DEPARTMENT OF NETWORKING AND COMMUNICATIONS

LABRECORD

Academic Year: 2022-23

EVENSEMESTER

Programme (UG/PG): UG

Semester: VI

Course Code: 18CSC312J

Course Title: Artificial Intelligence and Cloud computing applications

Student Name: D.Bhargav

Register Number: RA2011028010069

Branch with Specialization: Computer Science and Engineering with Specialization in Cloud Computing

Section: K2

SCHOOLOFCOMPUTING

FACULTY OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SRM Nagar, Kattankulathur- 603203

Chengalpattu District

DEPARTMENTOFNETWORKINGANDCOMMUNICATIONS

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SRM Nagar, Kattankulathur - 603203
Chengalpattu District

Name- D.Bhargav

Year………**III**…………**Semester**………**VI**………**Branch**……**CSE - CC**…

University Register No. RA2011028010069

Certified that this is a Bonafide Record work done by the above student in the year 2022 - 2023.

Signature
Dr. Vaishnavi Moorthy
Course Faculty
Assistant Professor
Department of NWC

Signature
Dr. Annapurani Panaiyappan K
Head of the Department
Professor
Department of NWC

INDEX

Exp 1: Implementation of toy problems

Aim: To design program to solve camel and banana toy problem

Pseudocode :

```
bananas = 3000 position = 0 carrying = 0 while position
< 3000: if carrying > 0 and carrying * 1000 + position
>= 3000: position += carrying * 1000 carrying = 0
  if carrying == 0 and bananas > 0: bananas -= 1
  if carrying == 0 and position < 3000:
    carrying = 1000
  if carrying > 0 and carrying * 1000 + position >= 3000: position
    += carrying * 1000
    carrying = 0
  position += 1
```

Optimisation Steps:

We have a total of 3000 bananas.
The destination is 1000KMs Only 1
mode of transport.

Camel can carry a maximum of 1000 bananas at a time. Camel eats
a banana every km it travels.

Source————IP1—————IP2——————-Destination

3000 x km 2000 y km 1000 z km

```
——————> |   —————> | ———————>
<——————- |  <————— |
```

——————->| —————>|

<—————— | |

——————-> | |

3000 – 5x = 2000 so we get x = 200

2000-3y = 1000 so we get y = 333.33 but here the distance is also the number of bananas and it cannot be fraction so we take y =333 and at IP2 we have the number of bananas equal 1001, so its 2000-3y = 1001

So the remaining distance to the market is 1000 -x-y =z i.e 1000-200-333 => z =467.

Now, there are 1001 bananas at IP2. However the camel can carry only 1000 bananas at a time, so we need to leave one banana behind.

So from IP2 to the destination point the camel eats 467 bananas. The remaining bananas

are 1000467=533 **Output:**

Result: The Total Bananas Delivered were 533 i.e 17.7 % efficiency

Exp 2: Developing agent programs for real world problems

Aim: To Develop agent program for graph coloring problem

Pseudocode: function graph_coloring(Graph

```
G, int k):
    // G is the graph to be colored
    // k is the maximum number of colors to be used

    for each vertex v in G:
        v.color = null // Initialize all vertex colors to

    null  for each vertex v in G: if v.color == null:

            // Use a set to keep track of the colors used by neighboring vertices
            used_colors = set()

            // Loop over neighboring vertices and add their colors to
            used_colors for each neighboring vertex u of v: if u.color is not null:
                    used_colors.add(u.color)

            // Find the first color not in used_colors
            for i in range(k):
                if i not in used_colors:
                    v.color = i break return G //
```

Return the colored graph

Optimisation Steps:

Lets take :

```
   1---2
/ \     \
3---4---5
```

```
 1:null---2:null
|         |
 3:null---4:null---5:null

 1:0---2:null
|         |
 3:null---4:null---5:null
    v
 used_colors = {0}

 1:0---2:null
|         |
 3:null---4:null---5:null
       v
 used_colors = {}

 4:1

 1:0---2:null
|         |
 3:null---4:1---5:null
       v
 used_colors = {1}

 1:0---2:null
|         |
 3:2---4:1---5:null
       v
 used_colors = {1, 2}

 1:0---2:2
|         |
 3:2---4:1---5:null
       v
 used_colors = {1, 2, 0}
```

1:0---2:2

| |

3:2---4:1---5:0 v
used_colors =

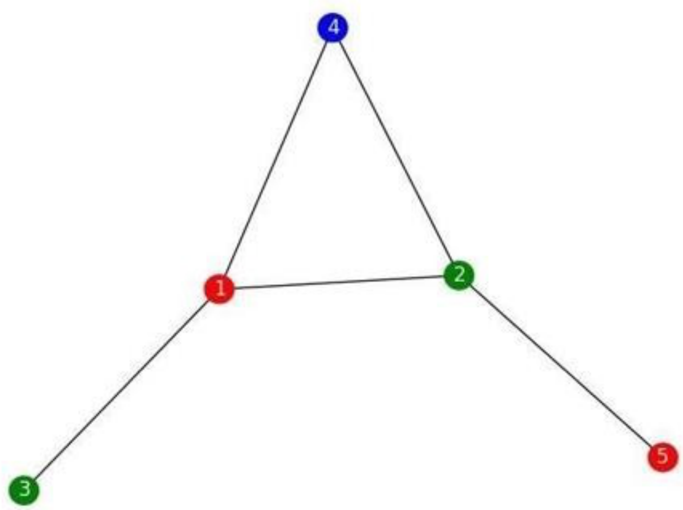
{0} 1:0---2:2

| |

3:2---4:1---5:0

The chromatic number is 3 **Output:**

```
Enter the number of vertices: 5
Enter the number of edges: 5
Enter the vertices for edge 1 (separated by a space): 1 2
Enter the vertices for edge 2 (separated by a space): 1 3
Enter the vertices for edge 3 (separated by a space): 1 4
Enter the vertices for edge 4 (separated by a space): 2 4
Enter the vertices for edge 5 (separated by a space): 2 5
```



