ELIZA: Proved machines are intelligent program

John Searel: Disproved

Page No. _____
Date

Introduction

AI: It is the study of how to make computers do things which at the moment people do better

The AI problems are classified into 3 tasks:

1) Mundane tasks: Daily routine tasks.
   eg. a) Perception: Vision, Speech
   b) Natural Language: Understanding, Generation, Translation
   c) Common Sense Reasoning
2) Formal tasks: a) Games: chess (Computer copies and predicts our moves after a few turns)
   b) Mathematics: Geometry, Logic, Integral Calculus, proving properties of program
3) Expert tasks: a) Engineering: Design, fault finding, Manufacturing & planning, Medical diagnosis, financial analysis

Intelligence: Thinking, decision making, problem solving, learning from experience, understanding, adaptation, reasoning, reaction to situations

There is no parameter to measure intelligence and hence it cannot be defined.

Fields of AI:

1) Smell i) Speech recognition
2) Image processing • ~~computerisation~~ Computer vision • Neural Networks
   • Knowledge discovery (data mining) • Pattern recognition
3) Natural Language understanding • NN generator
   • Quantitative reasoning fuzzy sets
   • Knowledge representation logic
   • Machine learning • Ontology • Semantics
   *. Search models • Robot control
   • Memory * Problem Solving • Planning
   • Adverse Serial learning
   • Handling uncertainty

Technique: AI problems span a very broad
ctrum. They appear to have very little in common except
t they are hard (Time complexity is exponential)

its of knowledge:
It is voluminous - Hard to categorize accurately
Constantly changing - It differs from data by being organized in a way that corresponds to the way it will be used.

technique in terms of knowledge: We are forced to
nclude that an AI technique is a method that
xploits knowledge that should be represented in such a
ay that-
The knowledge captures generalization. In other words it
not necessary to represent separately each individual
situation. Instead situations that share important properties
re grouped together.
. It can be understood by people who must provide it.
. It can easily be modified to correct errors and to reflect
changes in the world and in our world view.
4. It can be used in great many situations even if it is not
totally accurate or complete.
5. It can be used to help its own share bulk by helping to
narrow the range of possibilities that must usually be
considered.

Tic Tac Toe: 3 scenarios will be compared on basis of-
complexity, generalization, clarity of knowledge and
extensibility of approach.

2 DS: (Program 1)
Board: A 9 element vector representing the board, where the
elements of the vector corresponds to the board position
as follows:
0 → blank 1-X 2-O
Move table: It is a large vector of 19683 (3^9) elements,

Each square → 3 possibilities
9 squares
Total possibilities → 3^9

Page No. _____
Date | . | | |

each element of which is a 9 element vector:

Algorithm: To make a move, following steps:
1. View the board as ternary (base 3) no. convert it to a decimal no.
2. Use the no. computed in step 1 as an index into move table and access the vector stored there.
3. The vector selected in step 2 represents the way the board will look after the move that should be made.

Less time consuming, specialized approach, not much knowledge used, not extensible to 3D as no. of possibilities increase to 3^{27} (space complexity ↑)

Program 2: DS used:
Board: 9 element vector but instead of using 0, 1, 2 we use 2 → blank, 3 → X, 5 → O
Turn: It is an integer indicating which move of the game is about to be played. 1 → 1st move 9 → last move
Algorithm: 3 subprocedures:
1. Make 2: Returns 5 if the centre square of the board is blank. Otherwise it will return any blank non corner square. (0,1,3 / 7,5,8 / 6,9,4)
2. Posswin(p): It returns 0 if player p cannot win on his next move otherwise it returns a number of squares (position) that constitutes a winning move.
This function will enable the program both to win and to block the opponent's win.
Posswin operates by checking 1 at a time each of the rows, columns and diagonals. ~~Because~~ 3×3×2 5×5×2
possibility of wins.
. Go(n): It makes a move in square n. It sets
Board(n) = 3 if turn is odd
" 5 " " " even
It also increments turn by 1.

