1.Topological Sort:

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX 100


typedef struct {

    int edges[MAX][MAX];

    int numVertices;

} Graph;


void bfs(Graph *g, int start, int *distances) {

    bool visited[MAX] = {false};

    int queue[MAX], front = 0, rear = 0;


    visited[start] = true;

    distances[start] = 0;

    queue[rear++] = start;


    while (front < rear) {

        int current = queue[front++];

        for (int i = 0; i < g->numVertices; i++) {

            if (g->edges[current][i] && !visited[i]) {

                visited[i] = true;

                distances[i] = distances[current] + 1;

                queue[rear++] = i;

            }

        }

    }
}


int main() {
    Graph g = { .numVertices = 5, .edges = { {0, 1, 1, 0, 0}, {1, 0, 0, 1, 1}, {1, 0, 0, 0, 1}, {0, 1, 0, 0, 1}, {0, 1, 1, 1, 0} } };
    int distances[MAX] = {0};


    bfs(&g, 0, distances);


    for (int i = 0; i < g.numVertices; i++) {
        printf("Distance from 0 to %d: %d\n", i, distances[i]);
    }


    return 0;
}
```

Output:

```
Distance from 0 to 0: 0
Distance from 0 to 1: 1
Distance from 0 to 2: 1
Distance from 0 to 3: 2
Distance from 0 to 4: 2
```

2.Unweighted shortest path algorithm

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100
typedef struct {
    int edges[MAX][MAX];
    int numVertices;
} Graph;
void bfs(Graph *g, int start, int *distances) {
    bool visited[MAX] = {false};
    int queue[MAX], front = 0, rear = 0;
    visited[start] = true;
    distances[start] = 0;
    queue[rear++] = start;
    while (front < rear) {
        int current = queue[front++];
        for (int i = 0; i < g->numVertices; i++) {
            if (g->edges[current][i] && !visited[i]) {
                visited[i] = true;
                distances[i] = distances[current] + 1;
                queue[rear++] = i;
            }
        }
    }
}
int main() {
    Graph g = {
        .numVertices = 5,
        .edges = {
            {0, 1, 1, 0, 0},
            {1, 0, 0, 1, 1},
            {1, 0, 0, 0, 1},
            {0, 1, 0, 0, 1},
            {0, 1, 1, 1, 0}
        }
    };
    int distances[MAX] = {0};
    bfs(&g, 0, distances);
    for (int i = 0; i < g.numVertices; i++) {
        printf("Distance from 0 to %d: %d\n", i, distances[i]);
    }

    return 0;
}
```

Output:

```
 Distance from 0 to 0: 0
Distance from 0 to 1: 1
Distance from 0 to 2: 1
Distance from 0 to 3: 2
Distance from 0 to 4: 2
```

3. Weighted shortest path algorithm(Dijkstra)

Code:

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == 0 && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;
        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];
            }
        }
    }
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t\t %d\n", i, dist[i]);
    }
}
int main() {
    int graph[V][V] = { {0, 10, 0, 30, 100},
                        {10, 0, 50, 0, 0},
                        {0, 50, 0, 20, 10},
                        {30, 0, 20, 0, 60},
                        {100, 0, 10, 60, 0} };
    dijkstra(graph, 0);
    return 0;
}
```

Output:

```
Vertex Distance from Source
0                 0
1                 10
2                 50
3                 30
```

```
4                    60
```

4.PRIM'S:

Code:

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    int mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}
int main() {
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    primMST(graph);
    return 0;
}
```

Output:

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

5.KRUSKAL'S :

Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int u, v, weight;
} Edge;
int find(int parent[], int i) {
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}
void unionSet(int parent[], int x, int y) {
    int xset = find(parent, x);
    int yset = find(parent, y);
    if (xset != yset)
        parent[xset] = yset;
}
int compare(const void *a, const void *b) {
    return ((Edge *)a)->weight - ((Edge *)b)->weight;
}
void kruskal(Edge edges[], int numEdges, int numVertices) {
    qsort(edges, numEdges, sizeof(edges[0]), compare);
    int parent[numVertices];
    for (int i = 0; i < numVertices; i++)
        parent[i] = -1;
    printf("Edges in the Minimum Spanning Tree:\n");
    for (int i = 0; i < numEdges; i++) {
        int u = edges[i].u;
        int v = edges[i].v;
        if (find(parent, u) != find(parent, v)) {
            printf("%d -- %d == %d\n", u, v, edges[i].weight);
            unionSet(parent, u, v);
        }
    }
}
```

```
int main() {
    Edge edges[] = { {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4} };
    int numEdges = sizeof(edges) / sizeof(edges[0]);
    int numVertices = 4;
    kruskal(edges, numEdges, numVertices);
    return 0;
}
```

Output:

```
Edges in the Minimum Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

6. Breadth First Search:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int queue[MAX], front = -1, rear = -1;
void enqueue(int value) {
    if (rear == MAX - 1) {
        printf("Queue is full\n");
    } else {
        if (front == -1) front = 0;
        rear++;
        queue[rear] = value;
    }
}
int dequeue() {
    if (front == -1) {
        printf("Queue is empty\n");
        return -1;
    } else {
        int value = queue[front];
        front++;
        if (front > rear) front = rear = -1;
        return value;
    }
}
void bfs(int graph[MAX][MAX], int start, int n) {
    int visited[MAX] = {0};
    enqueue(start);
    visited[start] = 1;

    while (front != -1) {
        int current = dequeue();
        printf("%d ", current);

        for (int i = 0; i < n; i++) {
            if (graph[current][i] == 1 && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}
int main() {
    int n = 5;
    int graph[MAX][MAX] = {
        {0, 1, 1, 0, 0},
        {1, 0, 0, 1, 1},
        {1, 0, 0, 0, 0},
        {0, 1, 0, 0, 1},
        {0, 1, 0, 1, 0}
    };
    printf("BFS Traversal starting from vertex 0:\n");
    bfs(graph, 0, n);
    return 0;
}
```

Output:

```
BFS Traversal starting from vertex 0:
0 1 2 3 4
```

7. Death First Search:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int visited[MAX];
int graph[MAX][MAX];
int n;
void dfs(int vertex) {
    visited[vertex] = 1;
    printf("%d ", vertex);
    for (int i = 0; i < n; i++) {
        if (graph[vertex][i] == 1 && !visited[i]) {
            dfs(i);
        }
    }
}

int main() {
    n = 5;
    graph[0][1] = graph[1][0] = 1;
    graph[0][2] = graph[2][0] = 1;
    graph[1][3] = graph[3][1] = 1;
    graph[2][4] = graph[4][2] = 1;
    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }
    printf("Depth First Search starting from vertex 0:\n");
    dfs(0);
    return 0;
}
```

Output:

```
Depth First Search starting from vertex 0:
0 1 3 2 4
```