**Project Report**

**1.General Overview**

The problem presented is a simplified version of Uncapacitated Facility Location Problem (UFLP). We can think of dynos as being clients of UFLP, hosting a bucket is being the same as opening facilities, and bonds represent client-facility connections. But it is simplified for a number of reasons described below, which really help in introducing us to these very important CS problems:

1. **Uncapacitated:** Buckets can serve many dynos.
2. **Uniform cost:** All bonds and buckets have a fixed cost.
3. **Possible connections known:** The set of possible connections is given, and this tremendously helps us as we will see later.

Specifically, the problem resolves around a weather forecasting company named BigWeather and like all companies it is trying to minimize the spending budget by reducing unnecessary costs. In the big grid used by the company dynos are the processing machines, but for them to work they must have access to a set of data organized in specialized high performance cache components, namely buckets. So, it can do this directly by hosting the bucket itself or through another dyno it is connected via bidirectional bonds that exist between dynos.

However, like in real life certain restrictions like network zones and other grid restrictions make it so that only certain dynos can be connected directly. This is beautifully simulated and implemented by giving the number of possible bonds(k) and specifying exactly which bonds can exist.

This assignment makes a beautiful simulation of this real-world problem, by providing the required information in a specified file which we need to process. The first line has information regarding the number of dynos (1…n), possible bonds(k), bucket and bond cost. And in the following k lines the possible bonds are provided. The output should be the cost of the cheapest configuration; the base problem is only interested in the cost not the specific one. Other outputs are part of the bonus discussion, and we will describe and explain later.

**2.Why this algorithm?**

Before deciding on this algorithm that implements the base solution utilizing decomposition of the graph with DFS and then a greedy approach to estimate the lowest of two possible costs for each component, we were considering another very popular approach MST(Kruskal) + Virtual Node. We were introduced to this unique approach by Leetcode 1168 (*Optimize Water Distribution in a Village*). Our assignment problem is conceptually similar but a simplified version of it:

* Each **dyno** can either host a **bucket** (analogous to a well),
* Or connect to other dynos via **bonds** (analogous to pipes).

The algorithm for this approach was given and was guaranteed to be optimal (to find the cost of the cheapest configuration) but we decided to stick to the concepts provided in our book covered in **Chapters 3–5** of *Algorithms and Complexity* by Dasgupta, Papadimitriou, and Vazirani, particularly:

* **Chapter 3**: Traversing the graph using DFS which helped us find the connected components.
* **Chapter 5**: Utilizing the greedy approach logic. We didn’t explicitly construct a MST using Kruskal or Prims algorithms, but we used the greedy MST logic.

The way we formulated the algorithm was by asking questions and then trying to find their answers. So that is the way we are going to describe how this algorithm came to be at least the solution to the base problem the bonus discussion comes later. The answer to these questions helped form the general idea of the solution:

1. Why can’t all dynos connect to a single dyno containing a bucket?

Well because, like in real life there are restrictions on the number of bonds and also which bonds are possible. This is given in the file by the number of possible bonds and exactly which can potentially exist.

1. If we consider the dynos to be the nodes of a graph and bonds the edges, what type of graph are we dealing with?

It was mentioned that the bonds were bidirectional which means the edges exist in both directions. Or put simply we are dealing with a undirected graph.

1. What is the purpose of the possible bonds? If we try and recreate these edges, what do we get?

Well, these bonds present the restrictions they tell us that don’t think of the problem as one big connected graph but rather as one or more connected components. Each of these components has its own cheapest cost and should all be considered in calculating the overall lowest cost.

1. Ok, we have the connected components now do we need to implement MST to find the lowest cost for each component. How will this work exactly, we know all the edges here have the same cost?

Explicitly constructing a MST for these scenarios is overkill, since we know two important pieces of information:

1. All the edges have the same cost.
2. **Property 1:** “Removing a cycle edge cannot disconnect a graph.”

What does this property one taken from the book mean exactly? In our problem we are only interested in connecting all dynos in a component with the fewest number of bonds. So, we need a spanning tree which in itself is a minimum spanning tree as all bond costs are the same. This spanning tree consists of n -1 edges(bonds) where n is number of nodes(dynos). Why? Well because that is a property of all trees and this way we ensure no cycles are present. So, simply that all nodes are connected and this cost is simply the bond\_cost times the number of edges (n-1) for all components.

