**EXPERIMENT-9:**

**Title: BACKTRACKING**

1. **Find a subset of a given set S = {s l, s2,.....,s n} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions{1,2,6}and{1,8}.A suitable message is to be displayed if the given problem instance doesn't have a solution.**

**Code:**

// A Dynamic Programming solution for subset sum problem

#include <stdio.h>

// Returns true if there is a subset of set[] with sun equal to given sum

bool isSubsetSum(int set[], int n, int sum)

{

// The value of subset[i][j] will be true if there is a

// subset of set[0..j-1] with sum equal to i

bool subset[n+1][sum+1];

// If sum is 0, then answer is true

for (int i = 0; i <= n; i++)

subset[i][0] = true;

// If sum is not 0 and set is empty, then answer is false

for (int i = 1; i <= sum; i++)

subset[0][i] = false;

// Fill the subset table in botton up manner

for (int i = 1; i <= n; i++)

{

for (int j = 1; j <= sum; j++)

{

if(j<set[i-1])

subset[i][j] = subset[i-1][j];

if (j >= set[i-1])

subset[i][j] = subset[i-1][j] || subset[i - 1][j-set[i-1]];

}

}

// uncomment this code to print table

for (int i = 0; i <= n; i++)

{

for (int j = 0; j <= sum; j++)

printf ("%4d\t", subset[i][j]);

printf("\n");

}

return subset[n][sum];

}

int main()

{

int set[] = {3, 34, 4, 12, 5, 2};

int sum = 9;

int n = sizeof(set)/sizeof(set[0]);

if (isSubsetSum(set, n, sum) == true)

printf("Found a subset with given sum");

else

printf("No subset with given sum");

return 0;

}

1. **Implement N Queen's problem using Back Tracking.**

**Code:**

#include<stdio.h>

#include<math.h>

int board[20],count; //global variables

int main()

{

int n,i,j;

void queen(int row,int n);

printf(" - N Queens Problem Using Backtracking -");

printf("\n\nEnter number of Queens: ");

scanf("%d",&n);

queen(1,n);

return 0;

}

//function for printing the solution

void print(int n)

{

int i,j;

printf("\n\nSolution %d:\n\n",++count);

for(i=1;i<=n;++i)

printf("\t%d",i);

for(i=1;i<=n;++i)

{

printf("\n\n%d",i);

for(j=1;j<=n;++j) //for nxn board

{

if(board[i]==j)

printf("\tQ"); //queen at i,j position

else

printf("\t-"); //empty slot

}

}

}

/\*funtion to check conflicts

If no conflict for desired postion returns 1 otherwise returns 0\*/

int place(int row,int column)

{

int i;

for(i=1;i<=row-1;++i)

{

//checking column and digonal conflicts

if(board[i]==column)

return 0;

else

if(abs(board[i]-column)==abs(i-row))

return 0;

}

return 1; //no conflicts

}

//function to check for proper positioning of queen

void queen(int row,int n)

{

int column;

for(column=1;column<=n;++column)

{

if(place(row,column))

{

board[row]=column; //no conflicts so place queen

if(row==n) //dead end

print(n); //printing the board configuration

else //try queen with next position

queen(row+1,n);

}

}

}

1. **Implementation of GRAPH COLORING**

**Code:**

#include <iostream>

#include <list>

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

public:

// Constructor and destructor

Graph(int V) { this->V = V; adj = new list<int>[V]; }

~Graph() { delete [] adj; }

// function to add an edge to graph

void addEdge(int v, int w);

// Prints greedy coloring of the vertices

void greedyColoring();

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

// Assigns colors (starting from 0) to all vertices and prints

// the assignment of colors

void Graph::greedyColoring()

{

int result[V];

// Assign the first color to first vertex

result[0] = 0;

// Initialize remaining V-1 vertices as unassigned

for (int u = 1; u < V; u++)

result[u] = -1; // no color is assigned to u

// A temporary array to store the available colors. True

// value of available[cr] would mean that the color cr is

// assigned to one of its adjacent vertices

bool available[V];

for (int cr = 0; cr < V; cr++)

available[cr] = false;

// Assign colors to remaining V-1 vertices

for (int u = 1; u < V; u++)

{

// Process all adjacent vertices and flag their colors

// as unavailable

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (result[\*i] != -1)

available[result[\*i]] = true;

// Find the first available color

int cr;

for (cr = 0; cr < V; cr++)

if (available[cr] == false)

break;

result[u] = cr; // Assign the found color

// Reset the values back to false for the next iteration

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (result[\*i] != -1)

available[result[\*i]] = false;

}

// print the result

for (int u = 0; u < V; u++)

cout << "Vertex " << u << " ---> Color "

<< result[u] << endl;

}

// Driver program to test above function

int main()

{

Graph g1(5);

g1.addEdge(0, 1);

g1.addEdge(0, 2);

g1.addEdge(1, 2);

g1.addEdge(1, 3);

g1.addEdge(2, 3);

g1.addEdge(3, 4);

cout << "Coloring of graph 1 \n";

g1.greedyColoring();

Graph g2(5);

g2.addEdge(0, 1);

g2.addEdge(0, 2);

g2.addEdge(1, 2);

g2.addEdge(1, 4);

g2.addEdge(2, 4);

g2.addEdge(4, 3);

cout << "\nColoring of graph 2 \n";

g2.greedyColoring();

return 0;

}