DIGITAL ANALYSIS AND ALGORITHM

EXPERIMENT – 06

NAME :- MANTHAN AYALWAR

UID :- 2021700003

BATCH :- D1

Aim : Greedy-Approach single source shortest path- Dijkstra's Algorithm

Algorithm :

Djikstra's algorithm:

ALGORITHM 1 Dijkstra's Algorithm.

procedure *Dijkstra*(G: weighted connected simple graph, with
  all weights positive)
{G has vertices $a = v_0, v_1, \ldots, v_n = z$ and lengths $w(v_i, v_j)$
  where $w(v_i, v_j) = \infty$ if $\{v_i, v_j\}$ is not an edge in G}
for $i := 1$ **to** n
  $L(v_i) := \infty$
$L(a) := 0$
$S := \emptyset$
{the labels are now initialized so that the label of a is 0 and all
  other labels are ∞, and S is the empty set}
while $z \notin S$
  $u :=$ a vertex not in S with $L(u)$ minimal
  $S := S \cup \{u\}$
  for all vertices v not in S
    if $L(u) + w(u, v) < L(v)$ **then** $L(v) := L(u) + w(u, v)$
    {this adds a vertex to S with minimal label and updates the
    labels of vertices not in S}
return $L(z)$ {$L(z)$ = length of a shortest path from a to z}

Prim’s algorithm:

ALGORITHM 1 Prim’s Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for i := 1 **to** $n - 2$
    e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
    T := T with e added
return T {T is a minimum spanning tree of G}

Code :

Utilities.c

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// linked list implementation:


typedef struct edge

{

   int dest;

   int source; // for prims     int weight;

   int shortest_distance;     struct edge *next;

} *p_edge;

typedef struct linked_list

{

   p_edge head;     p_edge tail; } *p_linked_list;


p_edge create_edge(int dest, int weight)

{

   p_edge new = malloc(sizeof(struct edge));     new->next = NULL;     new->dest = dest;
new->weight = weight;     return new;

}
```

```
p_linked_list create_linked_list()

{

    p_linked_list ll = malloc(sizeof(struct linked_list));

    ll->head = NULL;     ll->tail = NULL;

}


void insert_edge(p_linked_list ll, int dest, int weight)

{

    if (ll->tail == NULL)

    {

        ll->head = create_edge(dest, weight);

        ll->tail = ll->head;

    }     else     {

        ll->tail->next = create_edge(dest, weight);

        ll->tail = ll->tail->next;

}

}


// priority queue implementation:


typedef struct priority_queue

{

    int arr_size;     int heap_size;     p_edge *arr;

    int type; // 0 -> min heap based on weights, 1 -> min heap based on shortest distance }
*p_priority_queue;


p_priority_queue create_priority_queue(int max_size, int type)

{

    p_priority_queue ppq = malloc(sizeof(struct priority_queue));     ppq->arr =
malloc(sizeof(p_edge) * max_size);

    ppq->arr_size = max_size;     ppq->heap_size = 0;     ppq->type = type;     return ppq;
```

```
}

void enqueue(p_priority_queue pq, p_edge edge)
{
    if (pq->heap_size >= pq->arr_size)
    {
        printf("Queue is full. Element cannot be inserted.\n");         return;
    }
    // heap insertion:     pq->arr[pq->heap_size] = edge;
    int parent = ((pq->heap_size + 1) / 2) - 1;
    int child = pq->heap_size;     pq->heap_size++;     p_edge temp;     if (pq->type == 1)
    {
        while (parent >= 0)
        {
            if (pq->arr[child]->shortest_distance < pq->arr[parent]-
>shortest_distance)             {
                temp = pq->arr[child];
                pq->arr[child] = pq->arr[parent];                 pq->arr[parent] = temp;                 child
= parent;
                parent = ((parent + 1) / 2) - 1;