1. What about buckets, how do we decide how many are in one component?

The specific number is not known beforehand, however we know that since each node is connected in a connected component of a undirected graph, there is a way for each node in the component to have access to the bucket. Following, this analogy results in the conclusion that we know for sure that at least one bucket must be present in each component.

1. Why did the professor said: “Yes a bucket can cost less than a bond, your algorithm must handle this case as well” ?

In this specific case when the cost of the bucket is less than that of a bond, for a component it is better that each of these nodes(dynos) host its own bucket. So, when considering the cost calculation for any component you have to decide whether 1 dyno hosting a bucket and the rest being bonds or all dynos hosting bonds is the best solution.

1. Can we guarantee that for sure all the time , the sum of the lowest cost from components is guaranteed to be the lowest cost solution of the overall graph(problem)? Is there any possibility that it is optimal for each component but fails to be so for the overall graph?

This can be confusing at first and needs some time to evaluate. Saying, it is optimal for the component but not for the graph means that there can exist a way we can merge these components so the cost is reduced , instead of hosting 2 bucket for example we host just 1 (we are considering the case where there is 1 bucket and the other bonds for each component as the other option all being buckets is trivial). But can this happen, can there exist such a bond that makes this connection possible? The answer is no. The possible bonds are all listed beforehand and if such connection is possible DFS would have explored and added it to the component. So, we know for sure that the algorithms is optimal , in other words it always finds the cheapest solution.

1. What other restrictions are there on the running time of the algorithm?

For this assignment what is more important is that the algorithm is correct and optimal. We are not asked of finding the fastest algorithm , as long as its run time is not exponential. As we will soon see this algorithm is one of the fastest to execute but that analysis comes later.

**3.Algorithm pseudocode and analysis**

The algorithms presented in this section use similar syntax and reasoning style that is familiar to us from *Algorithms* book by Dasgupta, Papadimitriou, and Vazirani. In some cases, like for example the DFS algorithm we added on top of the existing algorithm to make it so it works for our specific case. But in general, we try our best to follow as closely as possible to the material presented in the book, more specifically the one in chapters 3-5.

* 1. **Procedure: explore(G, v, component)**

procedure explore(G, v, component)

visited[v] = true

append v to component

**for** each u in G.adj[v]:

**if** not visited[u]:

explore(G, u, component)

This procedure has a similar structure to the procedure with the same name in Chapter 3.2.1 of the book. The only addition to this procedure is that it gets as an input a component list which helps keep track of all the nodes in the current subtree (component of the graph). So basically, we are trying to find each node that is reachable from a specific given node and then add (append) those nodes to components. So, like the book it first marks this specific vertex as visited and then recursively tries to explore all its neighbors and add them to component list.

**Correctness:** It is correct because we can follow a similar analogy as done in the book first proving the base case “for any k ≥ 0, all nodes within k hops from v get visited” which is trival. Then proving this for the more general case that if all nodes k hops away are visited, then so are all nodes k + 1 hops away.

**Time Complexity**: Every edge in the graph is visited (traversed) at least twice and every node exactly once. So, the run time is O(|V| + |E|) where |V| represent the number of nodes(vertices) and |E| the number of edges. Simply stated the run time is linear.

* 1. **Procedure: dfs(G)**

procedure dfs(G)

**for** all v ∈ V:

visited[v] = false

**for** all v ∈ V:

**if** not visited[v]:

component = empty list

explore(G, v, component)

processComponent(component)

So, this procedure also is similar to the one with the same name that appears on Chapter 3.2.2 of the book. Also, the only change is the addition of a component list to collect each connected component. We need these for cost evaluation later. So, utilizing the explore procedure we previously defined we can only find one such component, but a graph can be composed also of more than one. This is why to examine the whole graph we need to start elsewhere, in a node previously not explored. In the second loop, we iterate over each node of the graph. These unexplored nodes become the starting point for a new connected component, and we explore from it recursively. As we mentioned, what is new is that we collect each component in a list.

**Correctness:** I mean its correctness is closely related to the correctness of the explore method which we did previously. The first loop ensures that no vertex is mistakenly marked as visited before the algorithm begins. The second ensures each connected component is visited exactly once. No other outer loop iteration will re-enter that component. And it terminates after all these components have been explored. For each of them it makes a call to the processComponent , which as we will see later calculates the cost of the optimal solution and the number of such optimal solutions.