In 4th turn: at random any non-corner blank square

X	X	O
O	O	X
X	O	X

Page No. / Date

Turn (1) Go(1). (Assuming start)

Turn (2) Go(5) if blank otherwise Go(1)

Turn (3) Board(9) if blank Go(9) else Go(3)

Turn (4) If Posswin(x) != 0 then Go(Posswin(x))
else Go(make(2))

Turn (5) If Posswin(x) is not 0, Go(Posswin(x))
otherwise If Posswin(0) is not 0, Go(Posswin(0))
else if
Board(7) is blank Go(7) else Go(3)

Turn (6) If Posswin(0) is not 0, Go(Posswin(0))
otherwise if Posswin(x) is not 0,
Go(Posswin(x))
else
Go(make 2)

Turn (7) If Posswin(x) is not 0, then Go(Posswin(x))
else if Posswin(0) is not 0, Go(Posswin(0))
else find any position which is blank

Turn (8) If Posswin(0) is not 0, Go(Posswin(0))
else if Posswin(x) is not 0 Go "(x)"
else go anywhere where there is a blank.

Turn (9) Same as 7.

4th Turn decides who will win.

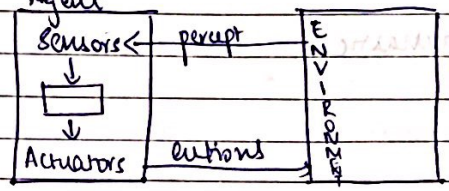
More time complexity, less space, specialized, clarity of knowledge, easier to extend to 3D compared to 4.

Program 2nd, Program 3, Many, Conclusions → From txt book

AGENTS: An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

Page No. / Date

	Sensors	Actuators
Humans	Eyes, ears, etc	Arms, legs, etc.
Robots	infrared rays	Motor
S/W	key strokes, file content, network packets	O/p screen, network packets, etc.



Percept Sequence: It is a complete history of everything the agent has ever perceived. An agent's choice of action at any given instance can depend on the entire percept sequence observed to date but not on anything it has not perceived.

Rational Agent: It is the one that does the right thing. 4 things are important:
- The performance measures that define the criteria of success
- The agent's prior knowledge of the environment
- The actions that agent can perform
- The agent's percept sequence to date

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measures, given the evidence provided by the percept sequence and whatever built in knowledge the agent has.

Omniscience Agent: It knows the actual outcome of its actions and can act accordingly.

Types of Environment:

1. Fully observable vs. Partially observable
   ↳ everything can be seen
2. Single Agent vs. Multiple Agent
   eg. cross word eg. chess (2)
3. Deterministic vs. Stochastic
   ↳ determinable according to current state & more can predict next state
4. Episodic vs. Sequential
   ↳ performance in different parts (episodes), individual parts ↳ all in one flow current state depends upon previous
5. Static vs. Dynamic
6. Discrete vs. Continuous
   ↳ finite no. of discrete states eg. taxi driving
   eg. chess
7. Known vs. Unknown
   ↳ outcome is given for every action of agents

Types of Agents:

1. Simple reflex agents: These agents select actions on the basis of current percept ignoring rest of percept history.

2. Model based reflex agents: The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now, i.e, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects atleast some of the unobserved aspects of the current state.

Goal based agents: As a current state description, the agents need some sort of goal information that describes situations that are desirable.

Utility based Agent: Goals alone are not enough to generate high quality behaviour in most environments. Goals just provide a crude binary distinction between happy & unhappy states. A more general performance measure allows a comparison of different world states according to exactly how happy they would make the agent.

Learning Agent: All the agents can improve their performance by learning. ⇒ constant feedback

Chapter – 2

Problem Solving Criteria

To build a system to solve a particular problem, we need to do 4 things:

1. Define the problem specifically / precisely: Definition must include precise specification of what the initial situations will be as well as what final situations constitute acceptable solutions to the problem.
2. Analyse the problem: A very few important features can have an immense effect on the appropriateness of various possible techniques for solving the problem.
3. Isolate & represent the task knowledge that is necessary to solve the problem
4. Choose the best problem solving techniques and apply it to the particular problems.

