Artifical Intelligence

→ AI is study of how to make computer do things better than humans.

→ AI.

Mundane	Formal	Expert
→ Perception	→ Game Playing	→ Enquiring
- Speech	- Chess	- Design
- Vision	- Checkers	- Fault Finding
→ Natural Lang.	- Go	→ Scientific Analysis
- Understanding	→ Mathematics	→ Financial "
- Generation	- Geometry	
- Translation	- Logic	
→ Common Sense Reasoning	- Integral Calculus	
	- Proving properties	

→ Alan Turing - Father of AI.

→ Experiments → " Can Machine Think" (Turing Test)

→ MIT Experiment

- Chatbot named as ELIZA in 1966 developed by Joseph Weizen.

→ Chinese Room Experiment.

→ AI:

(1) Speech Recoginition
(2) Image Processing
(3) Computer Vision
(4) Neural Networks
(5) Knowledge Discovery
(6) Pattern Recoginition
(7) Practical Sensing
(8) Natural language Understanding
(9) NC Generation
(10) Quantitative Reasoning

(11) Knowledge Representation
(12) Machine learning
(13) Semantic
(14) Search
(15) Fuzzy Sets
(16) Logic
(17) Speech Synthesis
(18) Graphics
(19) Robot Control.

→ AI Technique.

Problems can be soft problems or hard problems
- soft problems → Time complexity is polynomial
- hard problems → Solvable but time complexity is not polynomial
- hard problems → Not solvable

- Problems of AI:
  1. It is voluminous
  2. It is hard to categorize accurately.
  3. It is constantly changing
  4. It differs from data by being recogonized in a way that corresponds to a way it will be used.

→ AI Technique is a method that experts knowledge that should be represented in such a way that:
1. Knowledge should capture generalization.
2. It can be understood by the people, who must provide it.
3. It can easily be modified to correct errors and reflect changes in our world view.
4. It can be used in great many situations even if it is not totally accurate or complite.
5. It can be used to help overcome it's own share by helping to narrow range of possibilities that must usually be considered.

→ Tic Tac Toe.

① The complexity
② Use of generalization
③ Clarity of their knowledge
④ Extensibility of approach.

→ Board → Vector of 9 elements
0 – Blank
1 – X
2 – O

- Prog 1

Algorithm :
① View vector board as ternary number (base 3).
② Convert it to decimal number
③ Use number computed in ① as index in move table and access the vector stored there.
④ Vector selected in step ② represented the way board will look after move that should be made so set board equal to that array.

- Prog 2

Algorithm : 2 → – ; 3 → X ; 5 → O.
Turn is integer having value b/w 1 to 9
Turn is odd → X's turn
Turn is even → O's turn.

Functions : Go(n)
Make2()
PossWin(p) → Tells whether p can win or not.

Turn:
(1) Go(1) : Place X at pos 1.
(2) If Board(5) is blank, go(5).
(3) If Board(9) is blank, Go(9) else Go(3).

(4) If possWin(X)! = 0, then Go(PossWin(X))
Else Go(Make2)

(5) If possWin(X)! = 0, then Go(PossWin(X))
Else If possWin(O)! = 0, then Go(PossWin(O))
Else Go(Make2)

6) If possWin(O)! = 0, then Go(PossWin(O))
Else If possWin(X)! = 0 then Go(PossWin(X))
Else Go(Make2)

7) If possWin(X)! = 0, then Go(PossWin(X))
Else If possWin(O)! = 0 then Go(PossWin(O))
Else Go wherever blank.

8) If possWin(O)! = 0, then Go(possWin(O))
Else If possWin(X)! = 0 then Go(possWin(X))
Else Go wherever blank.

9) If possWin(X)! = 0, then Go(PossWin(X))
Else If possWin(O)! = 0, then Go(possWin(O))
Else Go wherever blank.

- Optimized Program 2.

We use magic square number.

8 (1)	3 (2)	4 (3)
1 (4)	5 (5)	9 (6)
6 (7)	7 (8)	2 (9)

↳ Sum of each row, col, diagonal = 15.

$x_1\ x_2\ x_3$

$15 - (x_1 + x_2) \rightarrow (1-9)$

→ Program 3

- Data Structure – BoardPosition
  - → A structure containing 9 element vector representing board.
  - → A list of board pos that could result from next move.
  - → A number representing an estimate of how likely board position is to lead to an ultimate win for player to move.

