Best - first Search Algorithm (Greedy Search)

① It always selects the path which appears best at that moment

② It is Combination of DFS & BFS

③ It uses the heuristic function $h(n) <= h^*(n)$ and search

$h(n)$ = heuristic cost

$h^*(n)$ = estimated cost

④ The greedy best first algorithm is implemented by Priority queue

Step 1 ① Place the starting node into the open list

② If the open list is empty, stop & return failure

③ Remove the node n, from the open list which has lowest value of $h(n)$ & places it in the closed list

④ Expand node n, and generate the successors of node n

⑤ Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success & terminate the search,

⑥ for each successor node, algorithm checks for evaluation function $f(n)$, & then check if the node has been in either OPEN & CLOSED list. If the node has not been in both list, then add it to OPEN list

(7) Return to step 2

Advantages:

(1) Best first search can switch between BFS & DFS by gaining the advantages of both the ~~[illegible]~~ algorithm

(2) This algorithm is more efficient than BFS & DFS Algorithm.

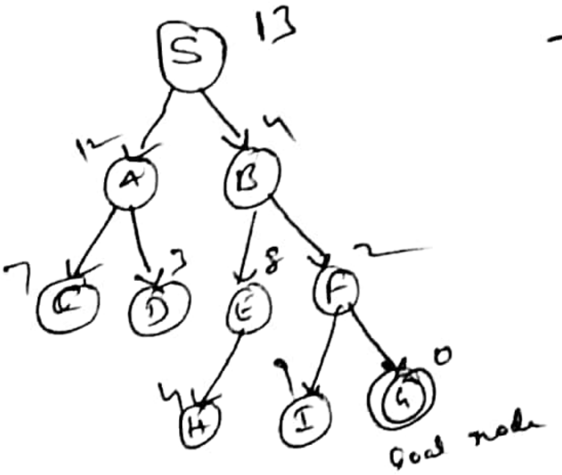
Disadvantages:

(1) It can behave as an unguided depth-first search in the worst case scenario

(2) It can get stuck in a loop as DFS

(3) This algorithm is not optimal.

Ex



node	h(n)
S	13
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
G	0

OPEN LIST, CLOSED LIST

initializing
OPEN (A,B), CLOSE[S]

iteration (1) OPEN[A), CLOSE[S,B]

(2) OPEN[A,E,F], CLOSE(S,B)
OPEN(A,E] CLOSE[S,B,F]

(3) OPEN[E,A,I,G], ~~CLOSE~~
CLOSE(S,B,F)

OPEN[E,A,I] CLOSE[S,B,F,G

$\boxed{S \rightarrow B \rightarrow F \rightarrow G.}$

A^* Search Algorithm

① A^* Search Algorithm finds the shortest path through the search space using the heuristic function

② It uses $h(n)$ is cost to reach the node n from the start state $g(n)$.

③ This Algorithm expands less search tree and provides optimal result

④ It is similar to UCS except that it uses

↓

Uniform Cost Search

$g(n) + h(n)$ instead of $g(n)$.

⑤ A^* use search heuristic as well as the cost to reach the node. Hence we combine both costs as,

$$f(n) = g(n) + h(n) \quad \{\text{fitness number}\}$$

$f(n)$ = estimated cost of the cheapest solution

$g(n)$ = cost to reach node n from start state

$h(n)$ = cost to reach from node n to goal state.

Algorithm:-

(1) place the starting node in OPEN List.

(2) check if the OPEN list is Empty or not, if the list is Empty then return failure & Stops.

(3) Select the node from the OPEN List which has the small value of Evaluation function (g+h), if node n is goal node then return success & stop, otherwise.

(4) Expand node n & generate all of its successors, & put n in the closed list.

- for Each successor 'n'. check whether 'n' is already in the OPEN or CLOSED list.

- If not then Compute Evaluation function for 'n' and place into OPEN list.

(5) Else if node 'n' is already in OPEN & CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

(6) Return to step 2

Advantages:-

(1) It is better algorithm than other search algorithms.

(2) It is optimal & complete.

(3) It can solve very complex problems.

