Experiment 7

Aim: Implement 0/1 Knapsack problem using Dynamic Programming

Theory: The Knapsack problem is an example of the combinational optimization problem. This problem is also commonly known as the "Rucksack Problem". The name of the problem is defined from the maximization problem. The 0/1 Knapsack problem can be defined as follows: We are given **N** items where each item has some weight ($\mathbf{w_i}$) and value ($\mathbf{v_i}$) associated with it. We are also given a bag with capacity **W**. The target is to put the items into the bag such that the sum of values associated with them is the maximum possible.

Note that here we can either put an item completely into the bag or cannot put it at all.

Mathematically the problem can be represented as:

Maximize $\sum i = 1/N \; vi \; xi$ subject to $\sum i = 1/N \; wi \; xi \leq W$ and $x_i \in \{0, 1\}$

Example:

Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}
Output: 3
Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.

Code:

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
        // Base Case
        if (n == 0 || W == 0)
        return 0;

        // If weight of the nth item is more than
        // Knapsack capacity W, then this item cannot
        // be included in the optimal solution
        if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
```

```
        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        else
        return max(
        val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
        knapSack(W, wt, val, n - 1));
}
int main()
{
        int profit[] = { 60, 100, 120 };
        int weight[] = { 10, 20, 30 };
        int W = 50;
        int n = sizeof(profit) / sizeof(profit[0]);
        printf("%d", knapSack(W, weight, profit, n));
        return 0;
}
```

Output

```
/tmp/qJ2kZfbSrq.o
220

=== Code Execution Successful ===
```

Experiment 8

Aim: Write a program to find single source shortest path in a given graph

Theory: The single-source shortest path (SSSP) problem is a graph problem that involves finding the shortest path from a single vertex to all other vertices in a graph:
The goal is to minimize the sum of the weights of the edges in the paths.
Here are some algorithms that can solve the SSSP problem:

- Breadth-First-Search (BFS): Used for unweighted graphs
- Dijkstra: Can be used to solve the SSSP problem

A correct SSSP algorithm must either return the shortest paths between all vertices or report that there is a negative cycle in the graph. A negative cycle allows for ever-smaller shortest paths because you can run around the cycle to reduce the total weight of a path.

There are two main types of shortest path algorithms: single-source and all-pairs. All-pairs algorithms take longer to run because of the added complexity.

Code:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

#define V 9
int minDistance(int dist[], bool sptSet[])
{
        // Initialize min value
        int min = INT_MAX, min_index;

        for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
        min = dist[v], min_index = v;

        return min_index;
}
```

```
void printSolution(int dist[])
{
	printf("Vertex \t\t Distance from Source\n");
	for (int i = 0; i < V; i++)
	printf("%d \t\t\t\t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src)
{
	int dist[V];
	bool sptSet[V]; // sptSet[i] will be true if vertex i is
	for (int i = 0; i < V; i++)
	dist[i] = INT_MAX, sptSet[i] = false;

	// Distance of source vertex from itself is always 0
	dist[src] = 0;
	for (int count = 0; count < V - 1; count++) {
	// Pick the minimum distance vertex from the set of
	// vertices not yet processed. u is always equal to
	// src in the first iteration.
	int u = minDistance(dist, sptSet);

	// Mark the picked vertex as processed
	sptSet[u] = true;
	for (int v = 0; v < V; v++)
	if (!sptSet[v] && graph[u][v]
		&& dist[u] != INT_MAX
		&& dist[u] + graph[u][v] < dist[v])
		dist[v] = dist[u] + graph[u][v];
	}

	// print the constructed distance array
	printSolution(dist);
}
int main()
{
	int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
		{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },
		{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },
		{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },
```

```
        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```

Output

```
/tmp/u18bFdBAq3.o
Vertex       Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14