- This algorithm is solved using AI Technique.

→ Three important AI Techniques:
① Search ② Use of Knowledge ③ Abstraction.

(I) Search
→ Provides a way of solving problems for which no more direct approach is available as well as framework into which any direct technique that are available can be embedded.

(II) Use of Knowledge
→ Provides a way of solving complex program by exploiting structures of object that are involved.

(III) Abstraction.
→ It provides a way of separating imp features and variations from the many unimportant ones.

→ How to build a system

① Define a problem precisely.
→ The defⁿ must include precise specifications of what initial situations will be as well as what final situations constitute acceptable sol. to the problem.

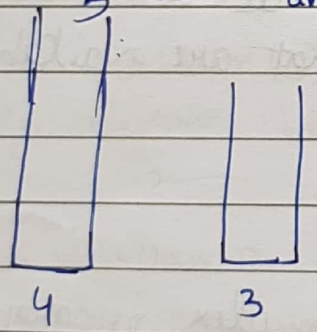
② Analyse the prob.
→ A very few important features can have an immence impact on the appropriateness of various techniques for solving the problems.

③ Isolate and represent task knowledge that is necessary to solve the prob.

④ Choose best problem solving techniques and apply to the problem.

→ Water Jet Problem.

2 jugs - 3 gallon water and 4 gallon water.



(x, y)
Initial state : (0, 0)
Go : (2, n).

Legal moves :
① (x, y) → (4, y) If x < 4
② (x, y) If y < 3 → (y, 3)
③ If x > 0 (x, y) → (x-d, y)
④ If y > 0 (x, y) → (x, y-d).
⑤ If x > 0 (x, y) → (0, y)
⑥ If y > 0 (x, y) → (x, 0)
⑦ If x+y ≥ 4 and y > 0 (x, y) → (4, y-(4-x))
⑧ If x+y ≥ 3 and x > 0 (x, y) → (x-(3-y), 3).

(9) If $x+y \leq 4$ and $y \geq 0$.
$(x, y) \rightarrow (x+y, 0)$

(10) If $x+y \leq 3$ and $x > 0$
$(x, y) \rightarrow (0, x+y)$.

(11) $(0,2) \rightarrow (2,0)$.

(12) $(2,y) \rightarrow (0,y)$.

→ Initially $(0,0)$
Applying (2), $\rightarrow (0,3)$
(9), $\rightarrow (3,0)$
(2) $\rightarrow (3,3)$
(7) $\rightarrow (4,2)$
(5) $\rightarrow (0,2)$
(11) $\rightarrow (2,0)$.

→ Production System.
A production system consists of:
① Set of rules each consisting of LHS that determines applicability of the rule and RHS that describes operation to be performed if the rule is applied.

② One or more knowledge base or database that contain whatever info is appropriate for particular task. Some tes. part of database may be permanent while other part of it may pertain only to the solution of current prob.

③ A control strategy that specifies the order in which rules will be compared to DB and way of resolving conflicts that arise when several rules match at once.

(0,0)

The req. of good control strategy is:
① It causes motion.
② It should be systematic.

- Problem Characteristics:

(1) Is Problem Decomposable.
(2) Can Solution Steps be ignored or undone.
- Ignorable → Eg in proving a theorem, a sub-step can be ignored.
- Recoverable → Eg 8 puzzle problem: Solution step taken which is not correct can be undone
- In-recoverable → Eg in chess: We can't go back when a move is played.

(3) Is Universe predictable?
→ Universe is environment in which problem is working.

(4) Is the solution absolute or relative.
→ If there are many solutions to the problem, and every solution is similar, then solution is absolute and relative.
(1) Markus was a man
(2) Markus was pompian
(3) No mortal lives longer than 150 year
(4) It is now 1991 AD.
(5) Markus was born in 1480 AD
(6) All pompian died in 1790 AD
(7) All men are immortal.

Markus is dead now by two ways: (1) He is a man (3), (5), (7)
(2) He is pompian (6), (4)

(5) Is the solution state or path?
→

(6) Role of knowledge.

→ Eg Chess and Newspaper Scanning Problem
→ It is based on less knowledge →

→ State Space: → Realm of all the states that are accessible.

