Optimize water distribution in C++

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

using namespace std;

class Pair {
public:
    int vtx;
    int wt;
    Pair(int vtx, int wt) {
        this->vtx = vtx;
        this->wt = wt;
    }
    bool operator>(const Pair& other) const {
        return this->wt > other.wt;
    }
};

int minCostToSupplyWater(int n, vector<int>& wells,
vector<vector<int>>& pipes) {
    vector<vector<Pair>> graph(n + 1);
    for (const auto& pipe : pipes) {
        int u = pipe[0];
        int v = pipe[1];
        int wt = pipe[2];
        graph[u].emplace_back(v, wt);
        graph[v].emplace_back(u, wt);
    }
    for (int i = 1; i <= n; ++i) {
        graph[i].emplace_back(0, wells[i - 1]);
        graph[0].emplace_back(i, wells[i - 1]);
    }

    int ans = 0;
    priority_queue<Pair, vector<Pair>, greater<Pair>> pq;
    pq.emplace(0, 0);
    vector<bool> vis(n + 1, false);

    while (!pq.empty()) {
        Pair rem = pq.top();
        pq.pop();
        if (vis[rem.vtx]) continue;
        ans += rem.wt;
        vis[rem.vtx] = true;
        for (const Pair& nbr : graph[rem.vtx]) {
            if (!vis[nbr.vtx]) {
                pq.push(nbr);
            }
        }
    }
    return ans;
}

int main() {
    int v = 3, e = 2;
    vector<int> wells = {1, 2, 2};
    vector<vector<int>> pipes = {{1, 2, 1}, {2, 3, 1}};

    cout << minCostToSupplyWater(v, wells, pipes) <<
```

```cpp
int v = 3, e = 2;
vector<int> wells = {1, 2, 2};
vector<vector<int>> pipes = {{1, 2, 1}, {2, 3, 1}};
```

- v = 3: Number of houses (vertices).
- wells = {1, 2, 2}: The cost to build a well at house 1, 2, and 3.
- pipes = {{1, 2, 1}, {2, 3, 1}}: The pipes connecting houses, with respective costs.

Step 1: Construct the Graph

We begin by creating an adjacency list that represents the graph, including both the pipes and wells.

- **Graph Construction**:
  - Create an adjacency list graph with v + 1 = 4 nodes (including the virtual node 0).
  - Add edges for the pipes between houses:
    - Pipe from 1 to 2 with cost 1.
    - Pipe from 2 to 3 with cost 1.
  - Add edges for the wells:
    - Well for house 1 (cost 1), connect node 0 to node 1.
    - Well for house 2 (cost 2), connect node 0 to node 2.
    - Well for house 3 (cost 2), connect node 0 to node 3.
- **Graph Representation**:

  Node 0 (virtual node) → {(1, 1), (2, 2), (3, 2)}
  Node 1 → {(2, 1), (0, 1)}
  Node 2 → {(1, 1), (3, 1), (0, 2)}
  Node 3 → {(2, 1), (0, 2)}

Step 2: Prim's Algorithm with Min-Heap

We will use **Prim's Algorithm** to find the Minimum Spanning Tree (MST) with a priority queue (min-heap).

- **Priority Queue Initialization**: Start with node 0 (virtual node), which has no cost yet, so we push (0, 0) into the priority queue.

endl;
 	
    return 0;
}
	Priority Queue: [(0, 0)]
	
	• **Step 3: First Iteration (start with node 0)**
	  ○ **Pop from the priority queue**: (0, 0) is popped, meaning we're at the virtual node 0 with a cost of 0.
	  ○ **Visit Node 0** and explore its neighbors (nodes 1, 2, 3):
	    ▪ Add the edges to the priority queue:
	      ▪ Edge (0 → 1, cost 1)
	      ▪ Edge (0 → 2, cost 2)
	      ▪ Edge (0 → 3, cost 2)
	
	After this step:
	
	Priority Queue: [(1, 1), (2, 2), (2, 3)]
	Visited nodes: [0]
	Total Cost: 0
	
	• **Step 4: Second Iteration (pop node 1)**
	  ○ **Pop from the priority queue**: (1, 1) is popped, meaning we're now at node 1 with a cost of 1.
	  ○ **Visit Node 1** and explore its neighbors:
	    ▪ Node 0 has already been visited, so ignore.
	    ▪ Add edge (1 → 2, cost 1) to the priority queue.
	
	After this step:
	
	Priority Queue: [(1, 2), (2, 3), (1, 2)]
	Visited nodes: [0, 1]
	Total Cost: 1
	
	• **Step 5: Third Iteration (pop node 2)**
	  ○ **Pop from the priority queue**: (1, 2) is popped, meaning we're now at node 2 with a cost of 1.
	  ○ **Visit Node 2** and explore its neighbors:
	    ▪ Node 1 has already been visited, so ignore.
	    ▪ Node 3 is unvisited, so add edge (2 → 3, cost 1) to the priority queue.
	    ▪ Node 0 has already been visited, so ignore.
	
	After this step:
	
	Priority Queue: [(1, 3), (2, 3), (1, 2)]

	Visited nodes: [0, 1, 2]
	Total Cost: 2
	
	• **Step 6: Fourth Iteration (pop node 3)**
	  ○ **Pop from the priority queue**: (1, 3) is popped, meaning we're now at node 3 with a cost of 1.
	  ○ **Visit Node 3** and explore its neighbors:
	    ▪ Node 2 has already been visited, so ignore.
	    ▪ Node 0 has already been visited, so ignore.
	
	After this step:
	
	Priority Queue: [(2, 3)]
	Visited nodes: [0, 1, 2, 3]
	Total Cost: 3
	
	• **Step 7: Termination**
	  ○ The priority queue is empty, and all nodes are visited.
	  ○ **Final Total Cost**: 3.
	
	Final Output
	
	The minimum cost to supply water is: 3

Output:- 3	