



You:

bfs,dfs,dijkstra,bellmanford,floyed-warshall code in cpp without using "auto","structure","typedef"



ChatGPT:

Here are the implementations of BFS, DFS, Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm in C++ without using auto, structure, or typedef:

Breadth-First Search (BFS)

```
cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

void bfs(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    queue<int> q;
    q.push(start);
    visited[start] = true;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";
        for (int i = 0; i < graph[current].size(); ++i) {
            int neighbor = graph[current][i];
            if (!visited[neighbor]) {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}

int main() {
    vector<vector<int>> graph = {{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1}, {2}};
    bfs(graph, 0);
    return 0;
}
```

Depth-First Search (DFS)

```
cpp
#include <iostream>
#include <stack>
#include <vector>

using namespace std;

void dfs(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    stack<int> s;
    s.push(start);
    visited[start] = true;
    while (!s.empty()) {
        int current = s.top();
        s.pop();
        cout << current << " ";
        for (int i = 0; i < graph[current].size(); ++i) {
            int neighbor = graph[current][i];
            if (!visited[neighbor]) {
                s.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}

int main() {
```

```
vector<vector<int>> graph = {{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1}, {2}};
dfs(graph, 0);
return 0;
}
```

Dijkstra's Algorithm

```
cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

void dijkstra(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<int> distance(n, numeric_limits<int>::max());
    distance[start] = 0;
    vector<bool> visited(n, false);

    for (int i = 0; i < n - 1; i++) {
        int minDist = numeric_limits<int>::max(), minIndex;
        for (int j = 0; j < n; j++) {
            if (!visited[j] && distance[j] <= minDist) {
                minDist = distance[j];
                minIndex = j;
            }
        }
        visited[minIndex] = true;
        for (int j = 0; j < n; j++) {
            if (!visited[j] && graph[minIndex][j] && distance[minIndex] != numeric_limits<int>::max() &&
                distance[minIndex] + graph[minIndex][j] < distance[j]) {
                distance[j] = distance[minIndex] + graph[minIndex][j];
            }
        }
    }

    cout << "Vertex    Distance from Source" << endl;
    for (int i = 0; i < n; i++) {
        cout << i << "          " << distance[i] << endl;
    }
}

int main() {
    vector<vector<int>> graph = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}};

    dijkstra(graph, 0);
    return 0;
}
```

Bellman-Ford Algorithm

```
cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

void bellmanFord(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<int> distance(n, numeric_limits<int>::max());
    distance[start] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                if (graph[j][k] && distance[j] != numeric_limits<int>::max() && distance[j] + graph[j][k] < distance[k]) {
                    distance[k] = distance[j] + graph[j][k];
                }
            }
        }
    }
}
```

```

    }
    }
}

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] && distance[i] != numeric_limits<int>::max() && distance[i] + graph[i][j] < distance[j]) {
            cout << "Graph contains negative weight cycle" << endl;
            return;
        }
    }
}

cout << "Vertex    Distance from Source" << endl;
for (int i = 0; i < n; i++) {
    cout << i << "    " << distance[i] << endl;
}
}

int main() {
    vector<vector<int>> graph = {{0, 6, 0, 0, 0},
                                {0, 0, 5, -4, 0},
                                {0, 0, 0, 0, -2},
                                {0, 1, 0, 0, 0},
                                {0, 0, 0, 2, 0}};

    bellmanFord(graph, 0);
    return 0;
}

```

Floyd-Warshall Algorithm

```

cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

void floydWarshall(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<vector<int>> distance(graph);

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (distance[i][k] != numeric_limits<int>::max() && distance[k][j] != numeric_limits<int>::max() &&
                    distance[i][k] + distance[k][j] < distance[i][j]) {
                    distance[i][j] = distance[i][k] + distance[k][j];
                }
            }
        }
    }

    cout << "The following matrix shows the shortest distances between every pair of vertices" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (distance[i][j] == numeric_limits<int>::max()) {
                cout << "INF ";
            } else {
                cout << distance[i][j] << " ";
            }
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> graph = {{0, 5, numeric_limits<int>::max(), 10},
                                {numeric_limits<int>::max(), 0, 3, numeric_limits<int>::max()},
                                {numeric_limits<int>::max(), numeric_limits<int>::max(), 0, 1},
                                {numeric_limits<int>::max(), numeric_limits<int>::max(), numeric_limits<int>::max(), 0}};

    floydWarshall(graph);
    return 0;
}

```

Feel free to run these codes and modify them as needed.