Result: The Chromatic number is 3

Exp 3: Implementation of constraint satisfaction problems

Aim: To design a constraint satisfaction problem such as cryptarithmetic and sudoku solver

Pseudocode :

Cryptarithmetic:

```
function solve_crypt_arithmetic(puzzle) : # Split the
   puzzle into the operands and the result operands =
   puzzle[:len(puzzle)-2].split("+") result =
   puzzle[len(puzzle)-1]

   # Get the unique letters from the puzzle letters =
   set(puzzle.replace("+", "").replace("=", ""))

   # Generate all possible combinations of digits for the unique letters
   digits = [i for i in range(10)] digit_combinations =
   itertools.permutations(digits, len(letters))

   # Check each combination of digits to see if it solves the
   puzzle for combination in digit_combinations: digit_map =
   dict(zip(letters, combination))

      # Check if the combination results in a valid solution if
            sum([int("".join([str(digit_map[char]) for char in op])) for op in operands]) ==
int("".join([str(digit_map[char]) for char in result])):
         return digit_map

   # If no solution is found, return None
return None
```

Sudoku solver:

```
function solve_sudoku(board) # Find the next
   empty cell on the board row, col =
   find_empty_cell(board)
   # If there are no more empty cells, the board is solved
   if row is None: return True
```

```
# Try filling the cell with each number from 1 to 9 for
num in range(1, 10):
    # Check if the number is a valid choice for the
    cell if is_valid_choice(board, row, col, num): #
    Fill the cell with the number board[row][col] =
    num

        # Recursively try to solve the board with the new number added
        if solve_sudoku(board): return True

        # If the new number does not lead to a solution, backtrack and try the next number
        board[row][col] = 0

# If none of the numbers lead to a solution, return False
return False
```

Optimisation Steps:

H E R E

+ S H E

C O M E S

H

+ _

C O

H + 1 = 9 + 1 = 0 (1 carry to next step)

9 E R E

+ S 9 E

----

1 0 M E S

Now, they have already given S value as 8

Now, E + E = 8, thus, E = 4

Let's find value R

R + 9 = E after substituting E value R + 9 = 4 Thus, R = 5 and (1 carry to next step) Lets, find the value of M

E + S + 1 (Carry) = M after substituting the value 4 + 8 + 1 = 3 (1 carry to next step) The final values are –

H = 9, E = 4, R = 5, S = 8, C = 1, O = 0, M = 3

Output :

```
+-----+-----+-----+
|7|4|8|6|3|5|2|9|1|
|1|9|3|4|7|2|6|5|8|
|6|5|2|8|1|9|4|3|7|
+-----+-----+-----+
|2|6|5|9|4|8|7|1|3|
|8|7|9|1|2|3|5|4|6|
|3|1|4|7|5|6|8|2|9|
+-----+-----+-----+
|9|2|7|3|6|4|1|8|5|
|5|3|6|2|8|1|9|7|4|
|4|8|1|5|9|7|3|6|2|
+-----+-----+-----+
```

```
Enter the number of elements in the array: 2
Enter element 1: HERE
Enter element 2: SHE
Enter the result: COMES
---------------------------------------------------------
HERE  5939
SHE  859
COMES  06798
---------------------------------------------------------
---------------------------------------------------------
HERE  9454
SHE  894
COMES  10348
---------------------------------------------------------
---------------------------------------------------------
HERE  2858
SHE  628
COMES  03486
---------------------------------------------------------
---------------------------------------------------------
HERE  3959
SHE  839
COMES  04798
---------------------------------------------------------
---------------------------------------------------------
HERE  2969
SHE  829
COMES  03798
---------------------------------------------------------
---------------------------------------------------------
HERE  5494
SHE  854
COMES  06348
---------------------------------------------------------


...Program finished with exit code 0
Press ENTER to exit console.
```

Result: Multiple solutions were found for constraint satisfaction programs.