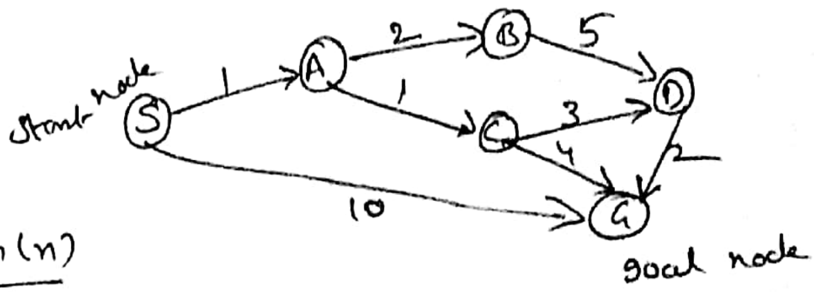
Scanned with CamScanner

Disadvantages:-

① It does not always produce shortest path.

② It is not ~~[illegible]~~ particular for varying large-scale problems.

Ex:-



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

① $S \to A \Rightarrow f(n) = g(n) + h(n)$
$= 1 + 3$
$= 4$ ✓

$S \to G \Rightarrow f(n) = 10 + 0 = 10$ (hold)

② $S \to A \to B \Rightarrow f(n) = 3 + 4 = 7$ (hold)

$S \to A \to C \Rightarrow f(n) = 2 + 2 = 4$ ✓

③ $S \to A \to C \to D = f(n) = 5 + 6 = 11$ (hold)

$S \to A \to C \to G = f(n) = 6 + 0 = 6$ ✓

$\boxed{S \to A \to C \to G}$ → optimal path

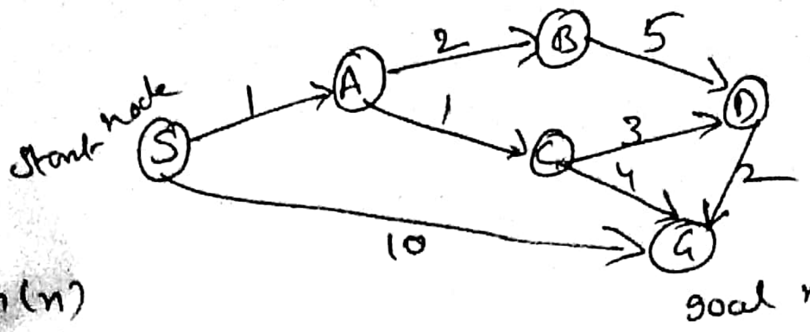
Cost = 6

Disadvantages:-

① It does not always produce shortest path.

② It is not ~~[illegible]~~ particular for varying large-scale problems.

Ex:-



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

OPEN [S] CLOSED[A, G]

OPEN[S, A] CLOSED[G]

① $S \to A \Rightarrow f(n) = g(n) + h(n)$
$= 1 + 3$
$= 4$ ✓

$S \to G \Rightarrow f(n) = 10 + 0 = 10$ (hold)

② $S \to A \to B \Rightarrow f(n) = 3 + 4 = 7$ (hold)

$S \to A \to C \Rightarrow f(n) = 2 + 2 = 4$ ✓

OPEN[S, A] CLOSED[G, B, C]

OPEN[S, A, C] CLOSED[G, B]

③ $S \to A \to C \to D = f(n) = 5 + 6 = 11$ (hold)

OPEN[S, A, C] CLOSED[G, B, D]

$S \to A \to C \to G = f(n) = 6 + 0 = 6$ ✓

OPEN[S, A, C, G] CLOSED[B, D]

$\boxed{S \to A \to C \to G}$ → optimal path

Cost = 6

Scanned with CamScanner

Best-First Search: Agendas

- An *agenda* is a list of tasks a system could perform.
- Associated with each task there are usually two things: a list of reasons why the task is being proposed (often called justifications) and a rating representing the overall weight of evidence suggesting that the task would be useful.

An agenda-driven system operates by cycling through the following sequence of operations:

1. **Choose the most promising task from t agenda. It can be thought of as an expli statement of what to do next or simply as indication of the next node to be expanded.**

2. **Execute the task by devoting to it the number of resources determined by its importance. The important resources to consider are time and space.**

- Executing the task will probably generate additional tasks (successor nodes). For each of them, do the following:

a) **See if it is already on the agenda. If so, then see if this same reason for doing it is already on its list of justifications. If so, ignore this current evidence. If this justification was not already present, add it to the list. If the task was not on the agenda, insert it.**

b) Compute the new task's rating, combining the evidence from all its justifications. Not all justifications need have equal weight.