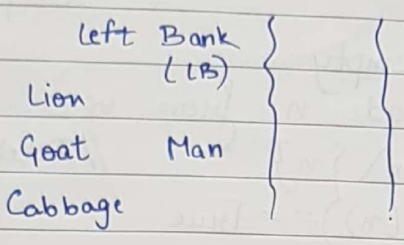
→ (I) Generate and Test Algorithm
→ 1) While more candidates exist
2. do Generate a candidate.
3. Test whether it is a solution.
4. return failure.

Functions: moveGen(n) } → It is problem specific.
goalTest(n) }

→ Generates all states possible to reach from n.

AI

→ River Crossing Problem. (Man, Lion, Goat, Cabbage Problem).
- A man needs to transfer set of objects from left bank to right bank using a boat.



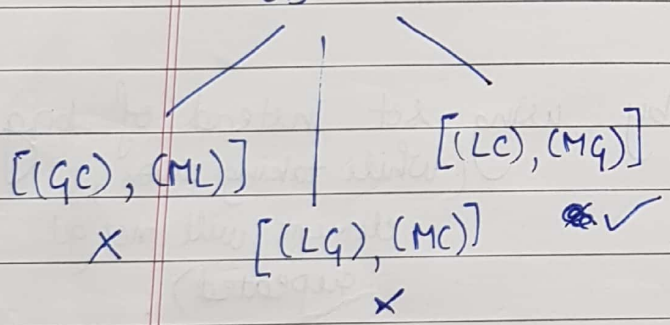
Constraints:

→ If man is not there, lion will eat goat and goat will eat cabbage.

→ Only single object can be transferred by man in a boat.

→ Initial State
[(M, L, G, C); ( )]
LB RB

Goal State.
[( ); (M, L, G, C)]
LB RB.



→ Another approach is maintaining only one list.

MoveGen():

(1) Initialize set of successors C to ϕ.
(2) Add M to complement of given state N to get a new state S
(3) If given state has left, then add right to S
Else add left.
(4) If S is legal, then add S to set of successors C.
(5) For each other entity E in N;
{ Make a copy S' of S.
Add E to S'
}

6) If Legal(s'), then add s' to C.
7) Return C.

→ Simple Search 1 Algorithm:

(Open is a bag)
↓
Multiple elements can be there

(1) Open ← {start}
(2) While open is not empty.
(3) do pick some node n from open.
(4) open ← open \ {n} // Deleting node n from open
(5) if (GoalTest(n) == True
(6) return n.
(7) Else open ← open ∪ {movegen(n)}
(8) return failure;

Drawbacks:

① It may happen that algo may get stuck in infinite loop.
② Algo will have redundant node.

→ Redundancy can be removed by using set instead of bag. (While taking union, element will not get repeated).

→ To avoid infinite loop, we will use another set closed which will consist of elements that are processed.

→ Simple Search 2 Algo:

(1) Open ← {start} closed = { }
(2)
(3)
(4) open ← open / {n}
closed ← closed ∪ {n}
(5)
(6)
(7) Else. open ← open ∪ {movegen(n) / closed(n)}

→ Drawbacks:

① It is not of much use in path finding questions.

→ To avoid the above problem, open is set of sublists. Every sublist has parent node and child node. Closed is simple list.

→ Simple Search 3

```
1) Open → { {start} }                    // {start} → {S, NIL}
2) Closed → ϕ.
3) While open is not empty
4) [ do pick some node n from open
5)       h ← Head(n)                     // child node.
6)       If GoalTest(h) == true.
7)       return (Reverse(n))             // Reverse(n) returns path
8)       Else
9)           closed ← closed ∪ (h)
10)          Sucessors ← { moveGen(n) / closed }
11)          Successors ← { successors / Mapchar(open) }
12)          Open ← { Open \ (n) }
13)          For each S in successor
14)              do Add Cons(S, h) to open.
15) return failure.
```

Cons(S, h) ⇓ For creating pair where h is parent of nodes S.

Mapchar(open) ⇓ Returns head of every pair in open

→ Simple Search 4

```
1) Open. { (Start, NIL) }
2) Closed { ( , ) }
3) While open is not empty.
       do node-pair ← Head(open)
          node ← Head(nodepair)
```

LAB 5

Build path

if (GoalTest(node) == TRUE)
then return ReconstructPath(node pair, closed)
Else closed ← Cons(nodepair, closed)
Children ← Moveopen(node)
noloops ← RemoveSeen(Children, open, closed)
new ← Make pairs(noloops, node)
Open ← Append(new, tail(open));

Return failure;

All nodes except head.