Exp 4: Implementation and Analysis of DFS and BFS for an application

Aim: To design and implement Breadth first search and Depth first search

Pseudocode :

BFS: create a queue Q

mark v as

visited and put v into Q while Q is non-empty

remove the head u of Q mark and enqueue all (unvisited)

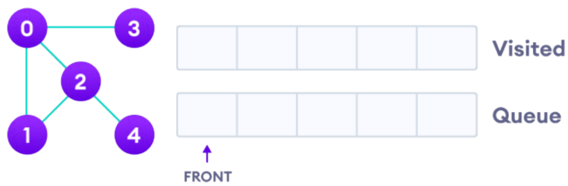
neighbours of u

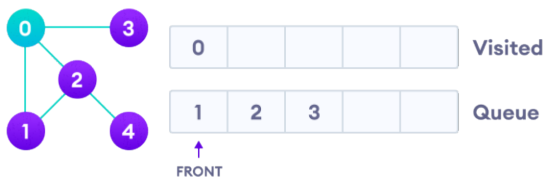
DFS:

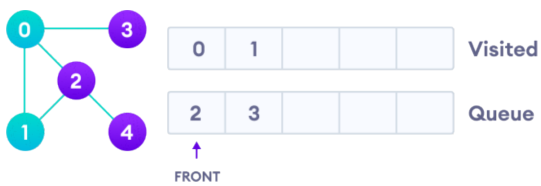
```
DFS(G, u)
    u.visited = true for
    each v ∈ G.Adj[u] if
    v.visited == false
    DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

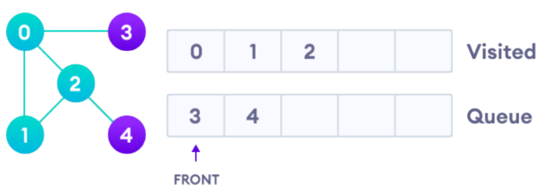
Optimisation Steps:

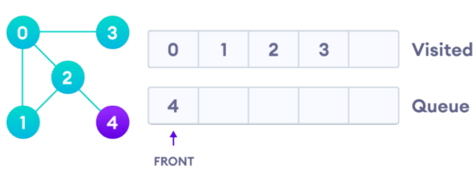
BFS:

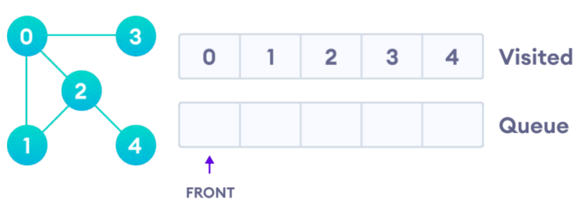




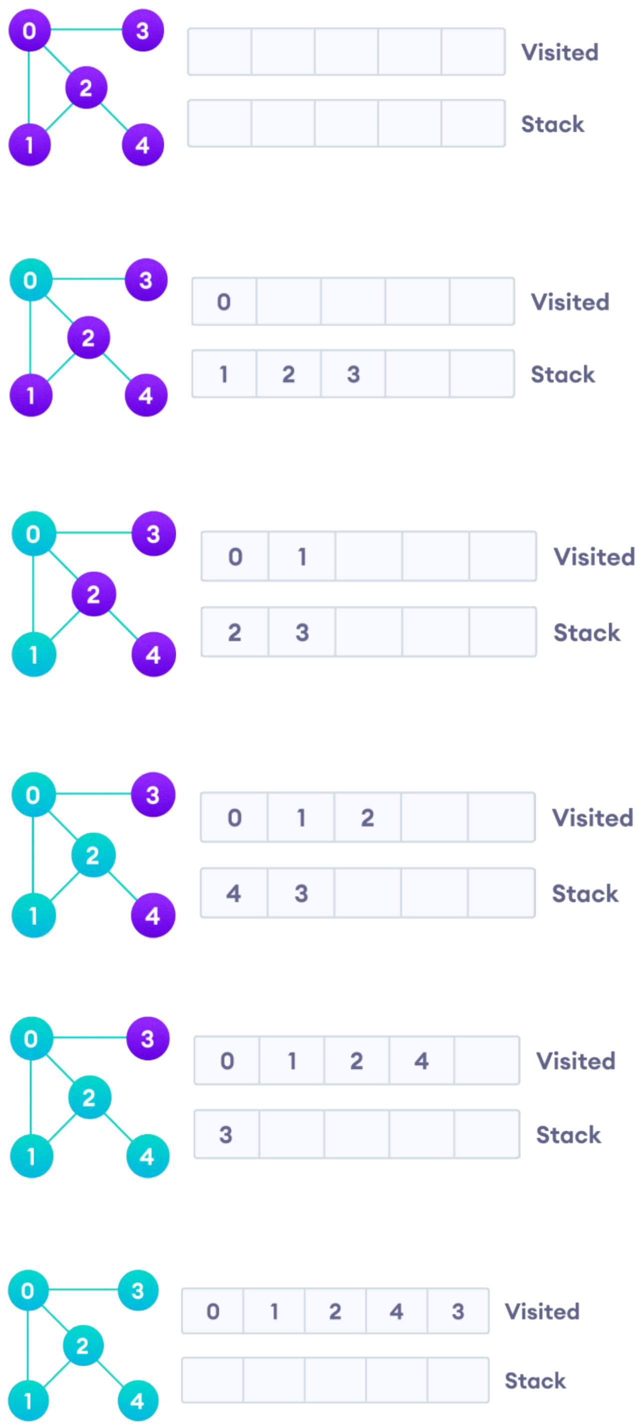








DFS:


Visited
Stack
0
Visited
1 2 3
Stack
0 1
Visited
2 3
Stack
0 1 2
Visited
4 3
Stack
0 1 2 4
Visited
3
Stack
0 1 2 4 3
Visited
Stack

Output:

```
Enter the tree as a list of edges, with each edge represented by a pair of nodes sep
arated by a space (e.g. '1 2'). Enter 'done' when finished:
0 1
0 3
0 2
1 2
2 4
done
Enter the target node value: 4
DFS is faster than BFS by 0.14 ms
```

Result: DFS is faster than BFS by 0.14ms

Exp 5: Developing Best first search and A* Algorithm for real world problems

Aim: To design best first search and A* program

Pseudocode :

Best first search: function best_first_search(initial_state, goal_test, successors, heuristic) returns solution or failure frontier <- Priority Queue containing initial_state explored <- empty set

while not frontier.empty() do state
<- frontier.pop() if goal_test(state) then return solution

explored.add(state) for successor in successors(state) do if

successor not in explored and successor not in frontier then

frontier.push(successor, heuristic(successor))

else if successor in frontier and heuristic(successor) < frontier.get_priority(successor) then frontier.replace(successor, heuristic(successor))

return failure

A*:

Input: A graph *G(V,E)* with source node *start* and goal node *end*.

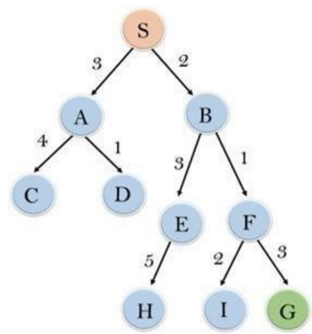
Output: Least cost path from *start* to *end*.

Steps:

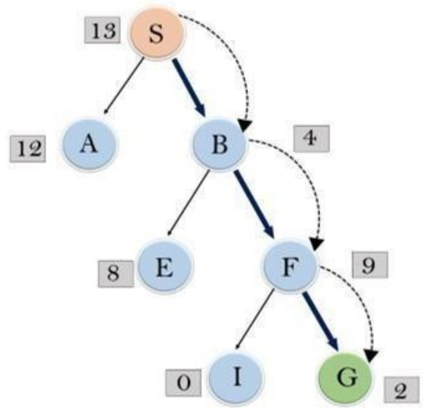
```
Initialise
        open_list = { start }                        /* Listof nodes to be traversed*/
        closed_list = { }                            /*List of already traversed nodes*/
        g(start) = 0                                 /*Cost from source node  to a node*/
        h(start) = heuristic_function (start,end)    /*Estimated cost from node to goal node*/
        f(start) = g(start)+h(start)                 /* Total cost from source to goal node*/

while open_list is not empty
        m = Node on top of open_list, with least f
        if m = = end
                return
        remove m from open _list
        add m to closed_list
        for each n in  child(m)
                if n in closed_list
                        continue
                cost= g(m)+ distance(m,n)
                if n in open_list and cost < g (n)
                        remove n from open_list as new path is better
                if n in closed_list and cost < g(n)
                        remove n from closed_list
                if n not in open_list and n not in closed_list
                        add n to open_list
                        g(n) = cost
                        h(n) = heuristic_function (n,end)
                        f(n) = g(n)+h(n)
return failure
```

Optimisation Steps:

Best first search



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



Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B] :

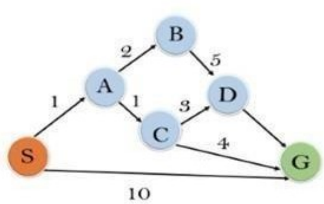
Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

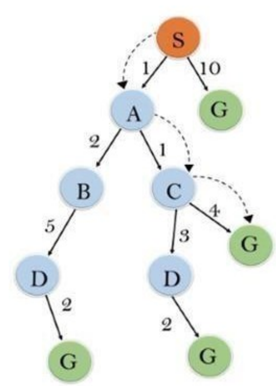
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: S----> B----->F----> G

The worst case time complexity of Greedy best first search is O(bm). **A***



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



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)} Iteration

4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6

Output:

```
Enter the number of nodes: 5
Enter node 1: A
Enter neighbors of A (comma-separated): B,C
Enter node 2: B
Enter neighbors of B (comma-separated): A,C,D
Enter node 3: C
Enter neighbors of C (comma-separated): A,B,D,E
Enter node 4: D
Enter neighbors of D (comma-separated): B,C,E
Enter node 5: E
Enter neighbors of E (comma-separated): C,D
Enter node 1: A
Enter heuristic value for A: 8
Enter node 2: B
Enter heuristic value for B: 6
Enter node 3: C
Enter heuristic value for C: 4
Enter node 4: D
Enter heuristic value for D: 4
Enter node 5: E
Enter heuristic value for E: 2
Enter the start node: A
Enter the goal node: E
```

```
Goal node E found!
Path: A -> C -> E
```

```
Enter the number of nodes: 5
Enter node 1: A
Enter neighbors of A (comma-separated): B,D
Enter node 2: B
Enter neighbors of B (comma-separated): A,C
Enter node 3: C
Enter neighbors of C (comma-separated): B,D,E
Enter node 4: D
Enter neighbors of D (comma-separated): A,C,E
Enter node 5: E
Enter neighbors of E (comma-separated): C,D
Enter node 1: A
Enter heuristic value for A: 8
Enter node 2: B
Enter heuristic value for B: 4
Enter node 3: C
Enter heuristic value for C: 3
Enter node 4: D
Enter heuristic value for D: 3
Enter node 5: E
Enter heuristic value for E: 0
Enter the start node: A
Enter the goal node: E
Goal node E found!
Path: A -> D -> E