=== Code Execution Successful ===
```

Experiment 9

Aim: Program for finding shortest path in a multistage graph using dynamic programming

Theory: A **Multistage graph** is a directed, weighted graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage). The vertices of a multistage graph are divided into n number of disjoint subsets S = { S_1 , S_2 , S_3 ..... S_n }, where S_1 is the source and S_n is the sink (destination). The cardinality of S_1 and S_n are equal to 1. i.e., $|S_1| = |S_n| = 1$.
We are given a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
// Function to find the minimum cost path in the multistage graph
typedef struct {
  int *path;
  int length;
} Result;
Result multistage_graph(int graph[][2], int num_edges, int num_vertices, int stages[][3], int num_stages) {
  // Initialize the lists to store the minimum costs and the next vertex in the path for each vertex
  int *min_costs = (int *)malloc(num_vertices * sizeof(int));
  int *next_vertex = (int *)malloc(num_vertices * sizeof(int));
  for (int i = 0; i < num_vertices; i++) {
    min_costs[i] = INT_MAX;
    next_vertex[i] = -1;
  }
  // Initialize the minimum cost for the sink vertex to 0
  min_costs[num_vertices - 1] = 0;
  // Traverse the graph in reverse order starting from the second-last stage
  for (int i = num_stages - 2; i >= 0; i--) {
    for (int j = 0; j < num_vertices; j++) {
      if (stages[i][0] == j || stages[i][1] == j || stages[i][2] == j) {
```

```
            for (int k = 0; k < num_edges; k++) {
              if (graph[k][0] == j) {
                int neighbor = graph[k][1];
                int cost = graph[k][2] + min_costs[neighbor];
                if (cost < min_costs[j]) {
                  // Update the minimum cost and next vertex for the current vertex
                  min_costs[j] = cost;
                  next_vertex[j] = neighbor;
                }
              }
            }
          }
        }
      }
      // Reconstruct the minimum cost path from source to sink
      int *path = (int *)malloc(num_vertices * sizeof(int));
      int current_vertex = 0; // Start from the source vertex
      int path_length = 0;
      while (current_vertex != -1) {
         path[path_length++] = current_vertex;
         current_vertex = next_vertex[current_vertex];
      }
      // Free the dynamically allocated memory for min_costs and next_vertex
      free(min_costs);
      free(next_vertex);
      // Store the result in a Result structure
      Result result;
      result.path = path;
      result.length = path_length;
      return result;
    }
    int main() {
      // Define the multistage graph represented as an adjacency list
      int graph[][2] = {
        {0, 1}, {0, 2},
        {1, 3}, {1, 4},
        {2, 3}, {2, 4},
        {3, 5},
        {4, 5},
        {5, 6},
```

```
    {6, 7},
    {7, 8}
  };
  int num_edges = sizeof(graph) / sizeof(graph[0]);
  // Define the stages of the multistage graph
  int stages[][3] = {
    {8},       // Sink stage
    {6, 7},    // Stage K-1
    {3, 4, 5}, // Stage K-2
    {1, 2}     // Source stage
  };
  int num_stages = sizeof(stages) / sizeof(stages[0]);
  int num_vertices = 9; // Total number of vertices in the graph
  // Find the minimum cost path and cost using the multistage_graph function
  Result result = multistage_graph(graph, num_edges, num_vertices, stages, num_stages);
  // Print the result
  printf("Minimum cost path: ");
  for (int i = 0; i < result.length; i++) {
    printf("%d ", result.path[i]);
  }
  printf("\nMinimum cost: %d\n", result.path[result.length - 1]);
  // Free the dynamically allocated memory for the path
  free(result.path);
  return 0;
}
```

Output

```
/tmp/g0G2ligFRB.o
Minimum cost path: 0 2
Minimum cost: 2


=== Code Execution Successful ===
```

Experiment 10

Aim: Program to implement 8-queen problem using backtracking

Theory: The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard.

Code:

```
#include<stdio.h>
#include<stdlib.h>
int t[8] = {-1};
int sol = 1;
void printsol()
{
	int i,j;
	char crossboard[8][8];
	for(i=0;i<8;i++)
	{
	for(j=0;j<8;j++)
	{
	crossboard[i][j]='_';
	}
	}
	for(i=0;i<8;i++)
	{
	crossboard[i][t[i]]='q';
	}
	for(i=0;i<8;i++)
	{
	for(j=0;j<8;j++)
	{
	printf("%c ",crossboard[i][j]);
	}
	printf("\n");
	}
}
int empty(int i)
```

```
{
        int j=0;
        while((t[i]!=t[j])&&(abs(t[i]-t[j])!=(i-j))&&j<8)j++;
        return i==j?1:0;
}
void queens(int i)
{
        for(t[i] = 0;t[i]<8;t[i]++)
        {
        if(empty(i))
        {
        if(i==7){
                printsol();
                printf("\n solution %d\n",sol++);
        }
        else
        queens(i+1);
        }
        }
}
int main()
{
        queens(0);
        return 0;
}
```

Output

```
/tmp/pwT6fKTeW5.o
q _ _ _ _ _ _ _
_ _ _ _ q _ _ _
_ _ _ _ _ _ _ q
_ _ _ _ _ q _ _
_ _ q _ _ _ _ _
_ _ _ _ _ _ q _
_ q _ _ _ _ _ _
_ _ _ q _ _ _ _

 solution 1
q _ _ _ _ _ _ _
_ _ _ _ _ q _ _
_ _ _ _ _ _ _ q
_ _ q _ _ _ _ _
_ _ _ _ _ _ q _
_ _ _ q _ _ _ _
_ q _ _ _ _ _ _
_ _ _ _ q _ _ _

 solution 2
q _ _ _ _ _ _ _
_ _ _ _ _ _ q _
_ _ _ q _ _ _ _
_ _ _ _ _ q _ _
_ _ _ _ _ _ _ q
_ q _ _ _ _ _ _
_ _ _ _ q _ _ _
_ _ q _ _ _ _ _

 solution 3
q _ _ _ _ _ _ _
```