GRAPH: (START FROM 0)

0 1 10

0 2 5

1 2 2

1 3 1

2 1 3

2 3 9

2 4 2

3 4 4

4 3 6

(CHANGE THE GRAPH ACCORDINGLY!!)

BELLMAN FORD ALGORITHM:

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

typedef struct node
{
  int u;
  int v;
  int w;
} node;

typedef struct graph
{
  int V;
  int E;
  node *edges;
} graph;

graph *createGraph(int V, int E)
{
  graph *g = (graph *)malloc(sizeof(graph));
  g->V = V;
  g->E = E;
  g->edges = (node *)malloc(E * sizeof(node));
  return g;
}
```

```c
void addEdge(graph *g, int i, int u, int v, int w)
{
  g->edges[i].u = u;
  g->edges[i].v = v;
  g->edges[i].w = w;
}

void initializeSingleSource(graph *g, int src, int *dist, int *parent)
{
  int i;
  for (i = 0; i < g->V; i++)
  {
    dist[i] = INT_MAX;
    parent[i] = -1;
  }
  dist[src] = 0;
}

void Relax(int u, int v, int w, int *dist, int *parent)
{
  if (dist[u] != INT_MAX && dist[u] + w < dist[v])
  {
    dist[v] = dist[u] + w;
    parent[v] = u;
  }
}

int BellmanFord(graph *g, int src, int *dist, int *parent)
{
  initializeSingleSource(g, src, dist, parent);
  int i, j;
  for (i = 0; i < g->V - 1; i++)
  {
    for (j = 0; j < g->E; j++)
    {
      Relax(g->edges[j].u, g->edges[j].v, g->edges[j].w, dist, parent);
    }
  }
  for (i = 0; i < g->E; i++)
  {
    if (dist[g->edges[i].u] != INT_MAX && dist[g->edges[i].u] + g-
>edges[i].w < dist[g->edges[i].v])
    {
      return 0;
    }
  }
  return 1;
}
```

```c
void printPath(int src, int dest, int *parent)
{
  if (dest == src)
  {
    printf("%c -> ", 65 + src);
  }
  else if (parent[dest] == -1)
  {
    printf("No path from %c to %c\n", 65 + src, 65 + dest);
  }
  else
  {
    printPath(src, parent[dest], parent);
    printf("%c -> ", 65 + dest);
  }
}