**Time complexity:** Each vertex is explored just once, thanks to the visited array. So in each exploration of a node there is done some fixed amount of work to mark the node as visited and then it proceeds to a loop where adjacent edges are scanned. We explained this loop a bit when we talked about the explore procedure. Since each node is visited exactly once and each edge at least twice, the run time is O(|V| + |E|), which determines the overall run time for dfs. A very efficient algorithm, a linear one.

**\*\*\* Kirchhoff’s Matrix-Tree Theorem \*\*\***

Before we continue to present and argue about the remaining algorithms it is super important that we first understand the theoretical and logical foundation that they rely on. The remaining algorithms as group accomplish two important tasks. They not only calculate the cheapest cost but also count the number of such cheapest configurations. The importance of this theorem is that it allows us to count the number of spanning trees in a graph by using linear algebra. In the discussion of question 6 we reached some important conclusions, namely we introduced two strategies that can be used to calculate the cost. **1.** Each dyno can host its own bucket. In this strategy there are no bonds. **2.** Only one dyno hosts the bucket and the others, utilizing bonds connect to this shared bucket. If we decide that the second strategy is the best for us the cheapest way to connect all dynos in the component is via a spanning tree. So, what is a spanning tree? Well it is a tree and as such it contain no cycle and as such it contains the minimal set of connections (edges) that keep the component fully connected without forming any cycles. They are minimum spanning trees by default because as explained previously all bond costs are the same**. \*However, for any given component there isn’t just one spanning tree.\*** And as such each of these spanning trees represents a a different valid configuration of bonds, and since we can choose any dyno to host the single bucket, the number of valid configurations for a component becomes: **nr of distinct configurations = nr of spanning trees​​ \* nr of ways to choose 1 bucket host​​.**

**Theorems main idea:** “ Let G=(V,E) be a connected undirected graph with n vertices. The number of distinct spanning trees of G is equal to any cofactor of its Laplacian matrix L.” What this actually means is that we first need to construct the Laplacian matrix L of the graph, then remove any one row and the corresponding column to get a smaller matrix L′. Then, number of spanning trees is simply det(L).

**Laplacian matrix:** The Laplacian matrix we mentioned is a n by n matrix where : **L[i][i]** = degree of node i (number of neighbors) **L[i][j]** = −1 if i and j are connected **L[i][j]** =0 otherwise. This matrix captures both the local and global connectivity of a graph in a matrix, n by n.

**Why does it benefit us? :** The determinant of the minor is the smaller matrix after we delete a row and a column. That captures how many cycle free ways we can connect all the dynos(nodes) which is exactly what spanning trees are. **So, instead of manually generating all trees, we use matrix operations to count them.** We will use this theorem to compute the number of distinct cheapest configurations for each component in our solution.

It is understandable this theorem is correct, however it is not the scope of this assignment to show the exact proof. We know it is true and use it in the following algorithms(procedures) that we will explore.

* 1. **Procedure: processComponent(component)**

procedure processComponent(component)

n = size of component **//With size being the number of nodes(dynos)**

costTree = bucketCost + (n - 1) × bondCost

costAllBuckets = n × bucketCost

**if** costTree < costAllBuckets:

totalCost = totalCost + costTree

treeCount = countSpanningTrees(component)

totalWays = totalWays + (treeCount × n)

**else if** costTree > costAllBuckets:

totalCost = totalCost + costAllBuckets

totalWays = 1 **//There is just one option all dynoes host buckets.**

**else:**

totalCost = totalCost + costTree

treeCount = countSpanningTrees(component)

totalWays = totalWays + (treeCount × n + 1)

We are now going to explore a new algorithm that was not present in the book but is still based heavily on Chapter 5, specifically on greedy comparison logic. The main goal of this algorithm is for any given component, to determine the cheapest way to cover all dynos with buckets and/or bonds and then calculate how many distinct configurations achieve that cost by calling the countSpanningTrees procedure. We should think of this algorithm as the decision-making core. As we know each configuration is identified uniquely by bonds and in which dyno the buckets are hosted. We as previously mentioned we don’t utilize Kruskal or Prim’s algorithms because all edge(bonds) weights are equal, so we rely on the graph theory “In any connected component of n nodes, every spanning tree uses exactly n−1 edges, and all such trees are equally optimal when edge weights are uniform”. So as we mentioned before for any connected component, there are only two reasonable strategies: **1.** Every dyno hosts its own bucket. For this strategy the **cost =** **n × bucketCost** , where n is number of nodes(dynos). And as such there is only **1 cheapest configuration.**

**2.** Only one dyno hosts the bucket and the others, utilizing bonds connect to this shared bucket. So , for this strategy **cost = bucketCost + (n - 1) × bondCost** , n being number of nodes. However here we have **multiple cheapest configurations**, depending on which dyno hosts the bucket or which spanning tree is used to connect the dynos.

