(m,r)]+……..[h(rk)+C(m,rk)] = ∑ (hri + C(m,ri))
    - Mark each succ of m
    - If every succ is SOLVED , mark m as SOLVED



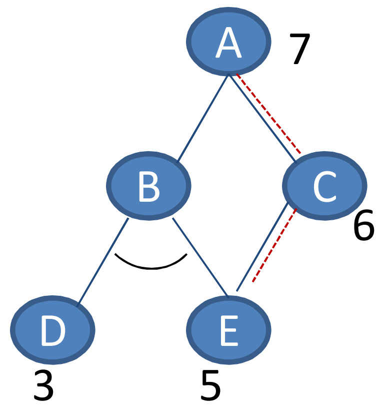
  - If m is OR node with succ r1, r2 , r3 …………. rk
    - Set h(m) = min{h(r1)+c(m,r1), h(r2)+c(m,r2)… h(rk)+c(m,rk)}= min{ h(ri)+c(m,ri)}
    - Mark the best succ
    - If best succ is SOLVED mark m as SOLVED



- If cost of m has changed then change must be propagated , hence insert all ancestor of m in S for which m is marked successor

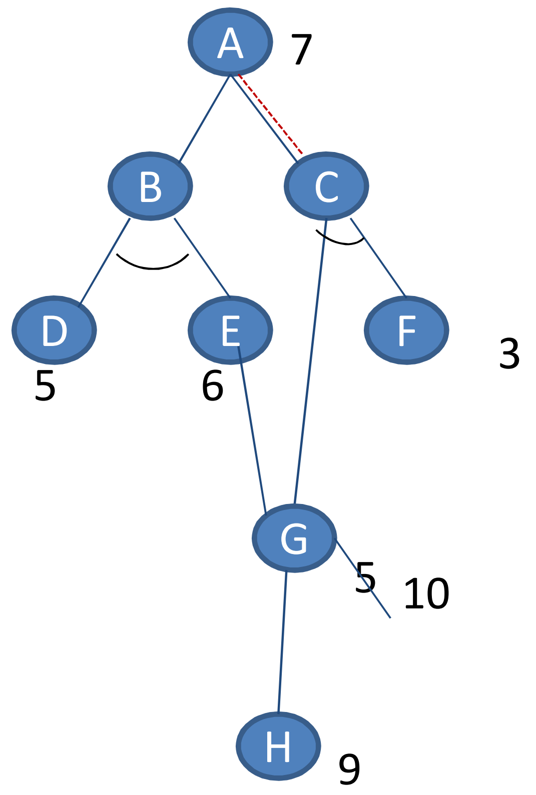
Back propagation to all parent nodes is necessary- why?

- For two main reasons
  - We may follow unfruitful path in state space there by decreasing the efficiency of algorithms – example 1
  - We may not find an optimal solution to the problem



Example 1 If node E value changes from 5 to 10 and cost is p[ropogated back to its parent C only,then path via C does not appear promosing.So, path via B is explored. While exploring and revising the cost of B , new cost of E is included making path via B again expensive. This navigation can be avoided if revised value is passed to both the parent pf E--- B and C

Back propagation to all parent nodes is necessary- why?