Water Problem

4

1. (x,y) If x<4 → (4,y)
2. (x,y) → If y<3 → (x,3)
3. (x,y) → (x-d,y)
   x>0
4. (x,y)
   y>0 → (x,y-d)
5. (x,y) → (0,y)
   x>0
6. (x,y) → (x,0)
   If y>0
7. (x,y) → (4, y-(4-x))
   If x+y ≥ 4 and y>0
8. (x,y)
   If (x+y) ≥ 3 and x>0 → (x-(3-y), 3)
9. (x,y) → (x+y, 0)
   If (x+y) ≤ 4 and y>0
10. (x,y) → (0, x+y)
    If (x+y) ≤ 3 and x>0
11. (0,2) → (2,0)
12. (2,y) → (0,y)

(0,0)

Rule 2. (0,0) → (0,3)
9 (0,3) → (3,0)
2 (3,0) → (3,3)
7 (3,3) → (4,2)
5 (4,2) → (0,2)
9 (0,2) → (2,0)

Learning Outcomes: (Must needed for implementation of state space search)

1. Define a state space that contains all the possible configurations of the relevant object. It is of course possible to define this state space without explicitly enumerating all of the states it contains.

2. Specify 1 or more states within that space that describes possible situations from which the problem solving process may start. These states are called initial states.

3. Specify 1 or more states that would be acceptable as solutions to the problem. These states are called goal states.

4. Specify a set of rules that describe the actions available.

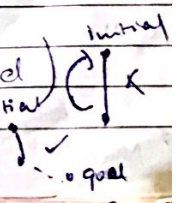
Production System: It consists of set of rules each consisting of left side that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.

One or more knowledge DB that contains whatever info is of priority for the particular task. Some parts of the DB may be permanent while other parts may pertain only to the solution of the current problem. The info in these DBs may be structured in any appropriate way.

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

Req. of a good control strategy:

1. It ~~pos~~ causes motion. (should move forward)



2. It should be systematic

Problem Characteristics:

1) Is the program decomposable?
eg. $\int (x + x^2 y)\,dx = \int x\,dx + \int x^2 dx$
Simple Blocksworld Problem
start: [C] on [A]; [A] on Table; [B] on Table
Two functions:
ON(x, Table)
ON(x, y)
Goal: A / B / C → Not decomposable problem

2) Can solution steps be ignored or undone?
8 puzzle problem
start state:

2	8	3
1	6	4
7		5

Goal state:

1	2	3
8		4
7	6	5

Recoverable Problem
Solving a Theorem through lemmas → ignored
Chess → Irrecoverable

3) Is it Universe predictable? (Generally, for single agent scenario)
Footballl → not predictable

4) Is the solution absolute or relative?

a. Marcus was a man
b. Marcus was Pompeian
c. " was born in 40 A.D.
d. All men are mortal
e. " Pompeians died when the volcano erupted in 79 A.D.
f. No mortal lives longer than 150 years
g. It is now 1991 A.D.

1) Is Marcus alive?
From a and d → Marcus is mortal
c and g → Marcus is 1991 − 40 = 1951 A.D
f → Marcus is dead.
g and e → All Pompeians dead
b. → Marcus is dead

Multiple paths
Absolute since same answer through any path
Each path is independent of each other

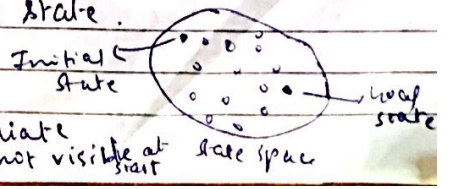
Travelling Salesman Problem:
↳ Relative Problem

5) Is the solution a state or a path?
Marcus problem → state problem (Path independent)
Water Jug Problem → Path (Interested in path which leads to goal state)

6) What is the role of knowledge?

State space Search:
Algorithm should be domain independent (can be designed for different set of problems)

We model the problem solving process as traversing a state space. The state space is a space in which each element is a state. A state is a description of the world in which the problem solver operates. The given situation is described by a state called start state. The desired or the goal situation is described by 1 or more goal states. In any given state, an action or a decision by an agent changes something and the agent moves to a new state. The task is to make sequence of moves such that the agent ends up being in a goal state.