→ This is DFS. because nodepair is acting as stack.
For implementing algo as bfs, we modify last line
Open ← Append(tail(open), new)

$S = \frac{a(r^n - 1)}{r - 1}$

How to compare two Search Algorithm

→ (1) Does algorithm always find a solution whether there exist one?

(2) Time Complexity → How much time does the algorithm run for before finding a solution. We will get measure of time complexity by no. of nodes the algo picks up before it finds the solution.

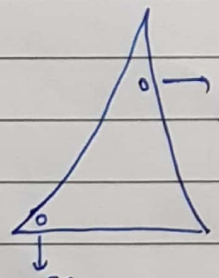
(3) Space Complexity → We will use number of nodes in open list as measure of space complexity.

4) Quality of solution → If some solutions are better than other, but what kind of solution algo finds. Shortest path to goal state.

→ BFS v/s DFS.

1) Both BFS and DFS are complete for finite space.
In case of infinite state space infinite → BFS is complete DFS incomplete (it considers every node).

2)



→ In worst case, both BFS and DFS involve visiting all nodes.

→ Branching factor k means every node has k children nodes.

→ If depth is d and branching factor b.

No. of nodes visited = $(d+1) + \frac{b^{d+1} - 1}{b-1}$

in DFS = $\frac{b^d}{2}$ If b and d are large value.

No. of nodes visited in BFS, N_{BFS} = $\frac{b^d}{2b}(b+1)$

BFS, and DFS are called blind searches as it does not care about direction of goal node.

$$\frac{N_{BFS}}{N_{DFS}} = \frac{b+1}{b}.$$

No. of nodes in open list in DFS $= (b-1)(d-1) + b.$

↳ linear wrt depth at any moment of time.

No. of nodes in open list in BFS $= b^{d+1}$

↳ It is exponential wrt depth.

DFS is less complex.

BFS returns shorter path than DFS.

Heuristic Search

Heuristic search provides sense of direction.

Heuristic Func → The idea of using heuristic func is to guide the search so that it has tendancy to explore region leading to goal. It is designed to help algorithm to pick that candidate from open list that is most likely to be on path to goal. A heuristic val is computed for each of candidate. Heuristic val could be an estimate of distance of goal from node. The heuristic func must not be computationaly expensive because idea is to cut down on computational cost of search algo.

$H(n)$ → Heuristic func on node n.

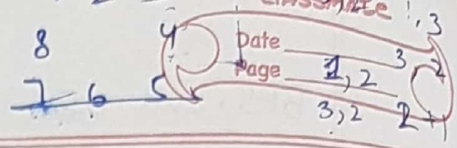
Eg To find Shortest route b/w two points:

(i) Euclidian distance.

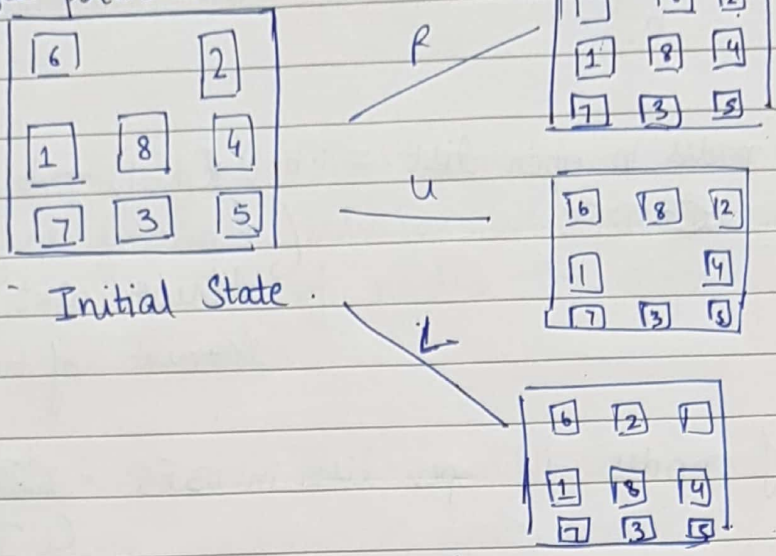
(ii) Manhattan method.

1, 1
2, 2

Goal →



→ 8-puzzle problem.