           }
            else                 return;
        }     }     else     {
        while (parent >= 0)
        {
            if (pq->arr[child]->weight < pq->arr[parent]->weight)
            {
                temp = pq->arr[child];
                pq->arr[child] = pq->arr[parent];                 pq->arr[parent] = temp;                 child
= parent;
```

```
            parent = ((parent + 1) / 2) - 1;
        }           else                return;
    }
  }
}


p_edge dequeue(p_priority_queue pq)
{
  if (pq->heap_size == 0)        return NULL;
  p_edge ret_val = pq->arr[0];     pq->heap_size--;
  pq->arr[0] = pq->arr[pq->heap_size];
  // heapify:     int parent = 0, smallest, left, right;     p_edge temp;     if (pq->type == 1)
  {
    while (parent < pq->heap_size)
    {
      left = parent * 2 + 1;            right = left + 1;            smallest = parent;
      if (left < pq->heap_size && pq->arr[left]>shortest_distance < pq->arr[smallest]-
>shortest_distance)                smallest = left;
      if (right < pq->heap_size && pq->arr[right]>shortest_distance < pq->arr[smallest]-
>shortest_distance)                smallest = right;            if (smallest != parent)
      {
        temp = pq->arr[smallest];
     pq->arr[smallest] = pq->arr[parent];
        pq->arr[parent] = temp;                parent = smallest;
      }           else                break;
    }     }     else     {
    while (parent < pq->heap_size)
    {
      left = parent * 2 + 1;            right = left + 1;            smallest = parent;
      if (left < pq->heap_size && pq->arr[left]->weight < pq-
>arr[smallest]->weight)                smallest = left;
```

```
        if (right < pq->heap_size && pq->arr[right]->weight < pq->arr[smallest]->weight)
smallest = right;          if (smallest != parent)
        {
            temp = pq->arr[smallest];
            pq->arr[smallest] = pq->arr[parent];            pq->arr[parent] = temp;
parent = smallest;
        }         else              break;
      }    }
   return ret_val;
}

int is_empty(p_priority_queue pq)
{
   return pq->heap_size == 0;
}

// graphs: typedef struct graph
{
   int directed;
   int V;              // no. of vertices     int E;              // no. of edges
   p_linked_list *edges; // array of pointers to linked lists of edges...index of the array
represents source vertex and each linked list containes all outgoing edges of that source
vertex
} *p_graph;

p_graph take_directed_graph_input()
{
   p_graph g = malloc(sizeof(struct graph));     g->directed = 1;
   printf("Enter number of vertices: ");
   scanf("%d", &g->V);
   printf("Enter number of edges: ");
   scanf("%d", &g->E);
```

```
    g->edges = malloc(sizeof(p_linked_list) * g->V);
    for (int i = 0; i < g->V; i++)          g->edges[i] = create_linked_list();      int source, dest,
weight;      for (int i = 0; i < g->E; i++)
    {
        printf("For edge %d:\nEnter source vertex: ", i + 1);          scanf("%d", &source);
        printf("Enter destination vertex: ");
        scanf("%d", &dest);          printf("Enter weight: ");          scanf("%d", &weight);
        insert_edge(g->edges[source], dest, weight);
    }
    return g;
}

void add_edge_to_undirected_graph(p_graph g, int source, int dest, int weight)
{
    insert_edge(g->edges[source], dest, weight);      g->edges[source]->tail->source = source;
insert_edge(g->edges[dest], source, weight);      g->edges[dest]->tail->source = dest;
}

p_graph take_undirected_graph_input()
{
    p_graph g = malloc(sizeof(struct graph));      g->directed = 0;
    printf("Enter number of vertices: ");
    scanf("%d", &g->V);
    printf("Enter number of edges: ");
    scanf("%d", &g->E);
    g->edges = malloc(sizeof(p_linked_list) * g->V);
    for (int i = 0; i < g->V; i++)          g->edges[i] = create_linked_list();      int source, dest,
weight;      for (int i = 0; i < g->E; i++)
{
        printf("For edge %d:\nEnter source vertex: ", i + 1);          scanf("%d", &source);
        printf("Enter destination vertex: ");
```