Move operation

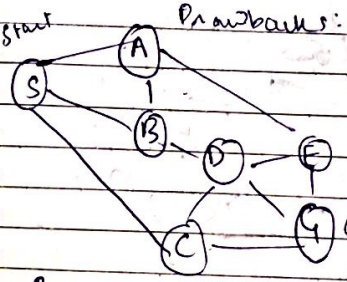
Generate & Test()
- While more candidates exist
  - do Generate a candidate
    Test whether it is the goal state
  - return Failure

River Crossing Puzzle:
A man needs to transport a lion, a goat & a cabbage across a river. He has a boat in which he can take only 1 of them at a time. It is only his presence that prevents the lion from eating goat & goat from eating cabbage. He can neither leave the goat alone with lion nor cabbage with goat. How can he take them all across the river.

Start state: ((MLGC) ()) (Two lists)
Goal state: (() (MLGC))

moveGen: → specific to a problem domain
1. Initialise set of successors C to an empty set.
2. Add m to the complement of the given state m to get the new state s.
3. If given state has left [L] then add right to s. else add left
4. If legal(s) then add s to the set of successors C.
5. For each other entity E in m make a copy s' of s. Add E to s'.
6. If legal(s') then add s' to C.
7. return to c

simpleSearch()
1. open ← {start}
2. while open is not empty
   do pick a node from open
   open ← open / {n}
   if GoalTest(n) = TRUE
   then return n
   else
   open ← open ∪ MoveGen(n)
- return Failure



Drawbacks:
1) Infinite loop
2) Redundant info.

Refining simple search:

Simple search 2()
1. open ← {start}
2. closed ← () (keeps track of visited nodes)
3. while open is not empty
4. do pick some node n from open
   open ← open / {n}
   closed ← closed ∪ {n}
   If GoalTest(n) = TRUE {
   return n
   else
   ~~open ← open ∪ {MoveGen(n)}~~ closed.
   open ← open ∪ {~~open~~ / MoveGen(n) / open}.
   open ← open / {closed(n)}
   return Failure

keeping track of path we using ?

SimpleSearch3()
- open ← {start}
closed ← { }
- while open is not empty
do pick some node n from open
h ← Head(n)
If GoalTest(h) == True
Return Reverse(h)
else closed ← closed ∪ {h}
successor ← MoveGen(h) / closed }
open ← successor ∪ open
for each s in successor
do Add cons(s, h) To open
return failure

open: (S..) (AS BS CS) BS, CS EAS, BS, CS
closed: ( ) (S) (A, S) (A, S)

successor

Modifying SimpleSearch3
DFS

1. open ← ((start NIL))
2. closed ← ( )
3. while not null(open)
do nodepair ← Head(open) — 1st pair of open goes into nodepair
node ← Head(nodepair) — 1st node nodepair goes into, node
if GoalTest(node) = True
then return
Reconstruct(nodepair, closed) — closed pair
else
closed ← cons(nodepair, closed)
children ← MoveGen(Node)
noloops ← RemoveSeen(open, children, closed)
new ← makepair(noloops, node)

SBF ADH
EA AS BS CS

Simple Search 3( ):

1. open ← {start}
2. closed ← { }
3. while open is not empty
   do pick some node n from open
   h ← Head(n)
   if GoalTest(h) = = True
   Return Reverse(h)
   else
   closed ← closed ∪ {h}
   successor ← More Gen(h) / closed
   successor ← successor / mapcar(open) ; open ← open{n,s}
   for each s in successor
   do add cons(n,s) to open
   return failure

// mapcar(open) gives head of nodes in open.