→ Heuristic func – Count no. of tiles that are out of place

$h_1(R) = 6$ $\quad h_1(U) = 5$

$h_1(L) = 5$

→ Manhattan distance :⟷ h_2 for each element (1 – 8, Space) → $|x_2 - x_1| + |y_2 - y_1|$

$h_2(R) = 1 + 1 + 3 + 0 + 0 + 2 + 0 + 1 + 2 = 10$

$h_2(L) =$

$h_2(U)$

• Best First Search.

→ DFS : Open ← append (New, tail(open))

↓

BeFS : Open ← sort(append(new, tail(open))

(Implement open as priority queue / heap).

→

(1) Finite state space → It is complete.
Infinite state space → Completeness depends on quality of heuristic function.

(2) Quality : Depends on heuristic function.

(3) Space Complexity → Depends on heuristic func.

(4) Time Complexity → Depends on heuristic function.

MidSem → Unit 1 + 2

It is used to judge the quality of heuristic func

→ ① Effective branching factor is equal to total node expanded divided by nodes in the solution. (≥ 1)

→ ② Penetration → Nodes in solution divided by total nodes expanded. ($0 \leq p \leq 1$)

→ Hill Climbing Algorithm – A person is blindfolded and he has to climb the hill. So the person will move in direction where there is some elevation. This is motivation of algo.

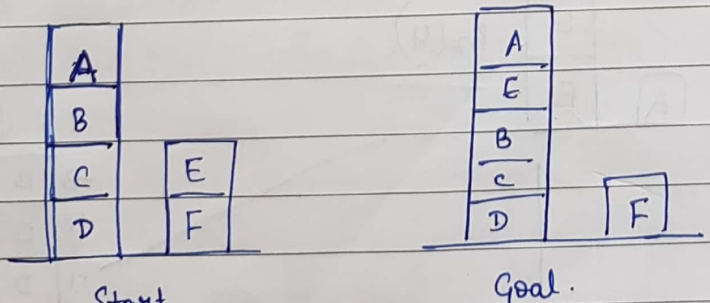
– Steepest gradient ascent

Algo:

(1) node → start
(2) newnode → Head(sort$_n$(MoveGen(node)))
(3) while h(newnode) < h(node)
(4) do node ⇐ newnode.
(5) newnode ← Head(Sort$_n$(MoveGen(node)))
(6) return node

Problem: ① There might be successor which have heuristic value greater than goal node but it is in the path. In this case, algo will give wrong sol^n. This problem is local maxima.

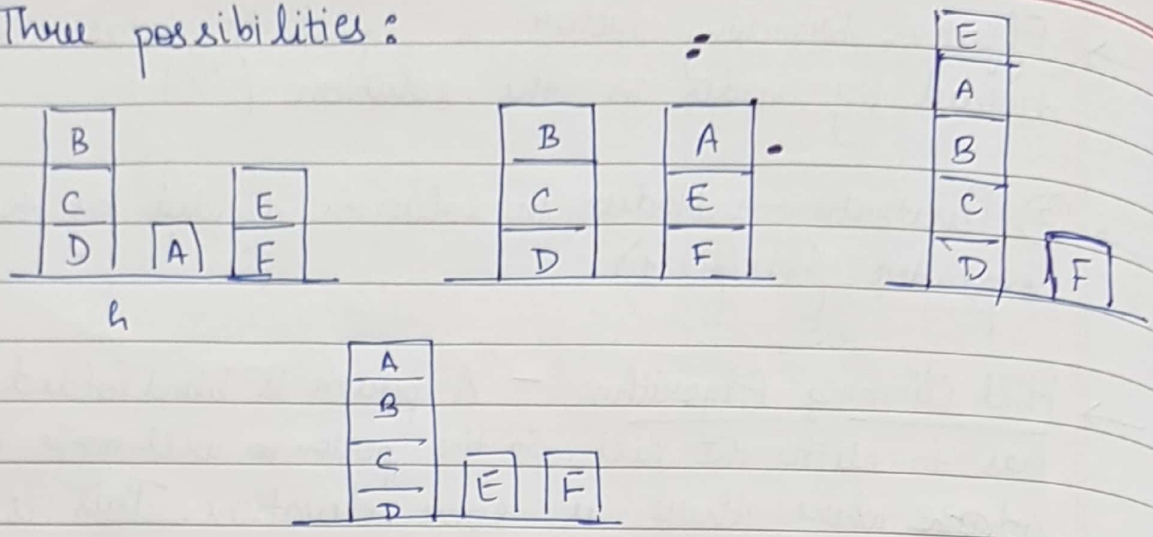
→



Start Goal.