- **There are some problem domains for which an agenda mechanism is inappropriate.**
- **The agenda mechanism assumes that if there is good reason to do something now, then there will also be the same good reason to do something later unless something better comes along in the interim.**
- **But this is not always the case, particularly for systems that are interacting with people.**

- **Person**: *I don't want to read any more about China. Give me something else.*
- **Computer**: *Ok. What else are you interested in?*
- **Person**: *How about Italy? I think I'd find Italy fascinating.*

Computer: *What things about Italy are you interested in reading about?*

- **Person**: *I think I'd like to start with its history.*
- **Computer**: *Why don't you want to read any more about China?*

- **It would have been fine to have tried to find out why the person was no longer interested in China right after he mentioned it.**

- **The computer chose instead to try to find a new area of positive interest, also a very reasonable thing to do.**

- **But in conversations, the fact that something is reasonable now does not mean that it will continue to be so after the conversation has proceeded for a while.**

Constraint Satisfaction

Constraint Satisfaction in which the goal is to discover some problem state that satisfies a given set of constraints.

Ex:- Cryptarithmetic Problem.

Algorithm:

(1) Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution.

(a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.

(b) If this set is different from the set that was assigned the last time OB was examined (or) if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB.

(c) Remove OB from OPEN.

(2) If the union of the constraints discovered above defines a solution, the quit and report the solution.

(3) If the union of the constraints discovered above defines a contradiction, then return failure.

(4) If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible

Solutions have been eliminated:

(a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.

(b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

Ex:- A Cryptarithmetic Problem

Problem :-

$$\begin{array}{r} SEND \\ +\ MORE \\ \hline MONEY \\ \hline \end{array}$$

initial state :-

No two letters have the same value.

The sums of the digits must be as shown in the Problem.

Rules ~~for~~ for Cryptarithmetic Problem

1. Digit ranges from 0 to 9 only.
2. Each variable should have unique + distinct value
3. Each letter symbol represents only one digit throughout the problem.
4. You have to find the value of letter in the CSP.
5. There must be only one solution to the Problem.

⑥ The numerical base unless specifically stated is 10.

⑦ Numbers must <u>not begin</u> with <u>zero</u> i.e. 0123 (X)

123 (✓)

$$\begin{array}{ccccc} C_4 & C_3 & C_2 & C_1 & \\ & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

$$\begin{array}{ccccc} 1 & & 1 & 1 & \\ & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