The algorithm compares these 2 costs and choses the cheapest one. However, if both strategies have the same cost we calculate the number of distinct cheapest configurations that result from both strategies(we sum them).

**Correctness:** From the definition of trees we know that any connected component with n nodes has a spanning tree with exactly n - 1 edges. Since bond costs are uniform, any such tree is an MST. We don't need to construct the MST — we just use its known cost. Our algorithm all coverage options(the 2 strategies we mentioned) and it counts the configurations accurately. All bucket strategy has exactly 1 such configurations. For the one bucket + bonds strategy each spanning tree is a valid bond structure, and each node can be the host. Her we utilize the Kirchhoff's Theoremto count the number of such configurations which we know is correct. The last case where cost are equal the counting is done similar to the last one but we add 1 configuration for the all bucket case. Also, it avoids invalid solutions because it allows only spanning tree connections, which ensure minimal bonds and full connectivity without cycles. **Running time:** To analyze the running time of this algorithms is easy because as we see most of the operations are basic computer step and they can be thought as executing in constant time. I mean these basic computer steps include assignments, small integer arithmetic operations (left shift , addition) etc. However, the procedure makes two calls to the countSpanningTrees procedure, which as will in explain later taker O(n power 3) in its worst case. So, our algorithm is fast for small components which realistically we expect to get.

* 1. **Procedure: countSpanningTrees(component)**

procedure countSpanningTrees(component)

L ← buildLaplacian(component)

L′ ← remove first row and column from L

return determinant(L′)

We mentioned a bit before when analyzing the previous procedure that this algorithm, counts the number of distinct spanning trees for any given component. We have already established that these represent the number of distinct cheapest configurations. This procedure does this by closely implementing the logic of **Kirchhoff’s Matrix-Tree Theorem** which we introduced before. So first it construct the Laplacian matrix L of the component by making a call to the buildLaplacian procedure which we will cover next. Then it deletes the first row and column from L that fives us the smaller matrix L′. And in the end through a call to the procedure determinant taking as argument this smaller matrix , it returns the determinant of L′ , which by the theorem, gives us the number of spanning trees.

**Correctness:** The correctness of this algorithm follows directly from the correctness of **Kirchhoff’s Matrix-Tree Theorem.** It resembles exactly its logic and computation in a code format. If we follow this analogy, we see that the Laplacian matrix captures the connectivity structure. Diagonal entries are the degree of each node and off-diagonal entries are -1 if there’s an edge. Any cofactor of L gives us the number of spanning trees because we consistently remove the first row and column to get the smaller matrix L′. Simply put as long as the component is connected which is guaranteed by DFS and Laplacian matrix is build correct the determinant of L′ is exactly the number of spanning trees.

**Running Time:** The calls to build the Laplacian matrix through the call to the buildLaplacian procedure take O(n power 2) as we will see later. And the whole algorithm is dominated by the call the determinant procedure which takes O(n power 3) time to execute. So the overall running time is still O(n power 3).

* 1. **Procedure: buildLaplacian(component)**

procedure buildLaplacian(component)

n = size of component

L = n × n zero matrix

for i from 1 to n:

for each neighbor j of node i:

if i ≠ j:

L[i][j] ← L[i][j] - 1

L[i][i] ← L[i][i] + 1

return L

So as mentioned above this algorithm makes possible the creation of the Laplacian matrix L of the undirected graph induced by the component. So, first it initializes an n×n zero matrix L, where as we mentioned n is the number of dynos in the component. Then for every node i and neighbor j in the component it correctly updates the Laplacian matrix , machining sure the nodes i and j do not represent the same node. The result is the correct matrix which is returned to the calling method explored before.

**Correctness:** This algorithm is correct because by definition:

* L[i][i] = degree of node iii
* L[i][j]= -1 if there’s an edge
* Matrix is symmetric, since the graph is undirected.

The algorithm maps node labels to indices using a hash map, ensuring matrix coordinates are accurate regardless of dyno ID ordering.