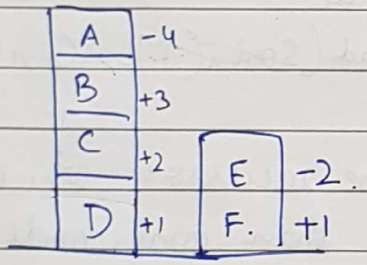
– $h_1(n)$ → Simply checks whether each block is on correct block wrt final configuration. We add 1 for every block that is on block it is supposed to be on and subtract 1 for every block i.e. on wrong one.

→ Three possibilities :



h

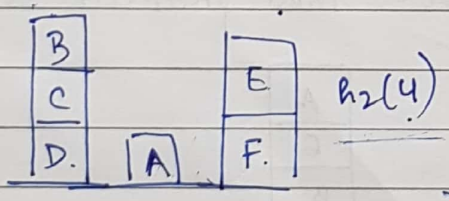
- $h_2(n)$ → Second heuristic function looks at entire pile that block is resting on. If configuration of pile is correct wrt goal, it adds 1 for every block in the pile or else it subtracts 1 for every block in that pile.



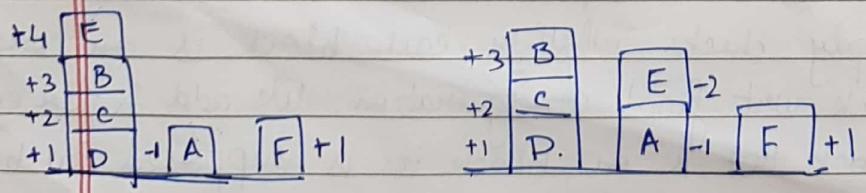
$h_2 = (1)$

When we apply h1, we encounter local maxima problem.

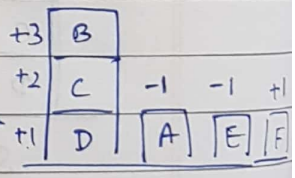
→



$h_2(4)$



$h_2(4)$



$h_2(7)$

h_2 (4) ✓

- A* → Dijkstra
  ← Best First Search

→ A* takes best features Dijkstra and best first search. Best first search does have a sense of direction. It uses a heuristic function to ~~estimate~~ decide which of candidate node is likely to be closest to goal and expands that. Dijkstra algo searches over a given graph, and keep exactly one copy of each node and back pointers for best route.

→ $f(n)$ → Estimated cost of a path from start to goal via node n.

$f^*(n)$ → Actual cost.

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

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

↳ Least cost b/w S and n.

$h^*(n)$ → Heuristic estimate

- A* admissable property

→ ① $g(n) \geq g^*(n)$ ↳ ② $h(n) \leq h^*(n)$

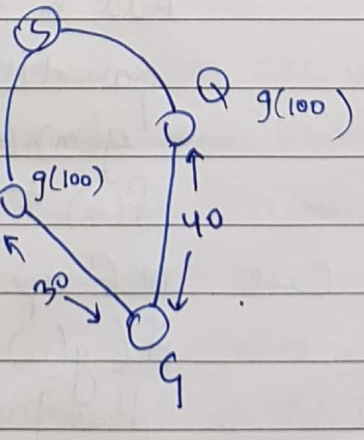
↓

May or may not be least.

Let us take two heuristic func:

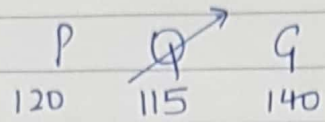
I) $h_1(P) = 50$ $h_1(Q) = 45$

II) $h_2(P) = 20$ $h_2(Q) = 15$



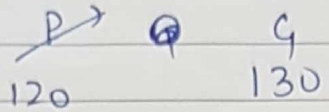
P 150 → Q 145 → G 140. → So this is not shortest path

Using h_2 (under-estimated func).



So, we get best path by choosing under estimated function.

Now P is selected



→ Procedure (A *)

(1) Open ← List (start)
(2) f(start) ← h(start)
(3) parent (start) ← NIL
(4) Closed ← { }
(5) While open is not empty
do
Remove node n from open such that f(n) has lowest value.
Add n to closed
If GoalTest(n) = TRUE
return ReconstructPath(n)
neighbours ← Movegen(n)
for each m ∈ neighbours
do switch
case m ∉ open and m ∉ closed
Add m to open.
parent(m) ← n
g(m) ← g(n) + k(n, m)
f(m) ← g(m) + h(m)
Case m ∈ open
if (g(n) + K(n, m)) < g(m)
then parent(m) ← n
g(m) ← g(n) + K(n, m).