Letter	code
S	9
M	1
E	5
O	0
N	6
R	8
D	7
Y	2

$C_4 < 1 ✓, 0$

$\boxed{C_4 = 1} \Rightarrow M = 1$

$C_3 + S + M = O \quad C_3 < 0 ✓, 1$

$0 + S + 1 = O + 10$ → base value

$S = 9 + O$

$S = 9$

$O = 0$

$C_2 + E + O = N \quad C_2 < 0 ✗, 1$

$0 + E + 0 = N$

$E = N$

$C_2 + E + O = N$

$1 + E + 0 = N$

$\boxed{1 + E = N}$

$C_1 + N + R = E + 10$ ← base

$C_1 + 1 + \cancel{E} + R = \cancel{E} + 10$

$C_1 + 1 + R = 10$

$C_1 + R = 9$

$C_1 < 0 ✗, 1 ✓$

$0 + R = 9$

$1 + R = 9 \Rightarrow R = 9 - 1$

$\boxed{R = 8}$

Y Y E E N
2 3 4 5 6 7

$$D + E = 10 + Y$$ base value

if we assuming $E = 6$ $N = 7$ $(1 + E = N)$

$$1 + 6 = 7$$

$$D + 6 = 10 + Y$$

$$D = 4 + Y$$

if we assuming $\boxed{Y = 2}$ | $Y = 3$

$D = 4 + 2 = 6$ $D = 4 + 3 = 7$

if we assuming $\boxed{E = 5}$ then $\boxed{N = 6}$

~~$D + E = 10 + Y$~~

~~$6 + 5 = 10 + Y$~~

~~$11 = 10 + Y$~~

~~$Y = 1$~~

$$D + E = 10 + Y$$

$$D + 5 = 10 + Y$$

$$D = 5 + Y$$

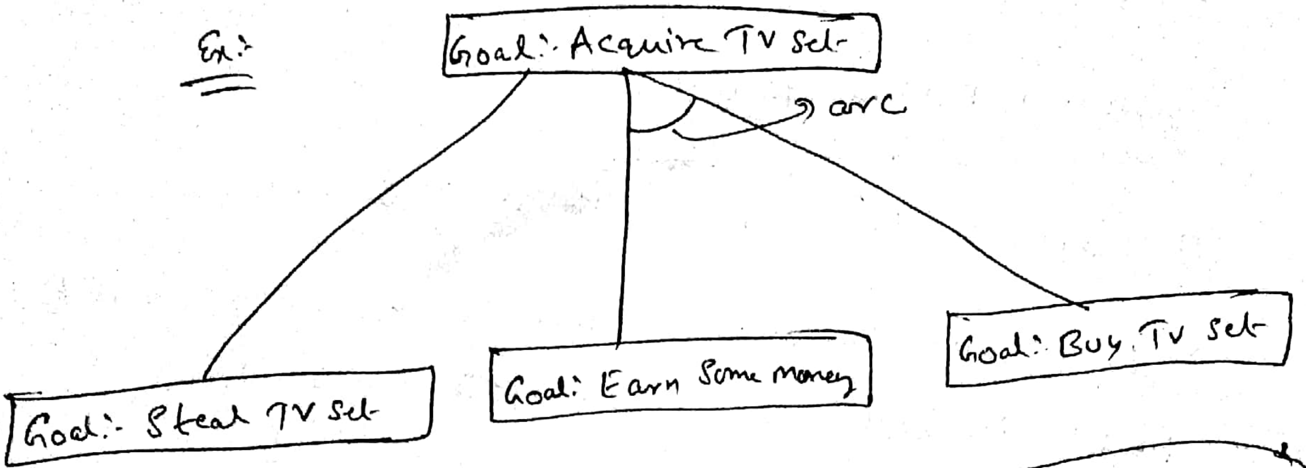
$$D = 5 + 2 \quad (Y = 2)$$

