Color a Graph

The steps required to color a graph G with n number of vertices are as follows –

Step 1 – Arrange the vertices of the graph in some order.

Step 2 – Choose the first vertex and color it with the first color.

Step 3 – Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

Dfs

We will create the graph for which we will use the depth-first search.

2. After creation, we will create a set for storing the value of the visited nodes to keep track of the visited nodes of the graph.

3. After the above process, we will declare a function with the parameters as visited nodes, the graph itself and the node respectively.

4. And inside the function, we will check whether any node of the graph is visited or not using the "if" condition. If not, then we will print the node and add it to the visited set of nodes and add it to the other list we have created.

5. Then we will go to the neighboring node of the graph and again call the DFS function to use the neighbor parameter.

6. At last, we will run the driver code which prints the final result of DFS by calling the DFS the first time with the starting vertex of the graph and implement a lambda function.

bfs

We will create the graph for which we will use the breadth-first search.

2. After creation, we will create two lists, one to store the visited node of the graph and another one for storing the nodes in the queue.

3. After the above process, we will declare a function with the parameters as visited nodes, the graph itself and the node respectively. And inside a function, we will keep appending the visited and queue lists.

4. Then we will run the while loop for the queue for visiting the nodes and then will remove the same node.

5. At last, we will run the for loop to check the not visited nodes and then append the same from the visited and queue list and add them to the sum

6. As the driver code, we will call the user to define the bfs function with the first node we wish to visit.

TOWER OF HANOI

```
START
```

```
Procedure Hanoi(disk,source, dest, aux)
```

```
   IF disk == 1, THEN
```

```
      move disk from source to dest
```

```
   ELSE
```

```
      Hanoi(disk - 1, source, aux, dest)
// Step 1
```

```
      move disk from source to dest
// Step 2
```

```
      Hanoi(disk - 1, aux, dest, source)
// Step 3
```

```
    END IF
END Procedure
STOP
```

BEST FIRST SEARCH

1)Create an empty PriorityQueue PriorityQueue pq;

2) Insert "start" in pq. pq.insert(start)

3) Until PriorityQueue is empty u = PriorityQueue.DeleteMin

If u is the goal

Exit

Else

Foreach neighbor v of u If v "Unvisited"

Mark v "Visited"

pq.insert(v)

Mark u "Examined"

End procedure

Maze Using best first search

1. Take an input of the maze in binary format.
2. Taking the starting point, find all adjacent paths that can be taken.
3. Keep traversing through the array while taking the adjacent cells closest to the destination while avoiding cells with value 0.
4. If the final point is reached, save the length of the path.
5. Compare lengths of all paths that reach the destination and print the length of the shortest path.

A* Search:

1. Initialize the open list
2. Initialize the closed list, put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
4. Find the node with the least f on the open list, call it ‘q’
5. Pop q off the open list
6. generate q's 8 successors and set their parents to q
7. for each successor
8. if successor is the goal, stop search successor.g = q.g + distance between successor and q successor.h = distance from goal to successor successor.f = successor.g + successor.h
9. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
10. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list
11. Push q on the closed list

Fuzzy Logic

Define Non Fuzzy Inputs with Fuzzy Sets. The non-fuzzy inputs are numbers from a certain range, and find how to represent those non-fuzzy values with fuzzy sets.
2. Locate the input, output, and state variables of the plane under consideration.
3. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
4. Obtain the membership function for each fuzzy subset.
5. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and output of fuzzy subsets on the other side, thereby forming the rule base.

6. Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0,1] and [-1, I] intervals.
7. Carry out the fuzzification process.
8. Identify the output contributed from each rule using fuzzy approximate reasoning.
9. Combine the fuzzy outputs obtained from each rule.
10. Finally, apply defuzzification to form a crisp output.

Algorithm of Unification:

1. If Ψ1 or Ψ2 is a variable or constant, then:
a) If Ψ1 or Ψ2 are identical, then return NIL.
b) Else if Ψ1is a variable,
a. then if Ψ1 occurs in Ψ2, then return FAILURE
b. Else return { (Ψ2/ Ψ1)}.
c) Else if Ψ2 is a variable,
a. If Ψ2 occurs in Ψ1 then return FAILURE,
b. Else return {(Ψ1/ Ψ2)}.
d) Else return FAILURE.
2. If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.
3. IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.
4. Set Substitution set(SUBST) to NIL.
5. For i=1 to the number of elements in Ψ1.
a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put
the result into S.
b) If S = failure then returns Failure
c) If S ≠ NIL then do,
a. Apply S to the remainder of both L1 and L2.
b. SUBST= APPEND(S, SUBST).
6. Return SUBST.

Algorithm of Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

successor and q successor.h = distance from goal to successor

successor.f = successor.g + successor.h

UNIFICATION

Step. 1: If Ψ1 or Ψ2 is a variable or constant, then:

a) If Ψ1 or Ψ2 are identical, then return NIL.

b) Else if Ψ1is a variable:

a. then if Ψ1 occurs in Ψ2, then return FAILURE

b. Else return { (Ψ2/ Ψ1)}.

c) Else if Ψ2 is a variable:

a. If Ψ2 occurs in Ψ1 then return FAILURE,

b. Else return {(Ψ1/ Ψ2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then

return FAILURE.

Step. 3: IF Ψ1 and Ψ2 have a different number of arguments, then return

FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ1.

a) Call Unify function with the ith element of Ψ1 and ith element of

Ψ2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S,SUBST).

Step.6: Return SUBST.

Cryptarthematic or constraint satisfaction

Start by examining the rightmost digit of the topmost row, with a carry of 0

If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise

If we are currently trying to assign a char in one of the addends

If char already assigned, just recur on the row beneath this one, adding value into the sum

If not assigned, then

for (every possible choice among the digits not in use)

make that choice and then on row beneath this one, if successful, return true

if !successful, unmake assignment and try another digit

return false if no assignment worked to trigger backtracking

Else if trying to assign a char in the sum

If char assigned & matches correct,

recur on next column to the left with carry, if success return true,

If char assigned & doesn't match, return false

If char unassigned & correct digit already used, return false

If char unassigned & correct digit unused,

assign it and recur on next column to left with carry, if success return true

return false to trigger backtracking

RESOLUTION

Convert the given axiom into CNF, i.e., a conjunction of clauses. Each clause should be disjunction of literals.

Apply negation on the goal given.

Use literals which are required and prove it.

Unlike propositional logic, FOPL literals are complementary if one unifies with the negation of other literal.

tic tac toe

If the game is over, return the score from X's perspective.

Otherwise get a list of new game states for every possible move

Create a scores list

For each of these states add the minimax result of that state to the scores list

If it's X's turn, return the maximum score from the scores list

If it's O's turn, return the minimum score from the scores list