|  |
| --- |
| 2025 |

Algorithms and Complexity

ASSIGNMENT 1

Aleksandre Avduli

Contents

1. Data Structures used…………………………………..……………2
2. Pseudo Codes…….…………………………………………………3
3. Proof of Correctness………………………………………………..5
4. Time Complecity……………………………………………………7
5. References…………………………………………………………..8

Algorithms and Complexity

1. Data Structures used

* ***MyList<T>:***

is a generic dynamic array (similar to Java’s ArrayList, but implemented from scratch) that supports automatic resizing and is used to store list of bond connections between dynos (edges), and stores data like component roots or selected dynos. It allows the program to handle unknown number of input bonds. Since the number of possible bonds *k* is only known after reading the input file, this dynamic array allows flexible storage of bond information. **-edge[]** array represents a possible bond between two dynos

* ***MySet:***

is an integer set implementation used to track unique components after union-find operations. It ensures that each connected group of dynos is counted once when determining how many buckets are needed. The number of unique components tells us how many buckets must be installed (one/component). This ensures the cost is minimized by installing the fewest possible buckets.

* ***Union-Find:***

is the basic algorithm structure that determines the dynos that are connected together. Each connected component of dynos share one bucket, so it is important to know how many connected components are there, in order to know how many buckets are needed.

-parent[] array tracks the root of each component;

-find(x) determines the representative of the component that a dyno belongs to -union(x, y) merges two components if there is a bond between them.

2. Pseudo-Codes of the Algorithms

//Union-Find Algorithm function initialize\_union\_find (n) : for i from 1 to n: parent [i] ← i

//Find Operation function find(x): if parent[x] ≠ x then: parent[x] ← find(parent[x]) return parent[x]

//Union Operation function union(x, y): x\_root ← find(x) y\_root ← find(y) if x\_root ≠ y\_root: parent[y\_root] ← x\_root

//Processing Bonds for i from 1 to k: read u, v edges.add(edge(u, v)) union(u, v)

//Identirying unique components initialize components as empty set for i from 1 to n:

root ← find(i) if root not in components: components.add(root)

//Calculating cost components\_count ← components.size() total\_edges ← n - components\_count

total\_cost ← components\_count \* bucket\_cost + total\_edges \* bond\_cost return total\_cost

3. Proof of correctness

The algorithm begins by reading input values: the number of dynos, the number of possible bonds, the cost of installing a bucket, and the cost of a bond. Each dyno is initially assigned to its component. As the bond list is processed, the union operation connects dynos into components. After all bonds are processed, the find function is used on each dyno to identify the root. These roots are added to MySet to determine how many distinct connected components there are. Each component requires at least one bucket, and within each component, there are needed components\_size-1 bonds. So, the total cost is calculated as the cost of 1 bucket per component plus the cost of the minimal number of bonds.

**total\_cost = components\_count \* bucket\_cost + total\_edges \* bond\_cost**

Since there are needed components\_size-1 bonds, the number of total\_edges is equal to n-components\_count, so the formula becomes:

**total\_cost = components\_count \* bucket\_cost + (n-components\_counts) \* bond\_cost**

To prove the correctness of this algorithm, we use mathematical induction. n = number of dynos

Base case: n=1 (when there is only 1 dyno, there are no bonds) total\_cost = 1 \* bucket\_cost + 0 \* bond\_cost = bucket\_cost

Inductive step:

Assume the algorithm (formula) is true. (n=k) **total\_costk  = components\_countk  \* bucket\_cost + (k-components\_countsk) \* bond\_cost** Prove for n=k+1 (adding a new dyno):

The new dyno either:

- connects to an existing component (not increasing the number of components) components\_countk+1 = components\_countk

(k+1) - components\_countk+1 = (k - components\_countk) + 1 Therefore: total\_costk+1 = components\_countk \* bucket\_cost+((k−components\_countk)+1) \* bond\_cost

**=** components\_countk \* bucket\_cost+(k−components\_countk) \* bond\_cost+1 \* bond\_cost

= (components\_countk \* bucket\_cost+(k−components\_countk) \* bond\_cost)+bond\_cost

= total\_costk+bucket\_cost

**This matches the formula, as it adds the cost of the new bond, so the inductive step is correct till now…**

-or forms a new component. components\_countk+1 = components\_countk + 1 (k + 1) – components\_countk+1 = k – component\_countk

total\_costk+1 = (components\_countk + 1) \* bucket\_cost+(k−components\_countk) \* bond\_cost

= components\_countk \* bucket\_cost+bucket\_cost+(k−components\_countk) \* bond\_cost

= [components\_countk \* bucket\_cost+(k−components\_countk) \* bond\_cost]+bucket\_cost

= total\_costk+bucket\_cost

**This also matches our formula for the same reason, indicating that the inductive step is fullfilled and the algorithm is correct.**

4. Time Complexity

1. Edge class

This class only contains a constructor, having constant time complexity O(1)

1. MyList<T> class

**add(T value) : O(1)**

**get (int index) : O(1)** (direct array access) **resize() : O(n)** (copies n elements) **size() : O(1)** (just gets the size of the array)

1. MySet class

**add(T value) : worst-case scenario O(n)** (because of *contains*) **contains(int value) : O(n)** (linear search through the array) **size() : O(1)**

1. BigWeather class

**Input reading : O(n+k)** (initializing parent array and reading bonds)

**Union-Find : O(k \* α(n)) ≈ O(k)** (“These optimizations reduce the average time complexity of each find and union operation to O(α(n)), where α(n) is the inverse Ackermann function.”) [1]

**Collecting components : O(n2)** (1 find per *dyno* + up to n *contains*)

**Calculating total cost : O(1)** (basic arithmetics)

Overall Time Complexity:

Having n as the number of dynos, and k as the number of bonds, the time complexity of the algorithm is:

**O (n2 + k)**

5. References

[1] The Linux Kernel Documentation. (n.d.). Union-find algorithm. https://docs.kernel.org/coreapi/union\_find.html