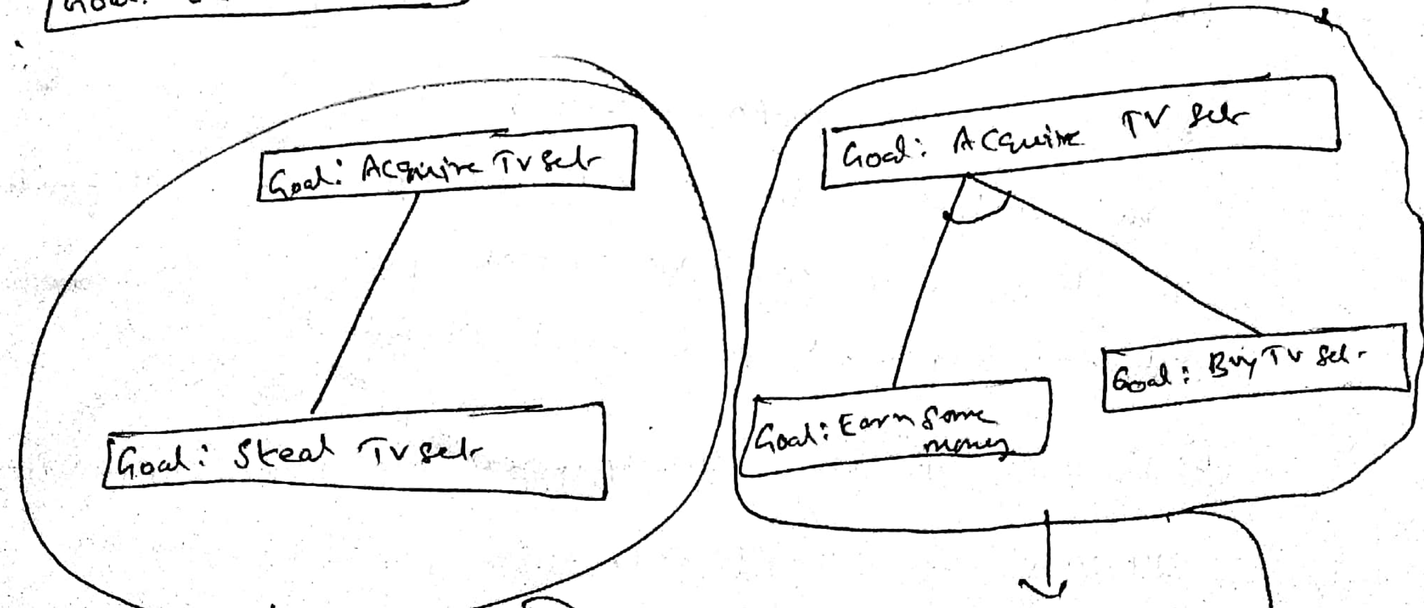
$$\boxed{D = 7}$$

Problem Reduction

The Process of decomposing a Complete Problem into a set of Sub Problems and then integrating all these Sub-solutions to get the solution of the given Complex Problem is known as Problem Reduction.

Ex:





OR representation

AND representation

It provides one solution of the given problem

It provides another solution of the given problem.

(*) It provides the AND-OR Graph representation

AND-OR graph

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 be solved.

⟹ OR node represents a choice between possible decomposition

⟹ AND node represents a give decomposition.

Algorithm: Problem Reduction

(1) Initialize the graph to the starting node.

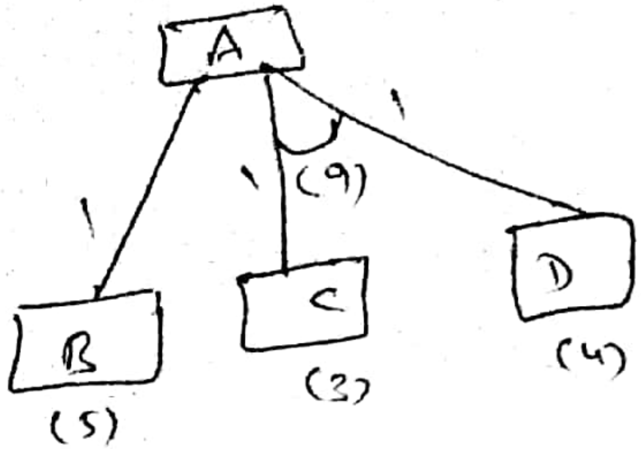
(2) Loop until the starting node is labeled SOLVED or ~~[illegible]~~ FUTILITY.
Until its cost goes above

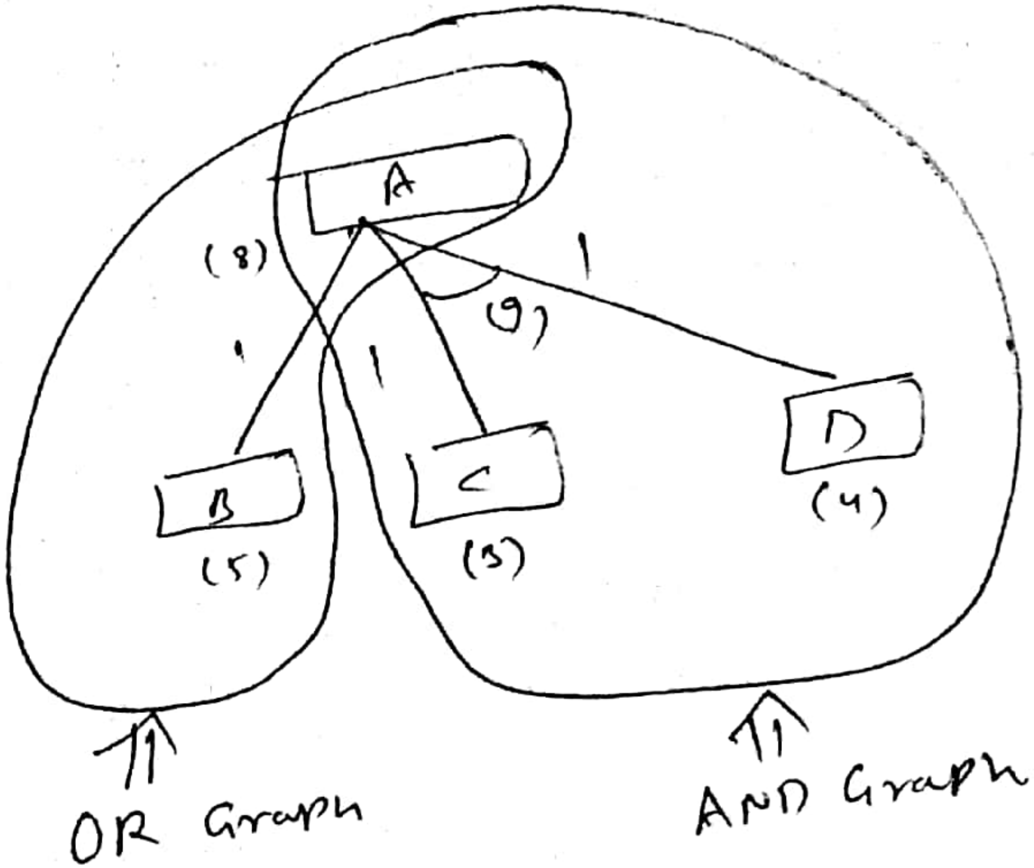
(a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.