...Program finished with exit code 0
Press ENTER to exit console.
```

Result: The worst case time complexity of Greedy best first search is O(bm).The time complexity of A* search algorithm is O(b^d), where b is the branching factor.

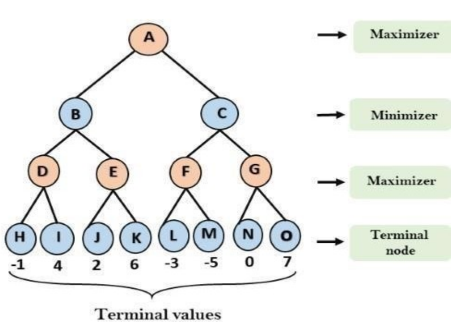
Exp 6: Implementation of minimax algorithm for an application

Aim: To implement minimax algorithm

Pseudocode:

```
function minimax(node, depth, maximizingPlayer)
   if depth == 0 or node is a terminal node return the
   heuristic value of node if maximizingPlayer
   bestValue = -infinity
      for each child of node
         v = minimax(child, depth - 1, FALSE)
         bestValue = max(bestValue, v)
      return bestValue

   else // minimizing player
      bestValue = +infinity
      for each child of node
         v = minimax(child, depth - 1, TRUE)
         bestValue = min(bestValue, v)
      return bestValue
```

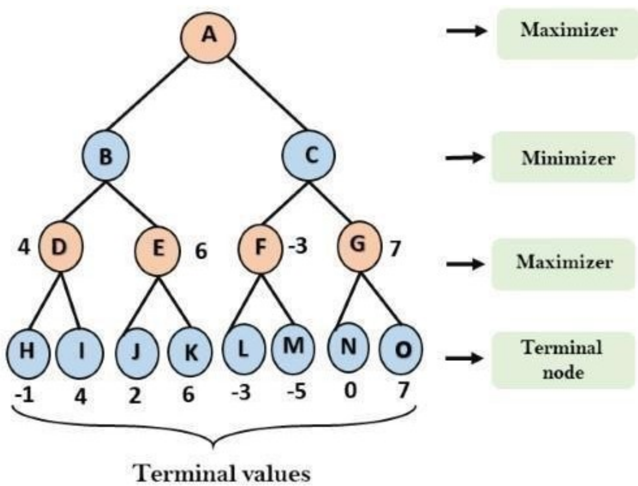
Optimisation Steps :



For node D max(-1,- -∞) => max(-1,4)= 4

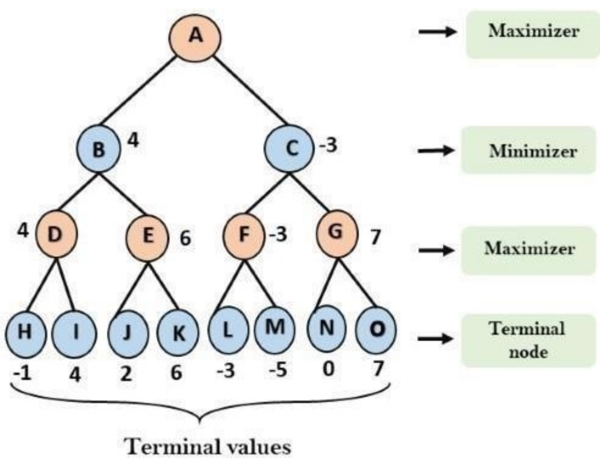
For Node E max(2, -∞) => max(2, 6)= 6

For Node F max(-3, -∞) => max(-3,-5) = -3 For node G max(0, -∞) = max(0, 7) = 7

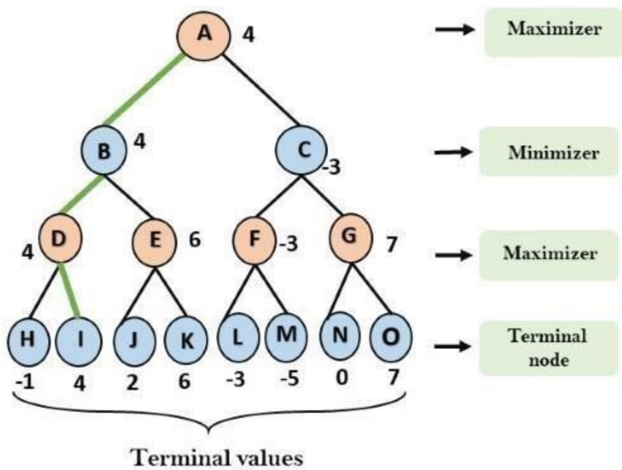


For node B= min(4,6) = 4

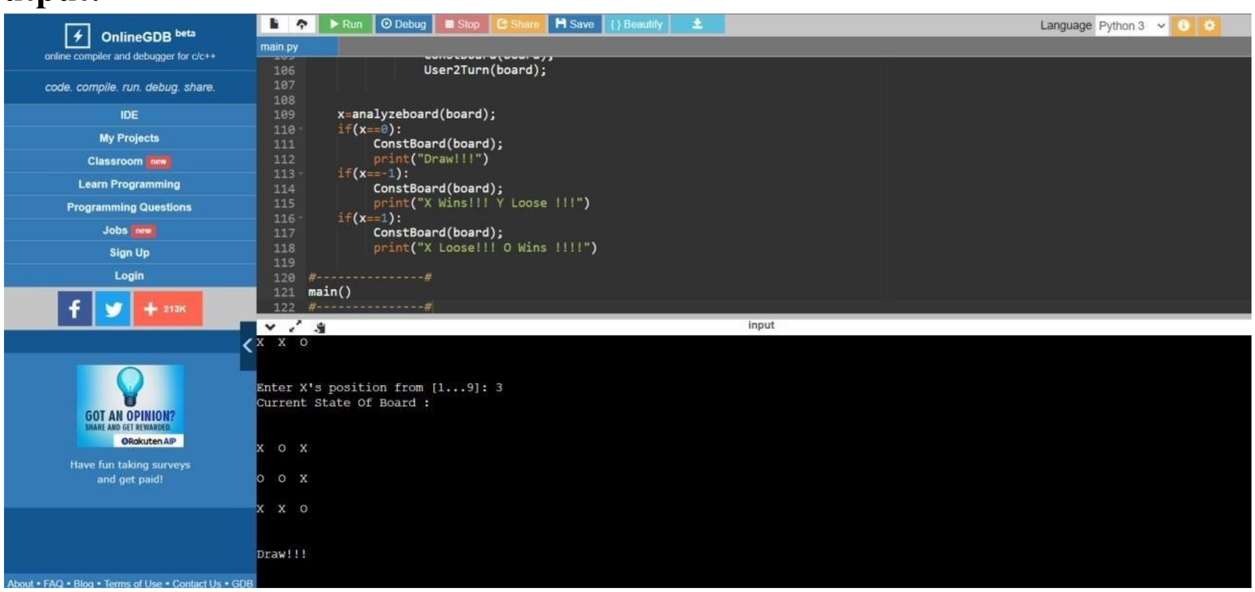
For node C= min (-3, 7) = -3



For node A max(4, -3)= 4



Output:



Result: The minimax algorithm was designed and tic tac toe game was designed

Exp 7: Implementation of unification and resolution for real world problems

Aim: To implement unification and resolution of in real life problem

Pseudocode :

Unification:

If Ψ1 or Ψ2 is a variable or constant, then:

If Ψ1 or Ψ2 are identical, then return NIL.
Else if Ψ1 is a variable, then if Ψ1 occurs in Ψ2,
then return FAILURE Else return { (Ψ2/ Ψ1)}.

Else if Ψ2 is a variable,
If Ψ2 occurs in Ψ1 then return FAILURE,
Else return {(Ψ1 / Ψ2)}.

Else return FAILURE.

If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.

IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.

Set Substitution set(SUBST) to NIL.

For i=1 to the number of elements in Ψ1.

Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.

If S = failure then returns Failure

If S ≠ NIL then do,

Apply S to the remainder of both L1 and L2.