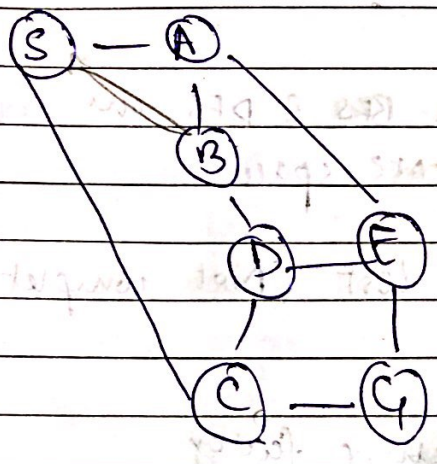
open	closed	successor
S	( )	
AS, BS, CS	S	A B C
~~A~~S, BS, CS	A, S	

→ new node will be added in front of tail.

Page No. _____
Date

- Open ← Append (new, Tail(open)) new node inserted in the front
  return "No solution"

eg.



Open	closed	no ops
(S Nil)		
(AS, BS, CS)	(S Nil)	A, B, C
(EA, BS, CS)	(AS, SNil)	E
(GE, DE, BS, CS)	(EA, AS, SNil)	

For BFS, modify • as
Open ← Append (Tail(open), new) → new node inserted at the end

Branching factor = 3 ⇒ every node will generate 3 nodes

4 factors on which BFS & DFS are compared:

a) completeness
b) Time complexity
c) Space "
d) Quality of solution

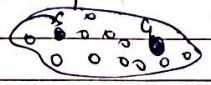
a) Does the algorithm always find a solution where there exists one?
b) How much time does the algorithm run before finding a solution? We will measure the time complexity by the no. of nodes the algo picks up before it finds the solution.
c) How much space does the algorithm need? We will use the no. of nodes in the open list as a measure of space complexity.

d) A simple criteria for quality might be the length of the path found from the start node to the goal node.

2 Scenarios: State space $\begin{cases} \text{finite} \\ \text{infinite} \end{cases}$

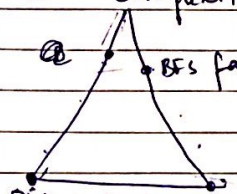
When the state space is finite, both BFS & DFS are complete (there is a goal state in state space).

State space infinite: BFS complete



DFS will get lost (Not complete)

Time complexity:



b → branching factor
d → depth.

If goal is on the extreme left then DFS finds d+1 nodes (If goal state is d+1).

Extreme right: $\frac{b^{d+1}-1}{b-1}$

Average, $N_{DFS} = \left(d+1 + \frac{b^{d+1}-1}{b-1}\right) / 2$.

$= \frac{(d+1)(b-1) + b^{d+1} - 1}{2(b-1)}$.

$= \frac{db + b - d - 1 + b^{d+1} - 1}{2(b-1)} \approx \frac{b^{d+1}}{2b}$ (d: very very larger b also)

$N_{BFS} = \left(\frac{b^{d}-1}{b-1} + \frac{b^{d+1}-1}{b-1}\right) / 2$ ∵ Internal nodes

(+1 can be ignored).

$= \frac{b^d - 2 + b^{d+1}}{2(b-1)} \approx \frac{b^d(b+1)}{2(b-1)} \approx \frac{b^d(b+1)}{2b}$

$\frac{N_{BFS}}{N_{DFS}} = \frac{b^d(b+1)}{b^{d+1}} = \frac{b+1}{b}$

⇒) BFS has slightly more complexity than DFS, otherwise same.

DFS:

Space Complexity: In general search tree with a branching factor b DFS dives down some branch at each level leaving b-1 nodes in open. When it enters the depth d it has O_{DFS} nodes and

$O_{DFS} = (b-1)(d-1) + b$

Size of open is linear with depth

BFS: No. of open nodes are exponential to depth

∴ Space complexity of BFS > DFS

Quality of solution: BFS gives better solution, it traverses node by node gives better solution even in worst case. (Measured in terms of no. of edges b/w goal & start states.) → Finds the minimal solution among multiple solutions

BFS & DFS: Blind algorithms, they have no intelligence, they explore the nodes in pre-defined order.

Heuristics: Prior known values based on some experience.

