**Algorithms & Complexity**

Assignment 1 - BigWeather cheapest configuration

Worked by: Parid Gjerasi and Altonel Alushi

**Table of Contents**

1. Introduction…………………………………..……………2

2. Data structures and their purpose…….…………………………………………………3

3. Pseudo Code used………………………………………………..4

4. Algorithm design and its implementation……………………………………………………6

5. Proof of correctness and mathematical foundation…………………………………………….7

6.Time and space complexity analysis………………………………………………………..8

7. References…………………………………………………………..9

**Introduction**

BigWeather operates a high-speed weather forecasting system on a distributed cluster of high-speed computing nodes called dynos. The dynos crunch huge volumes of sensor data stored in special cache units called buckets. The requirement of operation requires all the dynos to have uninterrupted access to bucket data, which they get either by direct mounting of the buckets on the dyno (at a flat rate per bucket) or by establishing network bonds with other dynos that have buckets (at bond connection charges). The system configuration is an optimization problem in which we must determine the least expensive configuration of buckets and bonds under physical network constraints that specify which dynos can bond.

It is a problem of a graph where dynos are nodes and possible bonds are edges. Our algorithm must traverse this graph and find configurations in which the total cost of bucket additions and bonding is low while every dyno can reach the bucket data directly or via connected paths. In this report, we provide our algorithm, code, and theoretical underpinnings for efficiently solving this optimization problem.

**Data structures and their uses**

The program employs a series of intricately crafted data structures to effectively address the dyno-bucket optimization problem. At its foundation, we created a bespoke MyList class that is fundamentally a dynamic array, with the particular application of holding adjacency lists for our graph representation. The custom solution was required to meet the assignment's prohibition on the use of Java's inherent collections. The MyList dynamically resizes itself using its resize() method, which doubles the capacity of the underlying array whenever needed, employing efficient use of memory while maintaining the time complexity for insertions at O(1) amortized.

In order to maintain a record of the connected components of the graph, the solution uses three parallel arrays: a two-dimensional components array in order to store dyno groupings, a componentSize array in order to store the number of dynos for each component, and a visited boolean array for bookkeeping in DFS traversal. This structure was favored over more complex structures because it enables direct, cache-friendly access to component information both during the DFS traversal phase and the cost calculation phase. The fixed array dimensions (1000×1000) are a reasonable limitation that could be enhanced in follow-on versions but are adequate for the problem as stated.

**Pseudo Code used**

**// Read input**

**function main:**

**read n, k, bucket\_cost, bond\_cost**

**build adjacency\_list from k bonds**

**// Case 1: One bucket per dyno**

**if bucket\_cost <= bond\_cost:**

**total\_cost = n \* bucket\_cost**

**total\_ways = 1**

**// Case 2: Use DFS to find components**

**else:**

**visited[1..n] = false**

**for i from 1 to n:**

**if not visited[i]:**

**dfs(i), count component**

**for each component of size s:**

**total\_cost = total\_cost + bucket\_cost + (s - 1) \* bond\_cost**

**total\_ways = total\_ways \* s**

**print total\_cost, total\_ways**

**visualize\_solution()**

**// DFS traversal**

**function dfs(node):**

**mark node as visited**

**add node to current component**

**for each neighbor of node:**

**if not visited[neighbor]:**

**dfs(neighbor)**

**// ASCII output**

**function visualize\_solution:**

**if bucket\_cost <= bond\_cost:**

**for i from 1 to n:**

**print "[i] (root)"**

**else:**

**for each component:**

**print first node as root**

**print remaining nodes indented under root**

**Algorithm design and its implementation**

The algorithm used takes a straightforward, three-step strategy that addresses the requirements of the problem in a methodical manner. During initialization, the program reads in and parses the input file, building an adjacency list representation of the dyno network. This step meticulously deals with the bond information, adding each bond in both directions to preserve the undirected property of the graph.

The solution's core is its use of depth-first search (DFS)[1] for connected component detection. The DFS algorithm was preferred because it operates in a simple manner and has a linear time complexity relative to the graph's size (O(n + k)). During graph exploration, the DFS visits dynos and marks them as well as completes the components array, effectively grouping dynos that can potentially share bucket access through bonds. This grouping is crucial for the subsequent cost optimization phase.

The cost calculation logic embodies the optimization insight of the algorithm. By comparing the bucketCost and bondCost arguments, the solution computes whether universal local bucket installation or shared bucket access through bonds is more cost-effective. The decision is made globally (when the costs are the same for all dynos) but applied at the component level to yield optimum resource allocation throughout the network.

**Proof of correctness and mathematical foundation**

The correctness of the algorithm follows from its consideration of all configuration possibilities along with optimal substructure criteria. We can use mathematical induction to prove that the solution always yields the minimum cost configuration. The base case (n=1) trivially only needs a bucket cost. For the inductive step, adding each new dyno either:

1. Connects to an existing component, adding only bond costs
2. Forms a new component, adding both bucket and bond costs.

The cost comparison logic guarantees that the cheaper option is always selected. The DFS traversal ensures complete component detection, while the configuration counting (for bonus points) leverages combinatorial mathematics to accurately enumerate all valid minimum-cost arrangements through multiplicative combination of component sizes.

**Time and space complexity analysis**

The solution demonstrates excellent computational efficiency with an overall time complexity of O(n + k), where n represents the number of dynos and k the number of bonds. This linear complexity arises from:

O(n) initialization of data structures

O(k) processing of all bonds

O(n + k) DFS traversal

O(n) cost calculation and output generation

Space complexity is O(n²) in the worst case due to the fixed-size component tracking arrays, though in practice it often performs better when component sizes are small. The custom MyList implementation contributes to memory efficiency by dynamically resizing only as needed.[2]

**References**

**[1] GeeksforGeeks. (n.d.). Connected Components in an Undirected Graph. Retrieved from** [**https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/**](https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/)

**[2] Sedgewick, R., & Wayne, K. *Algorithms* (4th ed.). Addison-Wesley, 2011.**[**https://algs4.cs.princeton.edu/home/**](https://algs4.cs.princeton.edu/home/)