int main()
{
  int V = 5;
  int E = 9;
  graph *g = createGraph(V, E);
  int *d = (int *)malloc(V * sizeof(int));
  int *pi = (int *)malloc(V * sizeof(int));
  FILE *file = fopen("Bellman.txt", "r");
  int u, v, w, i;
  for (i = 0; i < E; i++)
  {
    fscanf(file, "%d %d %d", &u, &v, &w);
    addEdge(g, i, u, v, w);
  }
  int check = BellmanFord(g, 0, d, pi);
  if (check == 1)
  {
    for (i = 1; i < V; i++)
    {
      printf("Distance from A to %c: %d\n", 65 + i, d[i]);
      printPath(0, i, pi);
      printf("NULL\n");
    }
  }
  else
  {
    printf("Graph contains a negative-weight cycle\n");
  }
  return 0;
}
```

DIJKSTRA ALGORITHM:

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

typedef struct node
{
  int v;
  int w;
  struct node *next;
} node;

typedef struct graph
{
  int V;
  node **adj;
} graph;

typedef struct MinHeapNode
{
  int v;
  int d;
} MinHeapNode;

typedef struct MinHeap
{
  int size;
  int capacity;
  int *pos;
  MinHeapNode **array;
} MinHeap;

graph *createGraph(int V)
{
  graph *g = (graph *)malloc(sizeof(graph));
  g->V = V;
  g->adj = (node **)malloc(V * sizeof(node *));
  for (int i = 0; i < V; i++)
  {
    g->adj[i] = NULL;
  }
  return g;
}

node *createNode(int v, int w)
```

```
{
  node *newNode = (node *)malloc(sizeof(node));
  newNode->v = v;
  newNode->w = w;
  newNode->next = NULL;
  return newNode;
}

void addEdge(graph *g, int u, int v, int w)
{
  node *newNode = createNode(v, w);
  newNode->next = g->adj[u];
  g->adj[u] = newNode;
}

MinHeap *createMinHeap(int capacity)
{
  MinHeap *minHeap = (MinHeap *)malloc(sizeof(MinHeap));
  minHeap->pos = (int *)malloc(capacity * sizeof(int));
  minHeap->size = 0;
  minHeap->capacity = capacity;
  minHeap->array = (MinHeapNode **)malloc(capacity * sizeof(MinHeapNode
*));
  return minHeap;
}

void swapMinHeapNode(MinHeapNode **a, MinHeapNode **b)
{
  MinHeapNode *t = *a;
  *a = *b;
  *b = t;
}

void minHeapify(MinHeap *minHeap, int idx)
{
  int smallest, left, right;
  smallest = idx;
  left = 2 * idx + 1;
  right = 2 * idx + 2;
  if (left < minHeap->size && minHeap->array[left]->d < minHeap-
>array[smallest]->d)
  {
    smallest = left;
  }
  if (right < minHeap->size && minHeap->array[right]->d < minHeap-
>array[smallest]->d)
  {
    smallest = right;
```

```
  }
  if (smallest != idx)
  {
    MinHeapNode *smallestNode = minHeap->array[smallest];
    MinHeapNode *idxNode = minHeap->array[idx];
    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;
    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}

int isEmpty(MinHeap *minHeap)
{
  return minHeap->size == 0;
}

MinHeapNode *extractMin(MinHeap *minHeap)
{
  if (isEmpty(minHeap))
  {
    return NULL;
  }
  MinHeapNode *root = minHeap->array[0];
  MinHeapNode *lastNode = minHeap->array[minHeap->size - 1];
  minHeap->array[0] = lastNode;
  minHeap->pos[root->v] = minHeap->size - 1;
  minHeap->pos[lastNode->v] = 0;
  --minHeap->size;
  minHeapify(minHeap, 0);
  return root;
}

void decreaseKey(MinHeap *minHeap, int v, int d)
{
  int i = minHeap->pos[v];
  minHeap->array[i]->d = d;
  while (i && minHeap->array[i]->d < minHeap->array[(i - 1) / 2]->d)
  {
    minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
    minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
    swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);
    i = (i - 1) / 2;
  }
}

int isInMinHeap(MinHeap *minHeap, int v)
{
```

```c
  if (minHeap->pos[v] < minHeap->size)
  {
    return 1;
  }
  return 0;
}

void prinPathRec(int *pi, int s, int v)
{
  if (v == s)
  {
    printf("%c -> ", 65 + s);
    return;
  }
  prinPathRec(pi, s, pi[v]);
  printf("%c -> ", 65 + v);
}

void printArr(int *d, int V, int *pi)
{
  int i;
  for (i = 1; i < V; i++)
  {
    printf("Distance from A to %c is %d\n", 65 + i, d[i]);
    printf("Path: ");
    prinPathRec(pi, 0, i);
    printf("NULL\n");
  }
}

void InitializeSingleSource(int *d, int *pi, int s, int V)
{
  for (int i = 0; i < V; i++)
  {
    d[i] = INT_MAX;
    pi[i] = -1;
  }
  d[s] = 0;
}

void Relax(int *d, int *pi, int u, int v, int w)
{
  if (d[v] > d[u] + w)
  {
    d[v] = d[u] + w;
    pi[v] = u;
  }
}
```

```
void Dijkstra(graph *g, int s, int *d, int *pi)
{
  MinHeap *minHeap = createMinHeap(g->V);
  int v;
  for (v = 0; v < g->V; v++)
  {
    d[v] = INT_MAX;
    pi[v] = -1;
    minHeap->array[v] = (MinHeapNode *)malloc(sizeof(MinHeapNode));
    minHeap->array[v]->v = v;
    minHeap->array[v]->d = d[v];
    minHeap->pos[v] = v;
  }
  minHeap->array[s]->d = d[s];
  minHeap->pos[s] = s;
  d[s] = 0;
  decreaseKey(minHeap, s, d[s]);
  minHeap->size = g->V;
  while (!isEmpty(minHeap))
  {
    MinHeapNode *minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v;
    node *temp = g->adj[u];
    while (temp != NULL)
    {
      int v = temp->v;
      if (isInMinHeap(minHeap, v) && d[u] != INT_MAX && temp->w + d[u] <
d[v])
      {
        d[v] = d[u] + temp->w;
        pi[v] = u;
        decreaseKey(minHeap, v, d[v]);
      }
      temp = temp->next;
    }
  }
}

int main()
{
  int V = 5;
  graph *g = createGraph(V);
  FILE *file = fopen("Dijkstra.txt", "r");
  if (file == NULL)
  {
    printf("File not found\n");
    return 0;
```

```
  }
  int u, v, w;
  while (fscanf(file, "%d %d %d", &u, &v, &w) != EOF)
  {
    addEdge(g, u, v, w);
  }
  fclose(file);
  int *d = (int *)malloc(V * sizeof(int));
  int *pi = (int *)malloc(V * sizeof(int));
  InitializeSingleSource(d, pi, 0, V);
  Dijkstra(g, 0, d, pi);
  printArr(d, V, pi);

  return 0;
}
```