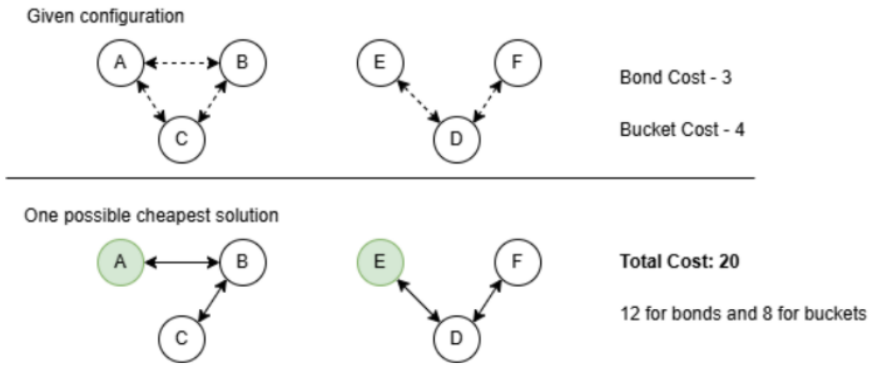
Big Weather Configuration - Documentation

Hesti Misiri, Endi Spahiu

May 2025

1 Introduction

The Java application, inside the **BigWeather** folder in this repository, has the purpose of processing a file containing the text representation of a *Grid Network* for a *Weather Forecasting Company.* The file contains information regarding *dynos* and their *bonds* with other *dynos.* A figure graph representation of this grid is listed below:



After processing said file, the application must output the cheapest possible configuration of the components in such a way where every *dyno* has a path towards a *bucket.* In addition, the application outputs another text file which contains a graph represented in text format that has the configuration; as well as, outputting the possible number of cheapest configurations (should there exist more than one).

The **GraphVisualisation** folder contains a python script which is used to visualise the exported graph of the cheapest configuration.

2 Data Structure Argumentation

2.1 Graph Data Structure

The repository contains two data structure files: *Graph.java* and *Grid.java.* Both of these structures are used to hold information which represent graphs, which include: *int verteces* which holds the number of vertexes, and *int edgeNum* which holds the number of edges. The *Grid* is an child sub-class of *Graph*, with the only added instance variables of *int bucketCost* and *int bondCost*, as per our parameter requirements.

We designate graph edges with an Adjacency List implementation is using a *LinkedList<Integer>[]* object, wherein the index value of the array is used to signify a vertex, and the *LinkedList<Integer>* object contains the other vertices it connects to. I much prefer this implementation of an *Adjacency List* as opposed to using a *Map<K, V>* object, as we already know how many vertices there are going to be when a *Graph* object is instantiated, and its a simpler implementation.

The only other method that these graph data structures have besides *getter* methods are an *addEdge(int u, int v)*, which should be self-explanatory.

2.2 Additional Classes for Graphs

Inside the **BigWeather** folder, you'll notice two additional classes called *Graphs.java* and *GraphReader.java*. The former class contains factory methods for the creation of directed and undirected graphs. In addition, it contains a public method which we can use to add a collection of strings as edges to a instantiated graph.

The latter class is used to read files that contain graph information in text format. After, it creates a new Graph (or Grid in our case) object which we can use. The text file format (for Grid objects) is as below:

```
6 5 4 3
1 2
1 3
4 5
4 6
2 3
```

The first line contains the number of vertices, the number of edges, the bucket cost, and the bond cost. Every other line below it contains the connections between vertexes.

3 Pseudo-Code Overview and Analysis

3.1 Main Methods

3.2 Depth First Search

We use a simple Depth First Search algorithm to find the *Connected Components* of the Grid, and then return them as a map after finishing. The time complexity of this algorithm is $O(n + e)$.

Algorithm 1 DFS(G)

```
Input: a graph G
Output: Collection of Connected Components
procedure DFS(G)
    for each node $v \in V$ do
        visited[v] = false
    end for
    for each node $v \in V$ do
        if not visited(u) then
            $CC \leftarrow (CCnum, newList)$            ▷ CC is a Dictionary Structure
            explore(G, v)
            CCnum++
        end if
    end for
end procedure
```

Algorithm 2 explore(G, v)

```
Input: a graph G
Output: Collection of Connected Components
procedure EXPLORE(G, v)
    visited[v] = true
    $CC[CCnum] \leftarrow v$
    for each edge $(v, u) \in E$ do
        if not visited(u) then
            explore(G, u)
        end if
    end for
end procedure
```

Algorithm 3 BFS(G, CC)

```
Input: a graph G
Output: a collection of Edges to form a Tree
procedure BFS(G)
    for all $u \in V$ do
        prev[v] = -1
    end for
    for all $L \in CC$ do
        if $L.size \leq 1$ then
            Continue
        end if
        if not visited(u) then
            $CC \leftarrow (CCnum, newList)$
            explore(G, v)
            CCnum++
        end if
    end for
end procedure
```

3.3 Breadth First Search

3.4 KirchhoffTheorem

3.5 Conclusion

4 Discussion of Proof

4.1 Finding the Minimum Cost

To prove our algorithm, we can use the definitions below:

Definition 1. *A component of an undirected graph is a connected sub-graph that is not part of any larger connected sub-graph.*

Definition 2. *A tree is an undirected acyclic graph, which has exactly one edge connecting any two vertices. A tree has $n - 1$ edges.*

We can suppose that for each component, there should exist only one bucket dyno to adhere to the configuration requirements. Then, we can also suppose that for each component, there must exist a tree which can be acquired by removing an edge until we reach $n - 1$. This is how we acquire our function:

$$Cost = BucketCost \cdot CCNum + \sum_{i=1}^{CCnum} BondCost \cdot n_i - 1 \tag{1}$$

There is also a special case for when the $BucketCost \leq BondCost$, wherein we can simply calculate the cost by multiplying the number of vertices of the graph with the *BucketCost*, as it would be cheaper (or simpler) to designate every dyno as a bucket.

$$Cost = BucketCost \cdot n \tag{2}$$

4.2 Finding the Number of Configurations

To find the number of configurations, we must first find the number of spanning trees per Connected Component of the graph. To do so, we may use Kirchhoff's Theorem, which is listed below:

Theorem 1. *For a given connected graph G with n labelled vertices, let* $\lambda_1, \lambda_2, ..., \lambda_{n1}$ *be the non-zero eigenvalues of its Laplacian matrix. Then the number of spanning trees of G is*

$$t(G) = \frac{1}{n}\lambda_1, \lambda_2, ..., \lambda_{n1} \tag{3}$$

The number that we acquire from the application of this theorem is then multiplied by the number of nodes of the Connected Component, to signify the changing position of buckets (we know from above that a Connected Component needs only one bucket to adhere to the requirements).

There are also two special cases for this solution: The first is when the $BucketCost < BondCost$, where there only one possible configuration. The second is when $BucketCost = BondCost$, wherein we must raise to the second power, the total number of configurations since for each additional bucket that we can add, we must remove one edge, and the position of buckets and removed edges can change by the total amount of configurations we've acquired by the application of our theorem.