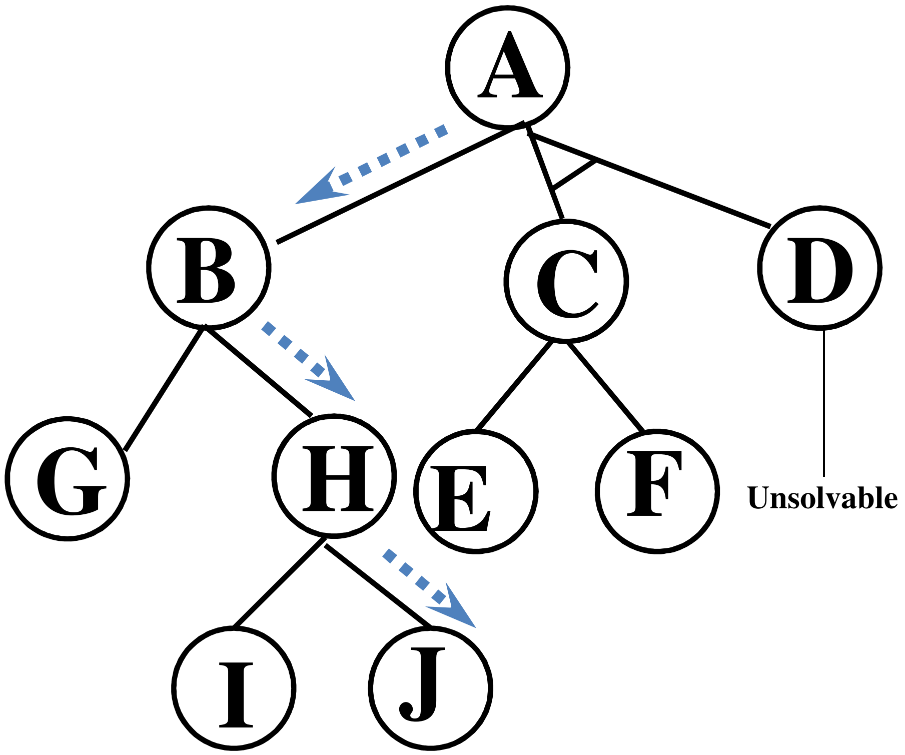


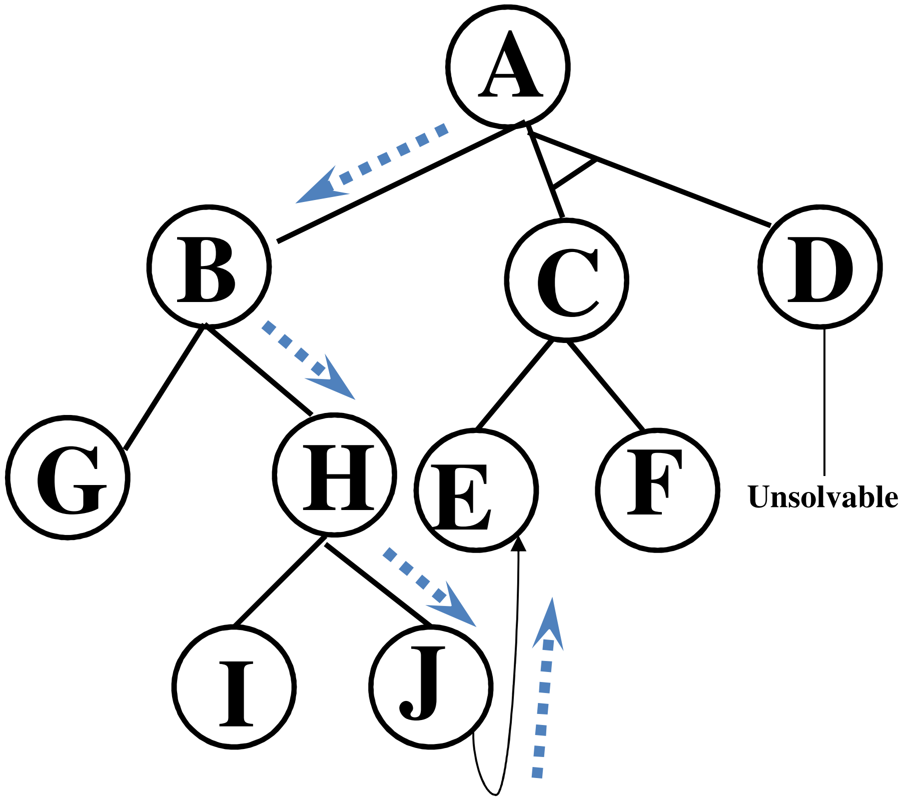
Example 2:If new cost of G is propagated to C only, then path via B appears better. When D is further explored the revised value of B will never include new value of G as it was not given to E and it will continue to explore path via B and may end in a non optimum solution