**Running time:** Since this algorithm visits each edge once , the cost for this is O(∣E∣). Also both for loops execute at most n times, for an overall running time of O(n power 2) . This represent also the overall running time of the algorithm.

* 1. **Procedure: determinant(matrix)**

procedure buildLaplacian(component)

n = size of component

L = n × n zero matrix

for i from 1 to n:

for each neighbor j of node i:

if i ≠ j:

L[i][j] ← L[i][j] - 1

L[i][i] ← L[i][i] + 1

return L

This last algorithm uses **Gaussian elimination** to compute the determinant of the Laplacian minor L′(the smaller matrix). So, the main idea behind this algorithm is the usage of pivots. For each row i we find a pivot which is a non-zero entry in column i. Then if it is necessary, we swap rows and update sign of determinant. To make sure that we eliminate all entries below the pivot we use row operations and then we multiply together the diagonal elements together to get the determinant. In the last sentence this is returned to the calling method , to be precise countSpanningTrees.

**Correctness:** The correctness of this algorithms follows from the correctness of **Gaussian elimination** itself which is a textbook method for computing determinants. The sign changes are handled by the raw swaps which we mentioned earlier and division is handled carefully by using floating point. Also important to mention rounding at the end ensures we recover the exact integer value.

**Running time:** The outer loop runs n times while the inner one iterates over all neighbors of node i. So the running time of this algorithm is the work done in inner loop which is O(n power 2 ) times work of the outer loop namely O(n) (linear). The overall running time of this algorithm is then O(n power 3).

**4.Data structure analysis**

In this section we will try to describe and argument the reason why we choose the key data structures in each class. We will consider each class separately and for each class we will show the key data structures(as they are implemented in code) and then present the reason behind such selection.

**1.Adjecancy list**

This data structure appears in class Graph , and is expressed as: **List<List<Integer>> adj.** The main purpose of it is to represent the undirected graph of dynos (nodes) and bonds (edges). Adj can be thought as a list of lists. Each adj.get(u) holds a list of integers representing neighbors of node u. As we know to represent the graph we can use adjacency matrix or adjacency list. Adjacency list was preferred here because our graph is sparse , because there are constraints on the possible edges(bonds). Also it allows O(1) time access to a node's neighbors and O(V + E) traversal time which is ideal when we think we have to use DFS.

**2.** **Visited Array**

This data structure appears in class DFS and is expressed as: **boolean[] visited.** The main purpose of it is to track visited dynos(nodes) during DFS and ensure each component is explored exactly once. This is ideal because it takes a constant amount of time to check or mark the status visited. Also it allows us to avoid re-processing nodes and ensures each component is visited exactly once which is super important for us.

**3.Component Collection**

This data structure is part of class DFS and is expressed in this form: **List<Integer> component.** Its main purpose is to store nodes belonging to the same connected component discovered during DFS. We mentioned when we explained the DFS procedure that this was an addition when compared to the book version. We need it later to pass the full component to processComponent(). This allows us to compute costs and distinct cheapest configurations for each component.

**4.** **HashMap**

This data structure is used in buildLaplacian() method called from SpanningTreeCounter class expressed as **Map<Integer, Integer> index.** Its main purpose is to map actual dyno IDs to row/column indices in the Laplacian matrix. This is done because Laplacian matrices are dense and indexed from 0, requiring a consistent index mapping. HashMap offers constant access, crucial for performance during matrix construction. Using a HashMap here avoids expensive search operations and ensures that Laplacian rows/columns correctly correspond to dyno nodes in the component.

**5.** **2D Array for Laplacian Matrix**

This data structure is used in buildLaplacian() procedure called from SpanningTreeCounter class and is expressed as **long [][] L.**  It is used to represent the Laplacian matrix of each component because doing so is required by Kirchhoff’s Matrix-Tree Theorem to count spanning trees. We use long because it ensures correctness during integer addition/subtraction. Also predefined size makes long[][] efficient and straightforward to work with.

**6.** **2D Array for Laplacian Matrix**

This data structure is used in determinant() method and is called from SpanningTreeCounter class via SpanningTreeCounter.determinant. It is expressed as: **double [][] mat.** Its purpose is to compute the determinant of the reduced Laplacian matrix using Gaussian elimination. We use double because this floating-point representation avoids integer overflow during row operations. Although determinant computation is a cubic-time, it still is manageable for small components.