```
        scanf("%d", &dest);         printf("Enter weight: ");         scanf("%d", &weight);
        add_edge_to_undirected_graph(g, source, dest, weight);
    }
    return g;
}

void display_graph(p_graph g)
{
    for (int i = 0; i < g->V; i++)
    {
        printf("%d : ", i);
        if (g->edges[i]->head != NULL)
        {
            printf("(%d,%d) ", g->edges[i]->head->dest, g->edges[i]-
>head->weight);
            for (p_edge e = g->edges[i]->head->next; e != NULL; e = e->next)
                printf(", (%d,%d) ", e->dest, e->weight);
        }
        printf("\n");
    }
}
```

Dijkstra.c

```
 #include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "utilities.h"

struct djikstra_result
{
    int *parent;
```

```
    int *shortest_distance;     int size;     int source;
};

struct djikstra_result *djikstras(p_graph g, int source)
{
    p_priority_queue pq = create_priority_queue(g->V, 1); // min heap of shortest distance
int visited[g->V];
    struct djikstra_result *dr = malloc(sizeof(struct djikstra_result));
    dr->parent = malloc(sizeof(int) * g->V);
    dr->shortest_distance = malloc(sizeof(int) * g->V);
    dr->parent[source] = -1;     dr->size = g->V;     dr->source = source;
    for (int i = 0; i < g->V; i++)
    {
        visited[i] = 0;
        dr->shortest_distance[i] = INT_MAX;
    }
    dr->shortest_distance[source] = 0;
    struct edge self = {.dest = 0, .weight = 0, .shortest_distance =
0, .next = g->edges[source]->head};     enqueue(pq, &self);     while (!is_empty(pq))
    {
        // extracting the edge with minimum weight         int curr_vtx = dequeue(pq)->dest;
        // traversing neighbours of current vertex
        for (p_edge i = g->edges[curr_vtx]->head; i != NULL; i = i-
>next)         {
            if (!visited[i->dest])
            {
                if (dr->shortest_distance[curr_vtx] + i->weight < dr->shortest_distance[i->dest])
                {

                    dr->shortest_distance[i->dest] = dr-
>shortest_distance[curr_vtx] + i->weight;
```

```
                dr->parent[i->dest] = curr_vtx;
            }
            i->shortest_distance = dr->shortest_distance[i-
>dest];
            enqueue(pq, i);
        }       }
        visited[curr_vtx] = 1;
    }
    return dr;
}

void print_path(int *parents, int dest)
{
    if (parents[dest] == -1)
    {
        printf("%d", dest);        return;
    }
    print_path(parents, parents[dest]);     printf(" -> %d", dest);
}

void print_all_shortest_paths(struct djikstra_result *dr)
{
    for (int i = 0; i < dr->size; i++)
    {
        printf("To %d : ", i);        print_path(dr->parent, i);
        printf("  :  %d\n", dr->shortest_distance[i]);
    }
}

int main()
```

```
{
    freopen("input.txt", "r", stdin);
    p_graph g = take_directed_graph_input();
    printf("Enter source vertex: ");
    int source;     scanf("%d", &source);
    struct djikstra_result *dr = djikstras(g, source);
    printf("\n\nAll single source (%d) shortest paths and their costs:\n\n", source);
    print_all_shortest_paths(dr);