Case . m ∈ closed

If (g(n) + K(n, m)) < g(m)

then parent(m) ← n

g(m) ← g(n) + K(n, m)

f(m) ← g(m) + h(m)

propagate Improvement(m)

Since m is in closed its neighbours generated will be either in open or closed

6. return failure

Propagate Improvement(m)

1. neighbours ← Movegen(m).
2. For each s ∈ neighbours. → Edge distance.

do newGvalue ← g(n) + k(m, s)

if newGvalue < g(s)

then parent(s) ← m

~~g(s) = g(m) + k(m,s)~~

g(s) ← newGvalue

if s ∈ closed

then PropagateImprovement(s)

AO * Algorithm

Two types of graph

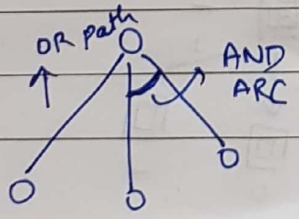
OR

→ For every node, there is choice of path.

→ All graph discussed till now are OR graphs

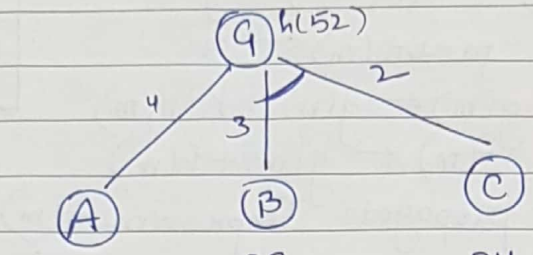
AND

→ There is AND arc and when AND arc is on two or more branches then there is deterministic solution and we have no choice. All paths need to be travelled in AND arc.



→ AO* is applicable on AND – OR paths.

