Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

Link : Click IT

Experiment 5

Aim: Implement DIJKSTRA'S ALGORITHM using OpenMP

Theory :

Dijkstra's algorithm is a popular shortest path algorithm used to find the shortest path between a source vertex and all other vertices in a weighted graph. OpenMP is a parallel programming model used to achieve parallelism in C/C++ programs.

The following are the steps involved in implementing Dijkstra's algorithm using OpenMP:

Initialize the distance array and visited array: In this step, we initialize the distance array to infinity and the visited array to zero for all vertices in the graph. This is done using an OpenMP parallel loop.

Set the distance to the starting node as 0: We set the distance to the starting node as 0.

Find the shortest path for all vertices: In this step, we find the shortest path for all vertices in the graph. We use an OpenMP parallel loop to iterate over all vertices in the graph. For each vertex, we find the vertex with the minimum distance value and mark it as visited. Then, we update the distance values of its adjacent vertices. This step is repeated until all vertices are visited.

Print the distances of all vertices from the starting vertex: In this step, we print the distances of all vertices from the starting vertex.

Dijkstra's algorithm is a popular shortest path algorithm that can be used to find the shortest path between two vertices in a graph. OpenMP is a programming model that allows for parallel programming in shared-memory architectures.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

When implementing Dijkstra's algorithm using OpenMP, the algorithm can be parallelized by dividing the work among multiple threads. This can be achieved by partitioning the graph into smaller subgraphs, and assigning each subgraph to a separate thread.

One way to do this is to use a priority queue to keep track of the vertices that need to be visited, with the vertex with the lowest distance being the next to be visited. Each thread can take vertices from the priority queue in parallel, and update their distances to neighboring vertices. This can be done by assigning each thread a portion of the graph and having it work on its own subgraph in parallel.

To ensure correctness, synchronization mechanisms such as locks or atomic operations may be necessary to avoid data races and ensure that only one thread updates a vertex's distance at a time.

Overall, using OpenMP can greatly speed up Dijkstra's algorithm, especially for large graphs, by taking advantage of multiple processors in a shared-memory system.

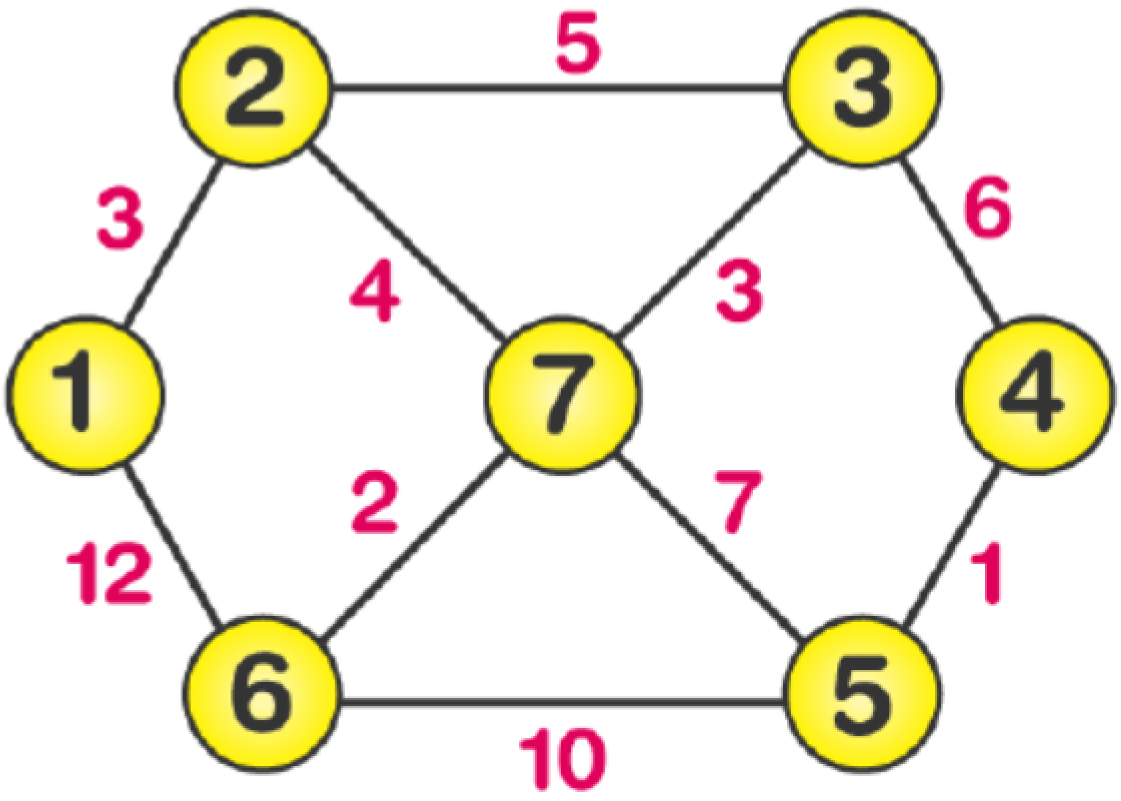
Code : Link

```
code ="""
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define INF 999999

void dijkstra(int** graph, int n, int start)
{
    int* dist = (int*)malloc(n * sizeof(int));
    int* visited = (int*)malloc(n * sizeof(int));

    // initialize the distance array and visited array
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        dist[i] = INF;
        visited[i] = 0;
    }
    // set the distance to the starting node as 0
    dist[start] = 0;

    // find shortest path for all vertices
    #pragma omp parallel for
    for (int count = 0; count < n - 1; count++) {
        // find the vertex with minimum distance value
        int min_dist = INF;
        int u;
        #pragma omp parallel for
        for (int v = 0; v < n; v++) {
            if (!visited[v] && dist[v] <= min_dist) {
                min_dist = dist[v];
                u = v;
            }
        }

        // mark the selected vertex as visited
        visited[u] = 1;

        // update the distance values of the adjacent vertices of the
selected vertex
        #pragma omp parallel for
        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // print the distances of all vertices from the starting vertex
    printf("Vertex \\t Distance from Start\\n");
    for (int i = 0; i < n; i++) {
        printf("%d \\t\\t %d\\n", i, dist[i]);
    }

    free(dist);
    free(visited);
}

int main()
{
    int n, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    int** graph = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        graph[i] = (int*)malloc(n * sizeof(int));
    }

    printf("Enter the adjacency matrix:\\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    dijkstra(graph, n, start);

    for (int i = 0; i < n; i++) {
        free(graph[i]);
    }
    free(graph);

    return 0;
}

"""
text_file=open("Dijkstra.c","w")
text_file.write(code)
text_file.close()
!gcc -o Dijkstra -fopenmp Dijkstra.c
```