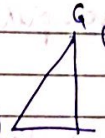
Heuristic function: The idea of using heuristic function is to guide the search so that it has a tendency to explore the region leading to the goal. A heuristic function is designed to help algorithm to pick next candidate from the open list that is most likely to be on the path to the goal. A heuristic value is computed for each of the candidates. The heuristic value could be an estimate of the distance to the goal of that node. The heuristic algorithm then simply has to choose the node with the lowest heuristic value to expand next.

The heuristic function must not be computationally expensive because the idea is to cut down on the computational cost of the search algorithm.

$h(n)$ $n \rightarrow$ node

Euclidean distance

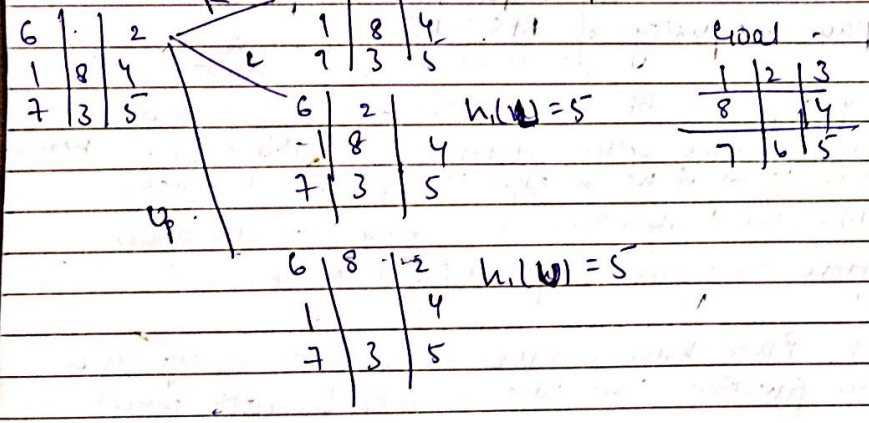
$h(n)$ = $\sqrt{(x_G - x_N)^2 + (y_G - y_N)^2}$



Manhattan distance:

$h(n) = |x_G - x_N| + |y_G - y_N|$

8 puzzle problem:



We are considering heuristic function as Manhattan distance.

1st function → $h_1(n)$ → To count the no. of tiles out of place.

2nd " → $h_2(n)$ → Adds the Manhattan distance of each tile from its destination.

Best First Search:

sort → Used to get nodes in ascending order of heuristic value.

Same as DFS algo but in the end

Open ← $sort_h$ (append (News, Tail(Open))

Completeness: Finite: Complete

Infinite: Depends on how good the heuristic function is.

Quality of solution: Not good.

Space: Heuristic function (Good/Perfect) → linear

" " Not good → exponential.

Time Complexity: Depends on Heuristic funct.

Parameters to decide among heuristic functions (which better):

1. Effective branching factor = total nodes expanded / Nodes in the solution (how much space explored)

Should be small. 1 → perfect. ↑, quality ↓ (No. of nodes b/w 0 & 5 in the path we have)

2. Penetrance (Inverse of branching factor)

Value tends to 1 ⇒ perfect.

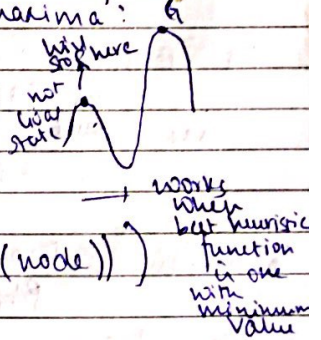
As it ↓, heuristic function quality ↓

Hill Climbing / Steepest Gradient Ascent Algo

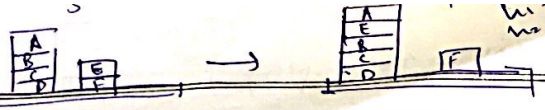
Open ← $sort_h$ (new)

```
Hill Climbing()
1. node ← start
2. newnode ← Head(sort_h(moveGen(node)))
3. while h(newnode) < h(node)
       do
          node ← newnode
          newnode ← Head(sort_h(moveGen(node)))
       return newnode
```

Problem of local maxima:



Blocks World:



$h_1 = 10$

1st heuristic function $h_1(n)$ → Checks whether each block is on correct block w.r.t final configuration. We add 1 for every block that is on the blok it is supposed to be on and subtract 1 for every one that is on the wrong one. Higher value better.