(b) Pick one of these unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph and for each of them compute f'.

(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. If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED.

Ex:-





AO* Algorithm:-

(1) AO* represents AND/OR Graph that is used to find more than one solution

(2) AO* is informed search Technique and worked as Best-First Search (BFS)

(3) AO* Algorithm is an efficient method to Explore a solution path.

(4) AO* is used to solve cyclic AND-OR Graph.

The Problem is divided into a set of sub-Problems where each sub-Problem can be solved seperately.

(5) Node in the Graph will point both Down to its Successors and UP to its Parent nodes.

(6) Each node in the Graph will also have a FUTILITY VALUE (F)

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

g = Evaluation function

h = cost function

Algorithm:-

(1) Let GRAPH consist only of the node representing the initial state. (call this node INIT). Compute h'(INIT)

(2) Until INIT is labeled SOLVED (or) until INIT's h' value becomes greater than FUTILITY, repeat the following procedure.

(a) Trace the labeled arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node NODE.

(b) Generate the successors of NODE. If there are none, then assign FUTILITY as the h' value of NODE. If there are successors, then for each one that is not also an ancestor of NODE do the following.

(i) Add successor to Graph

(ii) If successor is a terminal node, label it SOLVED and assign it an h' value of 0.

(iii) If successor is not a terminal node, compute its h' value.

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

(i) If possible, select from S a node none of whose descendants in GRAPH occurs in S.

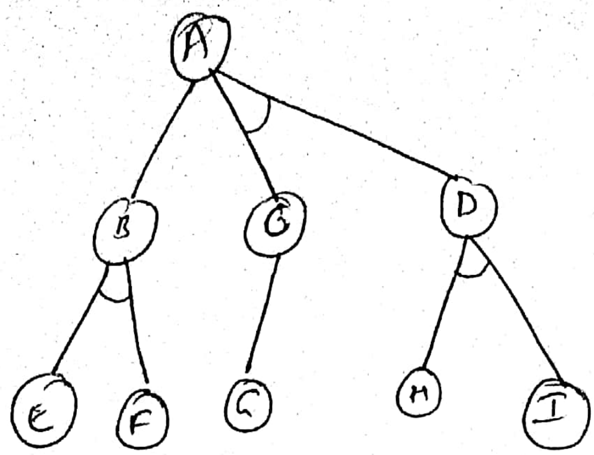
(ii) Compute the cost of each of the arcs emerging from CURRENT.

(iii) Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.

(iv) Mark CURRENT SOLVED if all of the nodes connected to it through the new labeled arc have been labeled SOLVED.

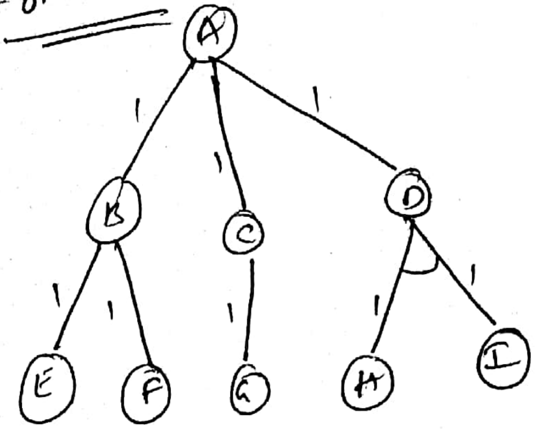
(v) If CURRENT has been ~~solved~~ labeled SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph.

