Mustafa Kemal Kücük mustafak@uni-bremen.com
Darius Dolha dolha@uni-bremen.de
Vasu Yaduvanshi vasu@uni-bremen.de

Übungsgruppe 51

1 Übung 4.1

Algorithm

The algorithm employs a **Breadth-First Search (BFS)** to attempt a 2-coloring of the vertices:

Algorithm 1 Check Bipartiteness

```
Initialize a color map color[v] for all v ∈ V to undefined.
for each vertex v ∈ V do
    if color[v] is undefined then
        Assign color[v] ← 0.
        Perform BFS starting from v.
        while exploring each vertex u in the BFS queue do
            for each neighbor w of u do
                if color[w] is undefined then
                    Assign color[w] ← 1 − color[u].
                else if color[w] = color[u] then
                    Return False (Graph is not bipartite).
                end if
            end for
        end while
    end if
end for
Return True (Graph is bipartite).
```

Proof of Correctness

Lemma: Properties of a Bipartite Graph

A graph is bipartite if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that all edges $e \in E$ connect vertices from different sets. Alternatively, a graph is bipartite if it does not contain any odd-length cycles.

Inductive Proof

We prove the correctness of the algorithm using induction on the number of vertices in the graph.

Base Case: Let $|V| = 1$. A single vertex has no edges and is trivially bipartite, as the vertex set can be partitioned into $V_1 = \{v\}$ and $V_2 = \emptyset$. The algorithm works correctly in this case.

Inductive Hypothesis: Assume the algorithm works correctly for all graphs with n vertices.

Inductive Step: Let $G = (V, E)$ be a graph with $n + 1$ vertices.

1. Select any starting vertex $v \in V$ and perform a BFS.
2. During the BFS, each visited vertex is assigned a color (either 0 or 1). A conflict occurs if two adjacent vertices share the same color.
3. If a conflict occurs, the graph contains an odd-length cycle, and thus is not bipartite.
4. If no conflict occurs, the graph is bipartite, as all vertices are successfully 2-colored.

Hence, the algorithm works for $n + 1$ vertices, completing the induction.

Runtime Analysis

The algorithm uses a Breadth-First Search (BFS):

- Each vertex is visited exactly once: $O(|V|)$.
- Each edge is explored exactly once: $O(|E|)$.

Thus, the total runtime is $O(|V| + |E|)$.

Conclusion

The algorithm correctly determines whether a graph is bipartite and meets the required runtime of $O(|V| + |E|)$.

2 Übung 4.2

1. Take any $e_1 \in T_1 \setminus T_2$. When we remove e_1 from T_1, the graph separates into two components A and B.
2. Since T_2 is a spanning tree, there must be a path in T_2 connecting these components. This path must contain at least one edge $e_2 \in T_2 \setminus T_1$ that crosses from A to B.
3. Adding e_1 to T_2 creates unique cycle C in $T_2 \cup \{e_1\}$. The edge e_2 we found must lie on this cycle C.
4. Consider the costs $c(e_1)$ and $c(e_2)$:
   - If $c(e_1) > c(e_2)$: Then replacing e_1 with e_2 in T_1 would give a tree with lower cost, contradicting T_1 being an MST
   - If $c(e_2) > c(e_1)$: Then replacing e_2 with e_1 in T_2 would give a tree with lower cost, contradicting T_2 being an MST
   - Therefore $c(e_1) = c(e_2)$
5. Let $T_3 = (T_1 \setminus \{e_1\}) \cup \{e_2\}$. We need to show T_3 is an MST:
   - T_3 is connected because e_2 connects the components A and B created by removing e_1
   - T_3 has $|V| - 1$ edges (same as T_1), so it's a tree
   - T_3 has the same total cost as T_1 since $c(e_1) = c(e_2)$
   - Since T_1 is an MST, T_3 must also be an MST
6. Therefore, it is proved, that for any edge $e_1 \in T_1 \setminus T_2$, we can find an edge $e_2 \in T_2 \setminus T_1$ such that replacing e_1 with e_2 yields another MST.