}
```

Prims.c

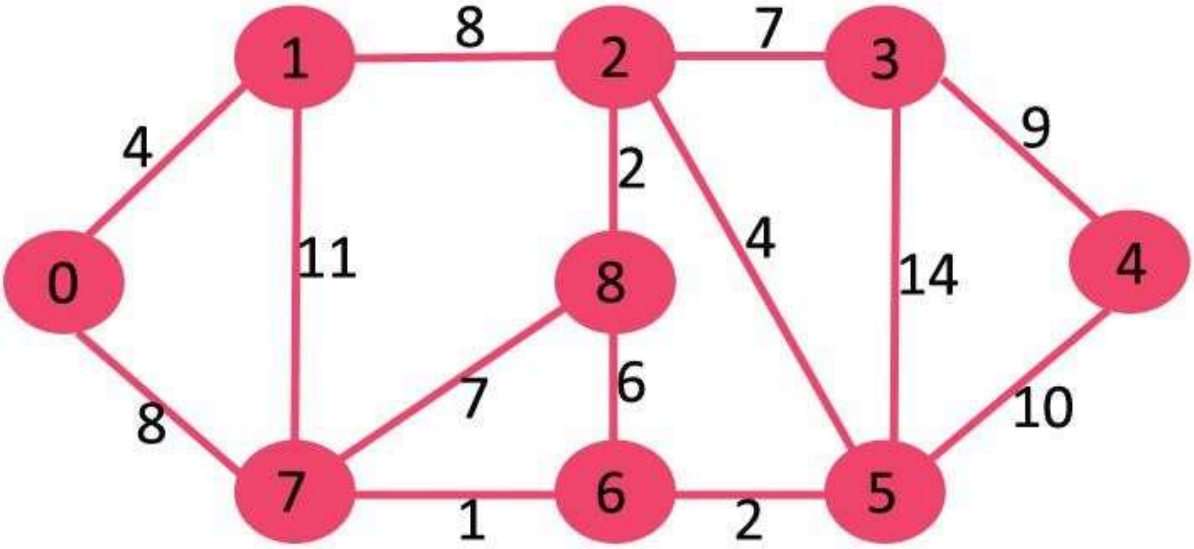
prims.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "utilities.h"

struct prims_result
{
    int cost;     p_graph mst;
};

struct prims_result *prims(p_graph g)
{
    p_graph mst = malloc(sizeof(struct graph));     mst->V = g->V;     mst->E = 0;     mst-
>directed = 0;
    mst->edges = malloc(sizeof(p_linked_list) * g->V);
    for (int i = 0; i < mst->V; i++)         mst->edges[i] = create_linked_list();
    p_priority_queue pq = create_priority_queue(g->E, 0);
    int added[g->V];     int cost = 0;
```

```
    for (int i = 0; i < g->V; i++)         added[i] = 0;     int curr_vtx = 0;     added[curr_vtx] = 1;
int vtx_counter = 1;     do     {

        for (p_edge e = g->edges[curr_vtx]->head; e != NULL; e = e-

>next)         {

            if (!added[e->dest])                 enqueue(pq, e);

        }

        p_edge min_edge = dequeue(pq);

        while (added[min_edge->source] && added[min_edge->dest])             min_edge =
dequeue(pq);

        add_edge_to_undirected_graph(mst, min_edge->source, min_edge->dest, min_edge-
>weight);

        printf("added edge (%d,%d)\n", min_edge->source, min_edge-

>dest);         mst->E++;

        cost += min_edge->weight;         added[min_edge->dest] = 1;         vtx_counter++;

        curr_vtx = min_edge->dest;


    } while (vtx_counter < g->V);

    struct prims_result *pr = malloc(sizeof(struct prims_result));

    pr->cost = cost;     pr->mst = mst;     return pr;

}


int main()

{

    freopen("input_2.txt", "r", stdin);

    p_graph g = take_undirected_graph_input();

    printf("\n\n");

    struct prims_result *pr = prims(g);

    printf("\nCost: %d\n", pr->cost);

    printf("Tree (graph):\n");

    display_graph(pr->mst);
```

}

Input :