Ex:-



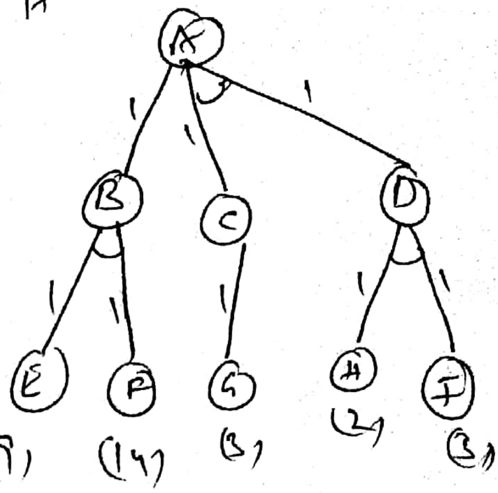
Cost of each node

1

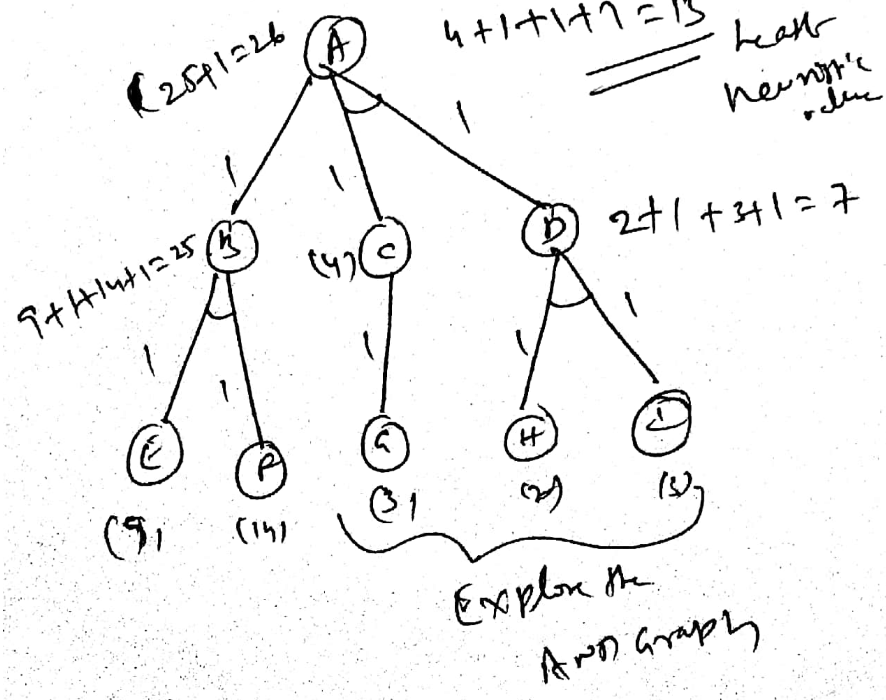


Heuristic values

2



3 $f(n) = g(n) + h(n)$



Means - Ends Analysis:-

We have presented a collection of search strategies that can reason either forward or backward, but for a given problem, one direction (or) the other must be chosen often, however, a mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the major parts of a problem first and then go back and solve the small problems that arise in "gluing" the big pieces together. A technique known as means-ends analysis.

① The means-ends analysis process centers around the detection of differences between the current state and the goal state.

② An operator that can reduce the difference must be found.

③ The kind of backward chaining in which operators are selected and then subgoals are set up to establish the preconditions of the operators is called operator subgoaling.

④ The means-ends analysis relies on a set of rules that can transform one problem state into another.

⑤ A seperate data structure called a difference table indexes the rules by the differences that they can be used to reduce.

Algorithm: Means-Ends Analysis (*CURRENT*, *GOAL*)

1. Compare *CURRENT* to *GOAL*. If there are no differences between them then return.

2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:

   (a) Select an as yet untried operator *O* that is applicable to the current difference. If there are no such operators, then signal failure.

   (b) Attempt to apply *O* to *CURRENT*. Generate descriptions of two states: *O-START*, a state in which *O*'s preconditions are satisfied and *O-RESULT*, the state that would result if *O* were applied in *O-START*.

   (c) If
   (*FIRST-PART* ← MEA(*CURRENT*, *O-START*))
   and
   (*LAST-PART* ← MEA(*O-RESULT*, *GOAL*))
   are successful, then signal success and return the result of concatenating *FIRST-PART*, *O*, and *LAST-PART*.