AO* has two phases:
① Forward
② Backward



h : 42 22 24

→

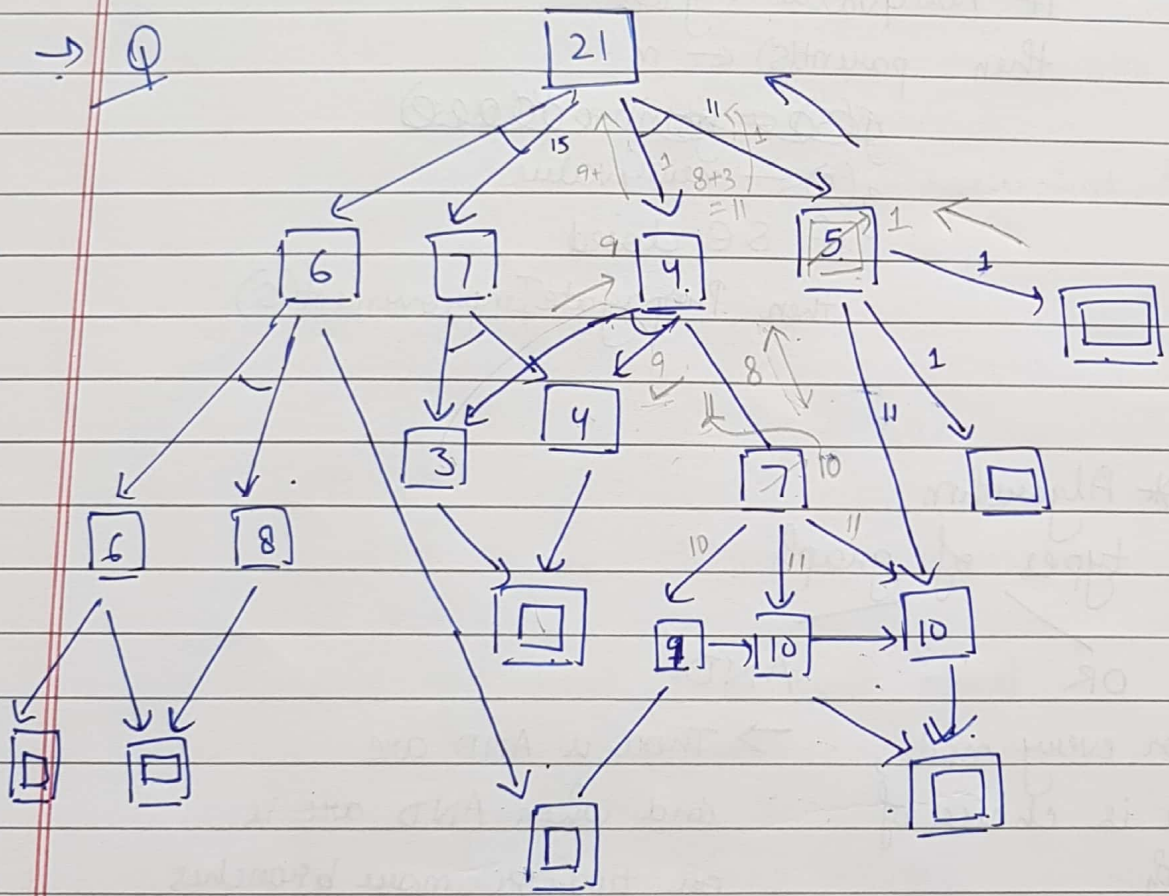
42 + 4 = 46

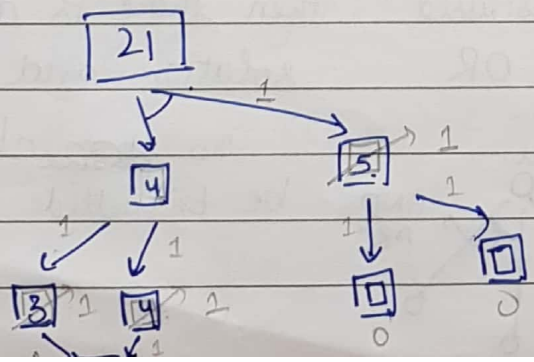
22 + 24 + 5 = 51.

→ Read algo from book

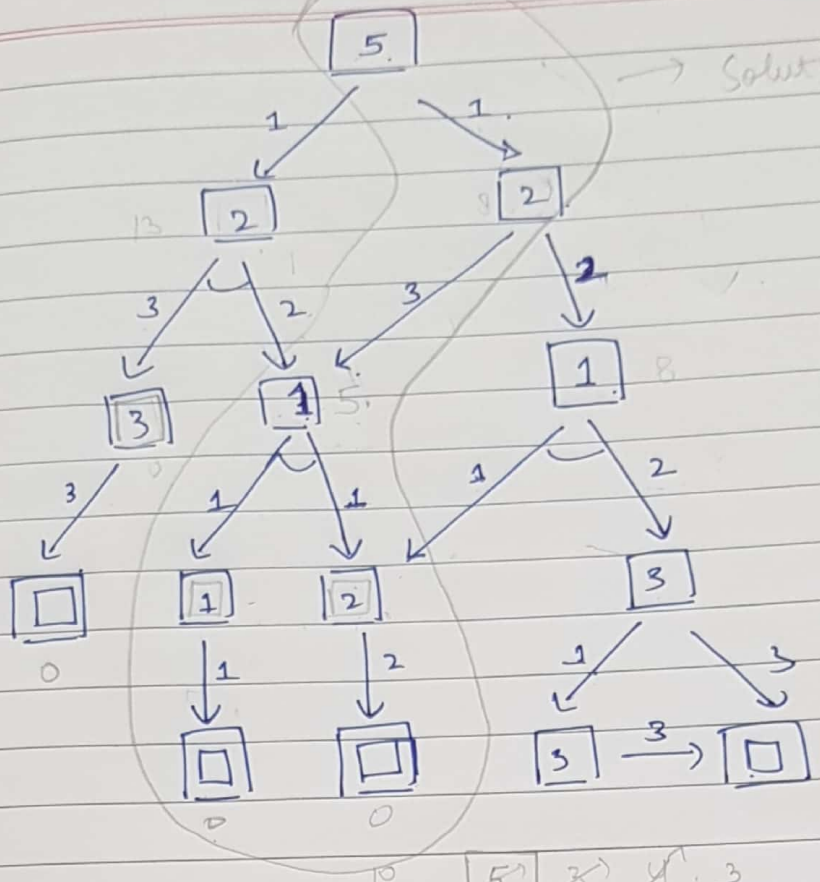

→ Sole Node
h = 0

→ Q





Q



→ Solution.

→