Name : Manthan Gopal Dhole TY IT
Id : 201080020 Parallel Computing

```
!./Dijkstra
```

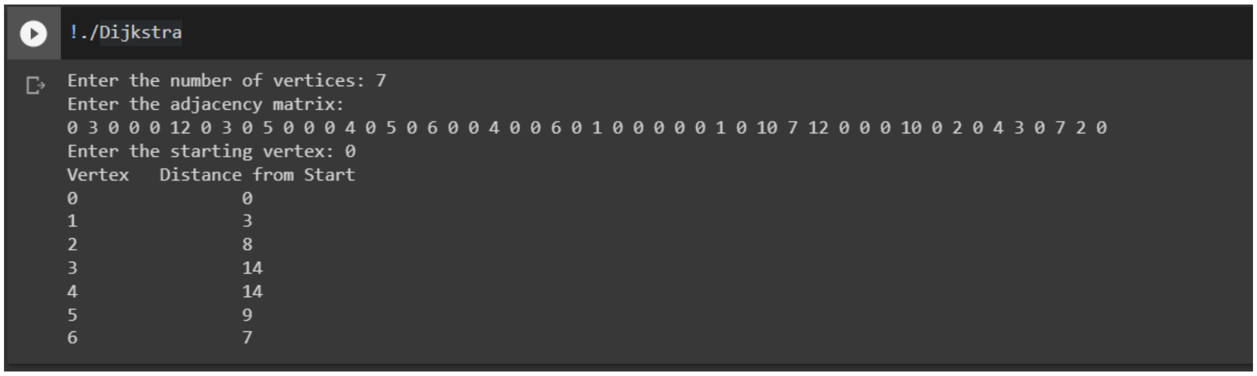
Input Graph :



```
0 3 0 0 0 12 0
3 0 5 0 0 0 4
0 5 0 6 0 0 4
0 0 6 0 1 0 0
0 0 0 1 0 10 7
12 0 0 0 10 0 2
0 4 3 0 7 2 0
```

Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

Output :



```
!./Dijkstra
```

```
Enter the number of vertices: 7
Enter the adjacency matrix:
0 3 0 0 0 12 0 3 0 5 0 0 0 4 0 5 0 6 0 0 4 0 0 6 0 1 0 0 0 0 0 1 0 10 7 12 0 0 0 10 0 2 0 4 3 0 7 2 0
Enter the starting vertex: 0
Vertex   Distance from Start
0               0
1               3
2               8
3               14
4               14
5               9
6               7
```

Conclusion:

Overall, using OpenMP can greatly speed up Dijkstra's algorithm, especially for large graphs, by taking advantage of multiple processors in a shared-memory system.