SUBST= APPEND(S, SUBST). Return

SUBST.

Resolution:

function resolution(KB, α): clauses
= KB ∧ ¬α new
= { } while True: for

ci in clauses: for cj in

clauses: if ci is not

cj:

resolvents = resolve(ci, cj) if
False in resolvents:
return True
new = new ∪ resolvents if

new is subset of clauses: return

False clauses = clauses ∪ new

Optimisation Steps:

a. ∀x: food(x) → likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)

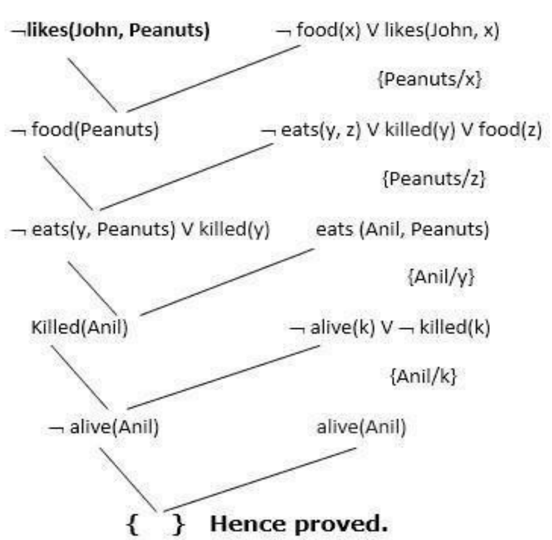
d. eats (Anil, Peanuts) Λ alive(Anil).

e. ∀x : eats(Anil, x) → eats(Harry, x)

f. ∀x: ¬ killed(x) → alive(x) } **added predicates.**

g. ∀x: alive(x) →¬ killed(x) }

h. likes(John, Peanuts)



1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S0 => Here, $\Psi 1$ = p(f(a), g(Y)), and $\Psi 2$ = p(X, X)

SUBST θ = {f(a) / X}

S1 => $\Psi 1$ = p(f(a), g(Y)), and $\Psi 2$ = p(f(a), f(a))

SUBST θ = {f(a) / g(y)}, Unification failed.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

S0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}

SUBST θ={b/Z}

S1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}

SUBST θ={f(Y) /X}

S2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))} SUBST

θ= {g(b) /Y}

S2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} Unified Successfully. And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, $\Psi 1$ = {p (X, X), and $\Psi 2$ = p (Z, f(Z))

S0 => {p (X, X), p (Z, f(Z))}

SUBST θ= {X/Z}

S1 => {p (Z, Z), p (Z, f(Z))}

SUBST θ= {f(Z) / Z}, Unification Failed.

Therefore, unification is not possible for these expressions.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)} Here, $\Psi 1$ = Q(a, g(x, a), f(y)), and $\Psi 2$ = Q(a, g(f(b), a), x)

S0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}

SUBST θ= {f(b)/x}

S1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}

SUBST θ={f(Y) /X}

S2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}

S1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, Successfully Unified.

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ1 = knows(Richard, x), and Ψ2 = knows(Richard, John)

S0 => { knows(Richard, x); knows(Richard, John)}

S SUBST θ= {John/x}

S1 => { knows(Richard, John); knows(Richard, John)}, Successfully Unified.

Unifier: {John/x}.

Output:

```
top/MISC/Codes/aiexp7.py
John does not like Italian food
```

Result: Implementation was successful and output was obtained

Exp 8: Implementation of knowledge representation schemes

Aim: To construct knowledge base and apply inference in FOL , Generate query based family relationship

Pseudocode:

person = [] parent = [] male = [] female = [] mother(x, y): return True if x is the mother of y and False otherwise. father(x, y): return True if x is the father of y and False otherwise.

grandparent(x, y): return True if x is a grandparent of y and False otherwise.

sibling(x, y): return True if x and y are siblings and False otherwise.

aunt(x, y): return True if x is the aunt of y and False otherwise.
uncle(x, y): return True if x is the uncle of y and False otherwise.
cousin(x, y): return True if x and y are cousins and False otherwise.

Optimisation Steps:

Enter the number of people: 5

Enter the name of person 1: Alice

Enter the gender of Alice (M/F): F

Enter the name of person 2: Bob

Enter the gender of Bob (M/F): M

Enter the name of person 3: Carol

Enter the gender of Carol (M/F): F

Enter the name of person 4: David

Enter the gender of David (M/F): M

Enter the name of person 5: Eve Enter the

gender of Eve (M/F): F person = ['Alice', 'Bob', 'Carol', 'David', 'Eve'] male = ['Bob',

'David'] female = ['Alice',

'Carol', 'Eve']

Enter the name of parent 1: Alice
Enter the name of child 1: Bob

Enter the name of parent 2: Alice

Enter the name of child 2: Carol

Enter the name of parent 3: Bob

Enter the name of child 3: David

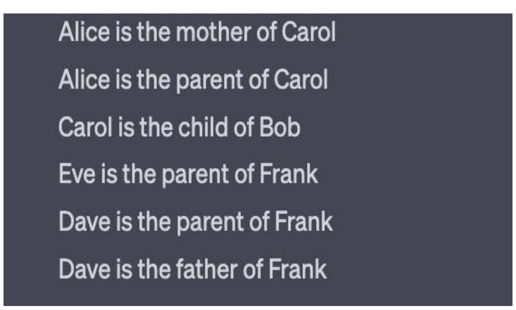
Enter the name of parent 4: Carol Enter the name of child 4: Eve parent

= [('Alice', 'Bob'), ('Alice', 'Carol'), ('Bob', 'David'), ('Carol', 'Eve')]

Output:

```
Enter the name of person 1: Alice
Enter the gender of Alice (M/F): F
Enter the name of person 2: Bob
Enter the gender of Bob (M/F): M
Enter the name of person 3: Carol
Enter the gender of Carol (M/F): F
Enter the name of person 4: David
Enter the gender of David (M/F): M
Enter the name of person 5: Eve
Enter the gender of Eve (M/F): F
```

```
Enter the name of parent 1: Alice
Enter the name of child 1: Bob
Enter the name of parent 2: Alice
Enter the name of child 2: Carol
Enter the name of parent 3: Bob
Enter the name of child 3: David
Enter the name of parent 4: Carol
Enter the name of child 4: Eve
```



```
Alice is the mother of Carol
Alice is the parent of Carol
Carol is the child of Bob
Eve is the parent of Frank
Dave is the parent of Frank
Dave is the father of Frank
```

Result: The result was obtained.

Exp 9: Implementation of uncertain methods for an application

Aim: To implement fuzzy logic in Monty Hall Problem

Pseudocode:

```
import random

A = "A"

B = "B"

C = "C"

doors = ["A", "B", "C"]

prize = random.choice(doors)

selection = input("Select door 'A', 'B', or 'C': ")

print("This problem relies on conditional probabilities.") print("It is suggested that you switch
doors, you will have a higher probability of winning of you
do.")
```