Output :

```
Enter number of edges: 14
For edge 1:
Enter source vertex: 0
Enter destination vertex: 1
Enter weight: 4
For edge 2:
Enter source vertex: 0
Enter destination vertex: 7
Enter weight: 8
For edge 3:
Enter source vertex: 1
Enter destination vertex: 7
Enter weight: 11
For edge 4:
Enter source vertex: 1
Enter destination vertex: 2
Enter weight: 8
For edge 5:
Enter source vertex: 7
Enter destination vertex: 6
Enter weight: 1
For edge 6:
Enter source vertex: 7
Enter destination vertex: 8
Enter weight: 7
For edge 7:
Enter source vertex: 2
Enter destination vertex: 8
Enter weight: 2
For edge 8:
Enter source vertex: 8
Enter destination vertex: 6
Enter weight: 6
```

```
For edge 9:
Enter source vertex: 2
Enter destination vertex: 3
Enter weight: 7
For edge 10:
Enter source vertex: 2
Enter destination vertex: 5
Enter weight: 4
For edge 11:
Enter source vertex: 6
Enter destination vertex: 5
Enter weight: 2
For edge 12:
Enter source vertex: 3
Enter destination vertex: 5
Enter weight: 14
For edge 13:
Enter source vertex: 3
Enter destination vertex: 4
Enter weight: 9
For edge 14:
Enter source vertex: 5
Enter destination vertex: 4
Enter weight: 10
```

```
All single source (0) shortest paths and their costs:

To 0 : 0  :  0
To 1 : 0 -> 1  :  4
To 2 : 0 -> 1 -> 2  :  12
To 3 : 0 -> 1 -> 2 -> 3  :  19
To 4 : 0 -> 7 -> 6 -> 5 -> 4  :  21
To 5 : 0 -> 7 -> 6 -> 5  :  11
To 6 : 0 -> 7 -> 6  :  9
To 7 : 0 -> 7  :  8
To 8 : 0 -> 1 -> 2 -> 8  :  14
```

Output: *0 4 12 19 21 11 9 8 14*

Explanation: *The distance from 0 to 1 = 4.*

The minimum distance from 0 to 2 = 12. 0->1->2

The minimum distance from 0 to 3 = 19. 0->1->2->3

The minimum distance from 0 to 4 = 21. 0->7->6->5->4

The minimum distance from 0 to 5 = 11. 0->7->6->5

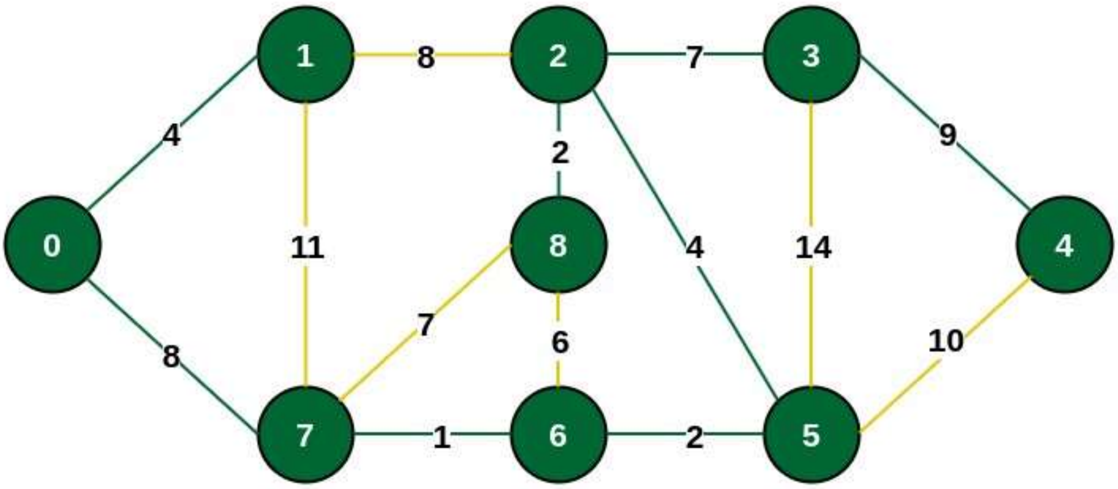
The minimum distance from 0 to 6 = 9. 0->7->6