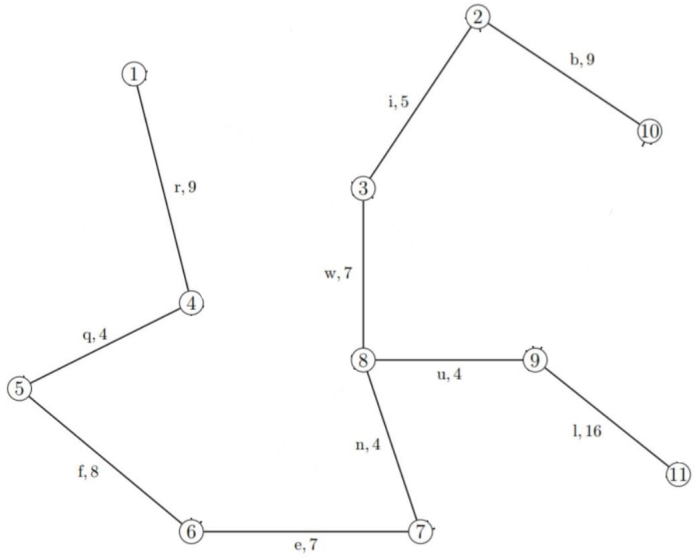
3 Übung 4.3

Prim algorithm

We will use Prim's algorithm to find the minimum spanning tree. We will select the edges in the following order, starting from node 1: We always select the edge with the lowest cost such that it has exactly one end point in our set S.

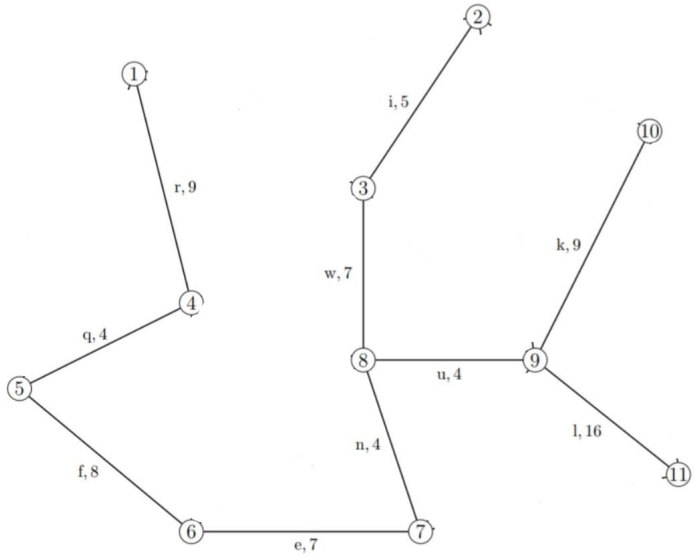
The edge r connects nodes 1 and 4. It has a weight of 9. $S = \{1, 4\}$
The edge q connects nodes 4 and 5. It has a weight of 4. $S = \{1, 4, 5\}$
The edge f connects nodes 5 and 6. It has a weight of 8. $S = \{1, 4, 5, 6\}$
The edge e connects nodes 6 and 7. It has a weight of 7. $S = \{1, 4, 5, 6, 7\}$
The edge n connects nodes 7 and 8. It has a weight of 4. $S = \{1, 4, 5, 6, 7, 8\}$
The edge u connects nodes 8 and 9. It has a weight of 4. $S = \{1, 4, 5, 6, 7, 8, 9\}$
The edge w connects nodes 8 and 3. It has a weight of 7. $S = \{1, 4, 5, 6, 7, 8, 9, 3\}$
The edge i connects nodes 3 and 2. It has a weight of 5. $S = \{1, 4, 5, 6, 7, 8, 9, 3, 2\}$

The edge b connects nodes 2 and 10. It has a weight of 9. $S = \{1, 4, 5, 6, 7, 8, 9, 3, 2, 10\}$
The edge l connects nodes 9 and 11. It has a weight of 16. $S = \{1, 4, 5, 6, 7, 8, 9, 3, 2, 10, 11\}$
Having found a tree with all the initial nodes we can stop.



Kruskal algorithm

We will use Kruskal's algorithm to find the minimum spanning tree. This algorithm involes selecting the edges with the lowet cost, making sure every time that we do not from a cycle. We will select the edges in the following order.
The edge q connects nodes 4 and 5. It has a weight of 4.
The edge n connects nodes 7 and 8. It has a weight of 4.
The edge u connects nodes 8 and 9. It has a weight of 4.
We also try node m, but see that it forms a cycle so it's invalid.
The edge i connects nodes 2 and 3. It has a weight of 5.
The edge w connects nodes 3 and 8. It has a weight of 7.
The edge e connects nodes 6 and 7. It has a weight of 7.
The edge f connects nodes 5 and 6. It has a weight of 8.
The edge r connects nodes 1 and 2. It has a weight of 9.
The edge k connects nodes 9 and 10. It has a weight of 9.
Then we try nodes j,b,a,s,h, but they are invalid since it we would have a cycle.
The edge l connects nodes 9 and 11. It has a weight of 16.
Having found a tree with all the initial nodes(spanning tree) we can stop.


1
2
3
4
5
6
7
8
9
10
11
r, 9
i, 5
w, 7
k, 9
q, 4
u, 4
n, 4
l, 16
f, 8
e, 7