```
if selection == prize:     remaining =

list(set(doors) - set(selection))

    open_door = random.choice(list(set(doors) - set(prize) - set(random.choice(remaining))))

    alternate = list(set(doors) - set(open_door) - set(selection))[0] else:
    open_door = random.choice(list(set(doors) - set(selection) - set(prize)))

    alternate = list(set(doors) - set(open_door) - set(selection))[0]



print("The door I will now open is: %r" % open_door)



second_chance = input("Would you like to select the third door? Type 'Yes' or 'No': ")

if second_chance == "Yes":     print("The door you will

switch to is: %r" % alternate)

    if alternate == prize:

        print("Congrats, you win! The prize was behind your original selection, %r" % selection)

    else:          print("Sorry, the prize was behind the original door %r"
% prize)



if second_chance != "Yes":     print("You decided to keep your
```

initial door, %r" % selection)

    if selection != prize:

        print("Sorry, the prize was behind the original door, %r" % prize)
    else:

        print("Congrats, you win! The prize was behind your original selection, %r" % selection)

print("This is a check:")

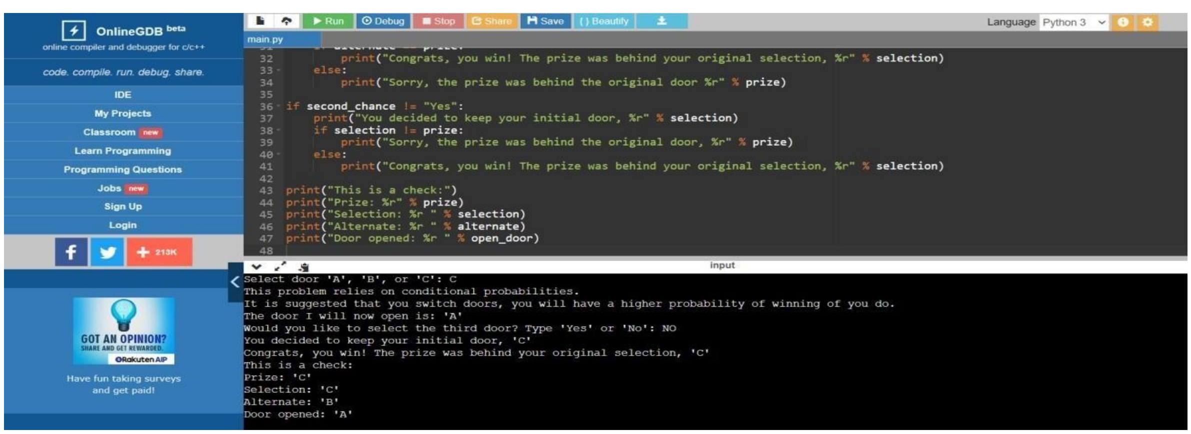
print("Prize: %r" % prize)

print("Selection: %r " % selection)

print("Alternate: %r " % alternate) print("Door

opened: %r " % open_door)

Output:



Result: The chlorine level is 8.0

Exp 10: Implementation of block world problem

Aim: To implement block world problem

Pseudocode:

world_state = initialize_world_state() actions

= ['MOVE', 'STACK', 'UNSTACK']

def is_valid_action(action, state):

check if the action is valid based on the current state of the

world Return true def perform_action(action, state):

perform the given action on the world state

Return (block,source,target,state) goal_state =

define_goal_state()

perform search to find a sequence of actions that transforms the initial state into the goal state

search_alg(world_state, goal_state, actions, is_valid_action, perform_action) //heap run_time =

str(end-start)*1000

Optimisation Steps :

Initial state:

Blocks: A, B, C, D

On table: A

On blocks: B on A, C on B, D on C Goal

state:

Blocks: A, B, C, D

On table: C, B

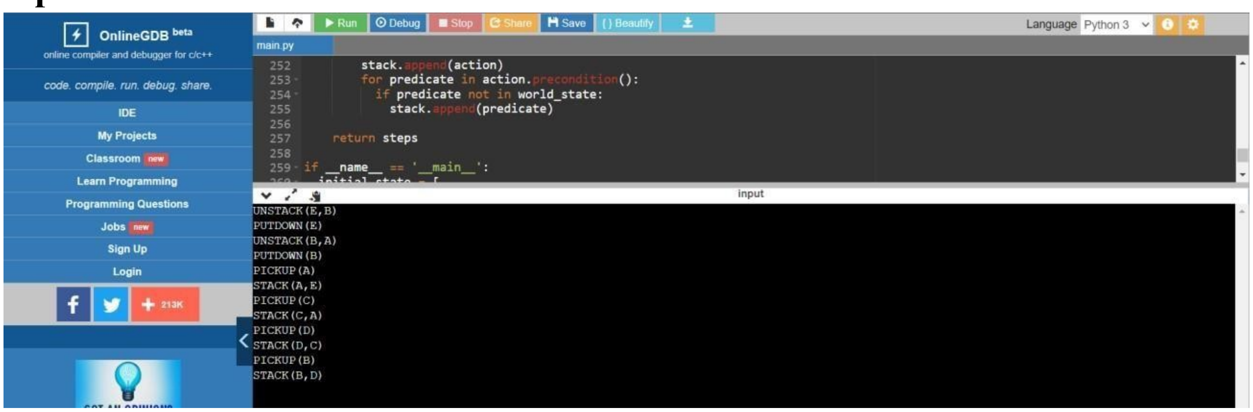
On blocks: A on C, B on D

Pick up block D and place it on the table. Pick up block C and place it on the table.

Pick up block B and place it on top of block D. Pick
up block C and place it on top of block A. Pick up
block A and place it on top of block C.

Pick up block B and place it on top of block D.

Output:



Result: The block world problem was implemented with running time of 5.08618 ms
.