EXPERIMENT 7

20CP209P – Design and Analysis of Algorithm Lab

Aim:

Implement the Floyd Warshall Algorithm for All Pair Shortest Path Problem. You are given a weighted diagraph $G = (V, E)$, with arbitrary edge weights or costs c between any node vw v and node w. Find the cheapest path from every node to every other node. Edges may have negative edge weights.

Code:

Floyd Warshall Algorithm:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>

#define fr(i, a, b) for (int i = a; i < b; i++)

#define inf INT_MAX

// note that floyd warhsall does not work for negative cycles

int **create_mat(int n);
void print_mat(int **mat, int n);
int **init_matr();
void copy_matrix(int **src, int **dest, int n);
int ***floydwar_with_hist(int **graph, int vertex);
void show_hist(int ***hist, int vertex);

void reconstruct_path_recursive(int i, int j, int k, int ***hist, char *path_str);
void print_all_shortest_paths(int ***hist, int vertex);
void free_mat(int **mat, int n);
void free_hist(int ***hist, int vertex);

int main(void)
{
   int **matr = init_matr();
   int vertex = 4;
   // int vertex = 9;
   // int vertex = sizeof(matr);
   // the above will always return 4, as it is the size of the pointer
   // int vertex = sizeof(matr) / sizeof(matr[0]);
   // int x = sizeof(matr); printf("x===> %d\n", x);
   // int y = sizeof(matr[0]); printf("y===> %d\n", x);
   // printf("vertex ===> %d \n", vertex);
```

```c
    // int **new_mat = create_mat(vertex);
    // print_mat(new_mat, vertex);

    printf("The Graph: \n");
    print_mat(matr, vertex);

    int ***hist = floydwar_with_hist(matr, vertex);
    printf("All the matrices (history using floyd warshall algo): \n");
    show_hist(hist, vertex);

    printf("All the matrices (history using floyd warshall algo): \n");
    show_hist(hist, vertex);

    printf("Shortest Paths:\n");
    print_all_shortest_paths(hist, vertex);

    return 0;
}

int **create_mat(int n)
{
    int **mat = (int **)calloc(n, sizeof(int*));
    if (mat == NULL)
    {
        printf("Memory Alloc failed!\n");
        return NULL;
    }
    fr(i, 0, n)
    {
        mat[i] = (int*)calloc(n, sizeof(int));
        if (mat[i] == NULL)
        {
            printf("Memory Alloc failed!\n");
            return NULL;
        }
    }

    return mat;
}

void print_mat(int **mat, int n)
{
    fr(i, 0, n)
    {
        fr(j, 0, n)
        {
            if (mat[i][j] == inf)
            {
                printf("inf ");
```

```
        }
        else
            printf("%d ", mat[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}

int **init_matr()
{
  // int matr[9][9] = {
  //    {0,1,1,1,0,0,0,0,0},
  //    {1,0,1,0,1,0,0,0,0},
  //    {1,1,0,1,1,1,1,0,0},
  //    {1,0,1,0,0,0,1,0,0},
  //    {0,1,1,0,0,0,0,0,1},
  //    {0,0,1,0,0,0,1,1,1},
  //    {0,0,1,1,0,1,0,1,0},
  //    {0,0,0,0,0,1,1,0,1},
  //    {0,0,0,0,1,1,0,1,0}
  // };
  int matr[4][4] = {
    {0, 2, inf, 5},
    {3, 0, inf, 4},
    {inf, 1, 0, inf},
    {inf, inf, 2, 0}
  };
  // the above is the graph for the problem
  int n = sizeof(matr) / sizeof(matr[0]);
  printf("n ===> %d\n", n);
  int **new_mat = create_mat(n);
  if (!new_mat)
  {
    printf("Memory Alloc failed\n");
    return NULL;
  }

  fr(i, 0, n)
  {
    fr(j, 0, n)
    {
      new_mat[i][j] = matr[i][j];
    }
  }

  return new_mat;
}

void copy_matrix(int **src, int **dest, int n)
```

```
{
    fr(i, 0, n)
    {
        fr(j, 0, n)
        {
            dest[i][j] = src[i][j];
        }
    }
}

int ***floydwar_with_hist(int **graph, int vertex)
{
    // here graph is matr or matr is graph
    int ***hist = (int ***)malloc(vertex + 1 * sizeof(int**));

    hist[0] = create_mat(vertex);
    copy_matrix(graph, hist[0], vertex);

    fr(k, 0, vertex)
    {
        hist[k + 1] = create_mat(vertex);
        copy_matrix(hist[k], hist[k + 1], vertex);
        fr(i, 0, vertex)
        {
            fr(j, 0, vertex)
            {
                if (hist[k][i][k] != inf && hist[k][k][j] != inf)
                {
                    int new_dist = hist[k][i][k] + hist[k][k][j];
                    if (new_dist < hist[k + 1][i][j])
                    {
                        hist[k + 1][i][j] = new_dist;
                    }
                }
            }
        }
    }

    return hist;
}

void show_hist(int ***hist, int vertex)
{
    fr(i, 0, vertex + 1)
    {
        printf("hist[%d]:\n", i);
        print_mat(hist[i], vertex);
    }
}
```

```
void reconstruct_path_recursive(int i, int j, int k, int ***hist, char *path_str)
{
  // Base case: If k < 0, it means no intermediate node from 0 to vertex-1
  // was found on the path between the *current* i and j segment.
  // This implies a direct edge (or i==j, handled outside).
  if (k < 0)
  {
    // We only need to add the intermediate nodes. The start and end are handled outside.
    // If i and j were directly connected in the original graph check:
    // if (hist[0][i][j] != inf && hist[0][i][j] != 0 ) { } // No action needed here
    return;
  }