2nd function $h_2(n)$ → looks at the entire pile that the block is resting on. If the configuration of that pile is correct w.r.t goal, it adds 1 for every block in the pile or else it subtracts 1 for every block in that pile

Possible successors of S: Choose order ABCDEF for h

1.

$h_1 = 1+1+1-1-1+1$
$= 2$

$h_2 = 2+1+0-0-1+0$
$= 2$

2.

$h_1 = 1+1+1+1+0+1$
$= 4$

$h_2 = 2+1+0-2-1+0$
$= 0$

3.

$h_1 = -1-1+1+1+1+1$
$= 2$

$h_2 = -4-3+2+1$
$= -4$

4.

$h_1(n) = -1+1+1+1-1+1$
$= 2$

$h_2 = -3+2+1+0+0+0$
$= 0$

If we choose but heuristic function in terms of one which has max value, put - before each value for previous algo to work.

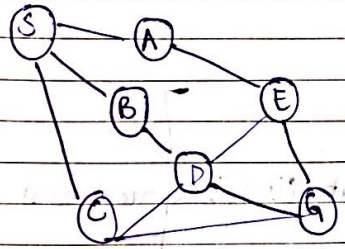
Completeness: Depends on Heuristic function.
Perfect: Complete Not Perfect: may get stuck in local maxima

Quality of solution: Depends on perfectness of Heuristic function.

Space Complexity: constant (all all nodes are overriden by nonegen in open).

Time Complexity: linear if heuristic function is perfect.

In Dijikshtra's algo, we calculate minimum distance from starting vertex. In best first search we calculate heuristic value of each child based on its distance from goal.



$f(n) = g(n) + h(n)$ — Hueristic function

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

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

$f^*(n)$ → actual but unknown cost of optimal path $S \to n \to G$.

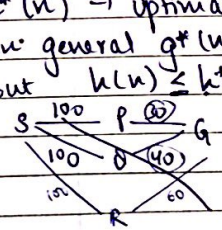
The evaluation function has two components: $g(n)$ & $h(n)$

$g(n)$ → inherited from branch and bound which is the known cost of the path from S to n. and it is a backward function

$h(n)$ → forward looking function which is inherited from best first search.

$g^*(n)$ → optimal cost from S to n } → May not be known.
$h^*(n)$ → optimal cost from n to G. }

In general $g^*(n)$ will be lower than $g(n)$ $g(n) \geq g^*(n)$
but $h(n) \leq h^*(n)$



$h_1(P) = 50$.
$h_1(Q) = 45$
$145 < 170$ will choose Q which is not optimal.

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

A* Algorithm:-

A* ()

1. open ← list(start)
2. f(start) ← h(start)
3. parent(start) ← NIL
4. closed ← { }
5. while open is not empty
6. do
7. Remove node n from open such that f(n) has the lowest cost
8. Add n to closed
9. If GoalTest(n) == true
10. Return (Reconstructpath(n))
11. neighbours ← MoveGen(n)
12. for each path m ∈ neighbours
13. do switch()

Case m ∉ ~~neig~~ open & m ∉ closed
Add m to open
parent(m) ← n
g(m) = g(n) + k(n, m) → From Dijkstra
f(m) = g(m) + h(m)

Case m ∈ open
If (g(n) + k(m,n)) < g(m)
then parent(m) ← n
g(m) = g(n) + k(n, m)
f(m) = g(m) + h(m)

Case m ∈ closed
If (g(n) + k(m,n)) < g(m)
then parent(m) ← n
g(m) = g(n) + k(n, m)
f(m) = g(m) + h(m)
propagate movement(m) //to update better path for children of m
return Failure

propagate : neighbours will be assigned Movegen(m) improvement

For each s ∈ neighbours
do
newvalue ← g(m) + k(m,n)
If newvalue < g(s)
then parent(s) ← m
g(s) ← newvalue
If s is closed
propagate improvement(s)