Some observations

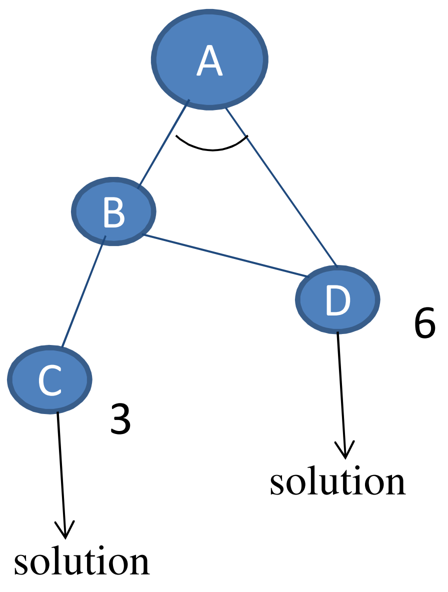
- Back propagation of cost ensures the optimum path solution
- Cost must be propagated to all path backwards . Hence all ancestor must be added to S.
- Algorithm may not find an optimum solution , if graph contain cycles
- At time longer path may be better than shorter path
- Disadvantage : No interaction amongst sub goals
  - If subgoals are independent then algorithm finds an optimum solution but fails when subgoals interact with each other

A Longer path may be better





Interaction between subgoals



AO* fails to take into account interaction between subgoal

- Both C and D lead to solution
- To solve A (AND node) both B and D should be solved
- AO* considers solution of B and solution of D as complete separate process
- Node B expands to C and D both of which leads to solution
- Node C is solved to solve B, as B-C is better path than B-D.
- It is necessary to solve D in order to solve A.
- It is observed that node D will also solve node B , hence there would be no need to solve C
- Cost of solving A through A-B-D is 9 where as cost via C is 12
- As AO * does not consider interaction between B and D optimum path is not taken

Difference between A* and AO*

parameters	A*	AO*
State space data structure	OR graph containing only OR arc in the graph	AND-OR graph containing AND and OR arc in the graph
Types of problem	Non decomposable types of problem	Problems which are solved by reducing it into small subparts
Cost	Cost is best cost of successor nodes	Summing up of cost of succ node is required as arc can be AND arc
Backpropagation	Not required	Back propagation of cost is done to choose the best path
Termination condition	When goal state is found	When the starting node is solved
Path	Pointer to parent node is maintained to trace the path from initial to goal state	Best marked path is maintained
Example	8-puzzle, travelling salesperson	Matrix multiplication, placing blocks one over another, production system

Branch and Bound

- This class of algorithm maintains a upper bound in form of cost of achieving the goal.
- There may be several path to goal
- This algorithm finds the optimum path goal by not allowing a path to explore further if it is greater than the upper bound
- Whenever a first path to goal is achieved , that cost is set as upper bound
- All the paths with cost less than set upper bound is explored , rest are ignored
- If a better path to goal is achieved , upper bound is set again
- And the process ends with optimal path solution

Branch and Bound

- Two list are maintained OPEN and CLOSE. Starting node is s
- Cost of node n is given C(n)
- Upper bound on cost is C*
- Starting node is S, any node is n , goal node is G
- **Algorithm:**
  1. **[Initialize]** Set OPEN={S}; CLOSE={ }; C(S)=0 ; C*=inifinity(large value)
  2. **[Terminate]** If open = { } than return C*
  3. **[Select]** Select a state n from OPEN and save it in CLOSE
  4. **[Set C*]** If n∈ G and C(n) < C* then C*=C(n) go to step 2
  5. **[Expand]** If C(n) < C* generate successor m of n

Branch and Bound

```
if m ~€ [open U close ]
        set C(m) = C(n)+C(n,m)
        Insert m in open
if m   € [open U close ]
        set C(m) =min {C(n), C(n)+C(n,m)}
        Insert m in open
  If cost has decreased and m is in OPEN , update the cost
  If cost has decreased and m is in CLOSE, bring m in OPEN
```

6. [Loop] Go to step 2

Branch and Bound

Advantages :

- Always maintains a upper bound of the cost
- Gives an optimum path solution
- Applicable in situations where multiple goals are present in state space and objective is to find the optimum goal
- Reduces the length of state space every time upper bound is changed