Operator	*Preconditions*	*Results*
PUSH(obj, loc)	at(robot, obj) ∧ large(obj) ∧ clear(obj) ∧ armempty	at(obj, loc) ∧ at(robot, loc)
CARRY(obj, loc)	at(robot, obj) ∧ small(obj)	at(obj, loc) ∧ at(robot, loc)
WALK(loc)	none	at(robot, loc)
PICKUP(obj)	at(robot, obj)	holding(obj)
PUTDOWN(obj)	holding(obj)	¬ holding(obj)
PLACE(obj1, obj2)	at(robot, obj2) ∧ holding(obj1)	on(obj1, obj2)

Figure 3.15: The Robot's Operators

	Push	Carry	Walk	Pickup	Putdown	Place
Move object	*	*				
Move robot			*			
Clear object				*		
Get object on object						*
Get arm empty					*	*
Be holding object				*		

Figure 3.16: A Difference Table

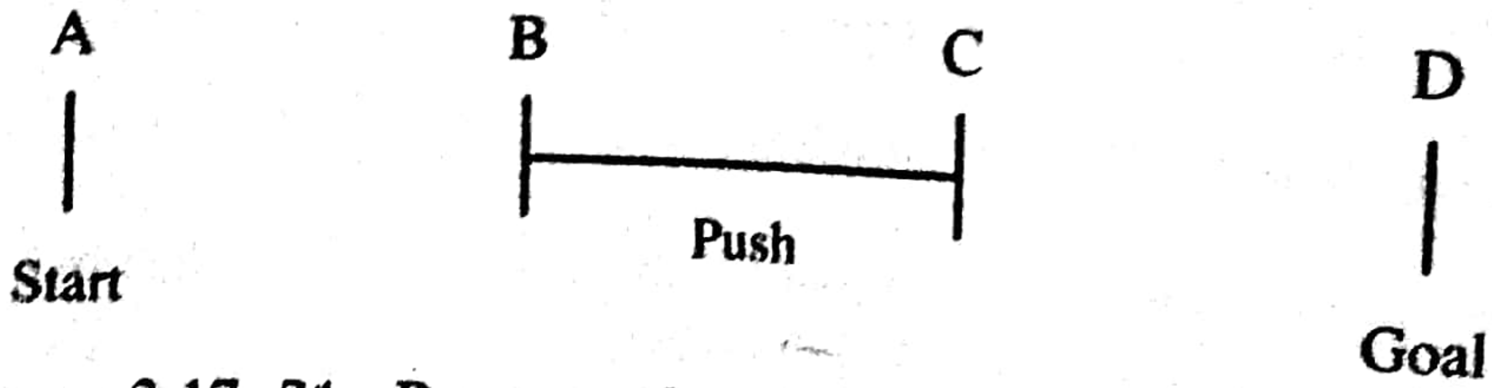


Figure 3.17: The Progress of the Means-Ends Analysis Method

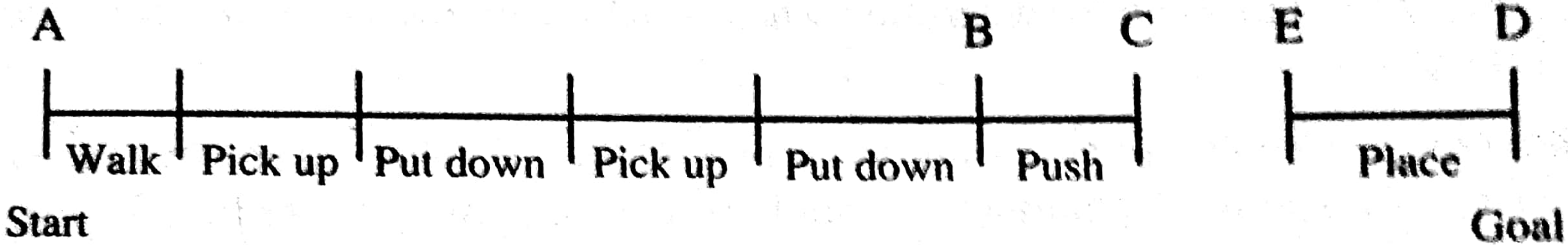


Figure 3.18: More Progress of the Means-Ends Method