  // Check if the shortest path from i to j *changed* when node k was introduced.
  // We compare the distance in hist[k+1] (using nodes up to k)
  // with the distance in hist[k] (using nodes up to k-1).
  if (hist[k + 1][i][j] < hist[k][i][j])
  {
    // Yes, node 'k' is essential for the shortest path between i and j.
    // The path must go i -> ... -> k -> ... -> j.
    // Recursively find the path from i to k (using intermediates up to k-1).
    reconstruct_path_recursive(i, k, k - 1, hist, path_str);

    // Append the intermediate node k to the path string.
    char buffer[20];
    sprintf(buffer, " -> %d", k);
    strcat(path_str, buffer);

    // Recursively find the path from k to j (using intermediates up to k-1).
    reconstruct_path_recursive(k, j, k - 1, hist, path_str);
  }
  else
  {
    // No, the shortest path from i to j did NOT require node 'k' at this stage.
    // The path is the same as the one found using intermediate nodes up to k-1.
    // Continue checking with the next lower intermediate node.
    reconstruct_path_recursive(i, j, k - 1, hist, path_str);
  }
}

void print_all_shortest_paths(int ***hist, int vertex)
{
  fr(i, 0, vertex)
  {
    fr(j, 0, vertex)
    {
      printf("Path from %d to %d: ", i, j);

      // Check if a path exists
      if (hist[vertex][i][j] == inf)
```

```
        {
            printf("No path\n");
        }
        else if (i == j)
        {
            printf("%d (Dist: 0)\n", i);
        }
        else
        {
            // Allocate a buffer for the path string. Size calculation can be tricky,
            // make it large enough (e.g., vertex * (max digits + arrow len)).
            char path_str[vertex * 15]; // Adjust size if needed
            sprintf(path_str, "%d", i); // Start path string with the source node 'i'

            // Call the recursive function to build the intermediate path nodes string.
            // Start checking from the highest possible intermediate node (vertex - 1).
            reconstruct_path_recursive(i, j, vertex - 1, hist, path_str);

            // Append the final destination node 'j'.
            char buffer[20];
            sprintf(buffer, " -> %d", j);
            strcat(path_str, buffer);

            // Print the reconstructed path and the final distance.
            printf("%s (Dist: %d)\n", path_str, hist[vertex][i][j]);
        }
    }
    printf("\n"); // Add a newline after processing all paths from node i
  }
}

void free_mat(int **mat, int n)
{
  if (!mat) return;
  fr(i, 0, n)
  {
    free(mat[i]); // Free each row's columns
  }
  free(mat); // Free the row pointers
}

void free_hist(int ***hist, int vertex)
{
  if (!hist) return;
  // Free matrices from hist[0] to hist[vertex]
  fr(k, 0, vertex + 1)
  {
    free_mat(hist[k], vertex);
  }
  free(hist); // Free the array of matrix pointers
```

}

Analysis:

→ Floyd Warshall Algorithm

```
1 for (int K=0; K<N; K++)
2     for(int i=0; i<N, i++)
3         for (j=0, j<N, K++)
4             if (D[i][k] != inf && D[k][j] != inf
5                 && D[i][j] > D[i][k]
                  D[i][j] = D[i][k]+D[k][j] (+ D[k][j])
6                 next[i][j] = next[i][k]
```

	Count	Worst Case
1	N	$O(N)$
2	$N \times K$	$O(N^2)$
3	$N \times K \times i$	$O(N3)$
4	N^3	$O(1)$
5	S	$N^3 \times$ update
6	C	$N^3 \times$ update

Time Complexity = $O(N^3)$

Output:

```
All the matrices (history using floyd warshall algo):
hist[0]:
18422184 18421992 18422280 18422160
3 0 inf 4
inf 1 0 inf
inf inf 2 0

hist[1]:
0 2 inf 5
3 0 inf 4
inf 1 0 inf
inf inf 2 0

hist[2]:
0 2 inf 5
3 0 inf 4
4 1 0 5
inf inf 2 0

hist[3]:
0 2 inf 5
3 0 inf 4
4 1 0 5
6 3 2 0

hist[4]:
0 2 7 5
3 0 6 4
4 1 0 5
6 3 2 0

Shortest Paths:
Path from 0 to 0: 0 (Dist: 0)
Path from 0 to 1: 0 -> 0 -> 1 (Dist: 2)
Path from 0 to 2: 0 -> 0 -> 3 -> 2 (Dist: 7)
Path from 0 to 3: 0 -> 0 -> 3 (Dist: 5)

Path from 1 to 0: 1 -> 0 (Dist: 3)
Path from 1 to 1: 1 (Dist: 0)
Path from 1 to 2: 1 -> 3 -> 2 (Dist: 6)
Path from 1 to 3: 1 -> 3 (Dist: 4)

Path from 2 to 0: 2 -> 1 -> 0 (Dist: 4)
Path from 2 to 1: 2 -> 1 (Dist: 1)
Path from 2 to 2: 2 (Dist: 0)
Path from 2 to 3: 2 -> 1 -> 3 (Dist: 5)

Path from 3 to 0: 3 -> 2 -> 1 -> 0 (Dist: 6)
Path from 3 to 1: 3 -> 2 -> 1 (Dist: 3)
Path from 3 to 2: 3 -> 2 (Dist: 2)
Path from 3 to 3: 3 (Dist: 0)
```