The minimum distance from 0 to 7 = 8. 0->7

The minimum distance from 0 to 8 = 14. 0->1->2->8

Prim’s algorithm:

Input graph:

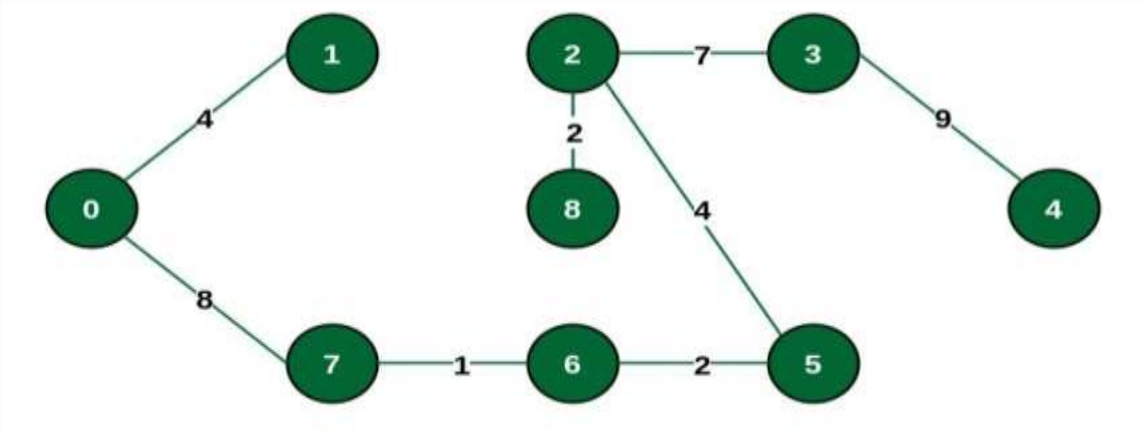



```
Enter number of edges: 14
For edge 1:
Enter source vertex: 0
Enter destination vertex: 1
Enter weight: 4
For edge 2:
Enter source vertex: 0
Enter destination vertex: 7
Enter weight: 8
For edge 3:
Enter source vertex: 1
Enter destination vertex: 7
Enter weight: 11
For edge 4:
Enter source vertex: 1
Enter destination vertex: 2
Enter weight: 8
For edge 5:
Enter source vertex: 7
Enter destination vertex: 6
Enter weight: 1
For edge 6:
Enter source vertex: 7
Enter destination vertex: 8
Enter weight: 7
For edge 7:
Enter source vertex: 2
Enter destination vertex: 8
Enter weight: 2
For edge 8:
Enter source vertex: 8
Enter destination vertex: 6
Enter weight: 6
```

```
For edge 9:
Enter source vertex: 2
Enter destination vertex: 3
Enter weight: 7
For edge 10:
Enter source vertex: 2
Enter destination vertex: 5
Enter weight: 4
For edge 11:
Enter source vertex: 6
Enter destination vertex: 5
Enter weight: 2
For edge 12:
Enter source vertex: 3
Enter destination vertex: 5
Enter weight: 14
For edge 13:
Enter source vertex: 3
Enter destination vertex: 4
Enter weight: 9
For edge 14:
Enter source vertex: 5
Enter destination vertex: 4
Enter weight: 10
```

```
added edge (0,1)
added edge (0,7)
added edge (7,6)
added edge (6,5)
added edge (5,2)
added edge (2,8)
added edge (2,3)
added edge (3,4)

Cost: 37
Tree (graph):
0 : (1,4) , (7,8)
1 : (0,4)
2 : (5,4) , (8,2) , (3,7)
3 : (2,7) , (4,9)
4 : (3,9)
5 : (6,2) , (2,4)
6 : (7,1) , (5,2)
7 : (0,8) , (6,1)
8 : (2,2)
```

The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = ***37***.



Conclusion : Greedy approach can be used on graphs to find shortest path from a vertex to all other vertices as well as to construct a minimum spanning tree of a graph