Ambra Boci

BIGWEATHER OPTIMIZATION SOLUTION DOCUMENTATION

Introduction

This document presents detailed documentation for the BigWeather system optimization

solution. The main objective of this program is to determine the minimum cost configuration for the BigWeather forecasting system, ensuring that each compute engine (dyno) has access to a data cache

(bucket) either by hosting one directly or by connecting through network connections (bonds).

Solution Summary

This solution models the system as a graph, where:

1. Nodes represent dynos.
2. Edges represent allowed bonds between dynos.
3. Each node may host a bucket (with a fixed cost).
4. Each edge may be activated (with a bond cost).

The algorithm then:

Divides the graph into connected components using Breadth-First Search (BFS).

For each component, it evaluates four optimization strategies:

1. Assigning a bucket on every dyno.
2. Placing a central bucket and connecting other dynos with a Minimum Spanning Tree (MST).
3. Using a greedy dominating set to minimize the number of buckets while ensuring coverage.
4. When the bucket cost is lower than the bond cost, testing high-degree bucket placement to reduce bond usage.

Then it selects the cheapest strategy for each component and combines the results to calculate the overall minimum cost.

Additionally it counts and visualizes all distinct optimal configurations when multiple solutions are equally optimal.

Ambra Boci

Data structures used in the solution

1. Adjacency List

Used in: BuildGraph, FindConnectedComponents, FindMST, MinimumDominatingSet

Efficiently represents sparse graphs like the dyno-bond network. Provides O(1) access to a node's neighbor.

2. Set

Used in: FindConnectedComponents, MinimumDominatingSet, FindMST

Fast membership checking O(1) average time complexity. Used for tracking visited nodes during graph traversal and it will maintain the set of uncovered nodes in dominating set calculation.

3. Queue

Used in: FindConnectedComponents

Ensures all the nodes are processed during BFS, level by level. It will guarantee that all nodes at distance k from start are po=rocessed before nodes at distance k+1.

4. Priority Queue

Used in: FindMST

Supports efficient minimum extraction (O(log n)). Essentialfor Prim's algorithm to always select the next minimum-cost edge. It will make sure the spanning tree is constructed optimally by always choosing lowest cost edge.

5. List

Used in:

1. Storing bucket-hosting dynos
2. Maintaining separate connected components
3. Optimal configurations
4. Tracks selected bonds

Simple and flexible for ordered collections of dynos and bonds, and easy to iterate over for cost calculations. Provides O(1) access by index and efficient iteration.

6. Tuple

Used in: Representing bonds between dynos

Immutable and hashable. It's ideal for representing the fixed pair relashionship of a bond.Used in min() and max() functions to create a normalized representation.

Ambra Boci

Pseudocodes for the algorithm used in the solution

Finding Connected Components (BFS)

```
ALGORITHM: FindConnectedComponents(graph):
    visited ← empty set
    components ← empty list
    FOR dyno FROM 1 TO num_dynos:
        IF dyno NOT IN visited:
            component ← empty list
            queue ← new queue with dyno
            ADD dyno TO visited
            WHILE queue is not empty:
                current ← DEQUEUE from queue
                ADD current TO component
                FOR neighbor IN graph[current]:
                    IF neighbor NOT IN visited:
                        ADD neighbor TO visited
                        ENQUEUE neighbor
            ADD component TO components
    RETURN components
```

You treat each dyno and bond as part of a graph.

Use Breadth-First Search (BFS) to find all dynos that are connected (directly or indirectly).

This forms a connected component.

You repeat this for every dyno that hasn't already been visited.

Ambra Boci

Minimum Spanning Tree (Prim's Algorithm for Each Component)

```
ALGORITHM:  FindMST(component, graph, bond_cost):
    IF component is empty:
        RETURN empty list
    start ← component[0]
    visited ← set with start
    mst_edges ← empty list
    pq ← priority queue with (bond_cost, start, neighbor) for each neighbor of start
    WHILE pq is not empty AND size of visited < size of component:
        cost, frm, to ← EXTRACT-MIN from pq
        IF to NOT IN visited:
            ADD to TO visited
            edge ← (min(frm, to), max(frm, to))  # keep edge format consistent
            ADD edge TO mst_edges
            FOR neighbor IN graph[to]:
                IF neighbor NOT IN visited:
                    ADD (bond_cost, to, neighbor) TO pq

    RETURN mst_edges
```

To find the cheapest way to connect all dynos within a component using bonds.

Start with any dyno.

Always choose the cheapest bond (edge) that connects to a new dyno not already in the tree.

Use a priority queue (min-heap) to always select the smallest cost edge.

This helps reduce the number of buckets by connecting dynos to a central bucket host, minimizing bond costs.

Ambra Boci

Greedy Approximation for Minimum Dominating Set

```
ALGORITHM:  MinimumDominatingSet(component, graph, bucket_cost, bond_cost):
    uncovered ← set of all nodes in component
    bucket_dynos ← empty list
    bonds ← empty list
    WHILE uncovered is not empty:
        best_node ← null
        best_coverage ← -1
        FOR node IN component:
            coverage ← 1 IF node IN uncovered ELSE 0
            FOR neighbor IN graph[node]:
                IF neighbor IN uncovered:
                    coverage ← coverage + 1
            IF coverage > best_coverage:
                best_coverage ← coverage
                best_node ← node
        IF best_node is null:
            BREAK
        ADD best_node TO bucket_dynos
        REMOVE best_node FROM uncovered
        FOR neighbor IN graph[best_node]:
            REMOVE neighbor FROM uncovered
            IF neighbor IN component AND neighbor NOT IN bucket_dynos:
                bond ← (min(best_node, neighbor), max(best_node, neighbor))
                IF bond NOT IN bonds:
                    ADD bond TO bonds
    cost ← size of bucket_dynos × bucket_cost + size of bonds × bond_cost
    RETURN cost, bucket_dynos, bonds
```

Ambra Boci

Start with all dynos "uncovered."

Select the dyno that covers the most uncovered neighbors.

Place a bucket there.

Mark that dyno and its neighbors as covered.

Repeat until all dynos are covered

Ambra Boci

Component Optimization Strategy

```
ALGORITHM:  OptimizeComponent(component, graph, bucket_cost, bond_cost):
    all_buckets_cost ← length of component × bucket_cost
    min_cost ← all_buckets_cost
    bucket_dynos ← copy of component
    selected_bonds ← empty list
    # Strategy 2:
    mst_bonds ← FindMST(component, graph, bond_cost)
    FOR EACH central_bucket IN component:
        centralized_cost ← bucket_cost + length of mst_bonds × bond_cost
        IF centralized_cost < min_cost:
            min_cost ← centralized_cost
            bucket_dynos ← [central_bucket]
            selected_bonds ← mst_bonds
    # Strategy 3: Greedy dominating set
    dom_cost, dom_buckets, dom_bonds ← MinimumDominatingSet(component, graph, bucket_cost,
bond_cost)
    IF dom_cost < min_cost:
        min_cost ← dom_cost
        bucket_dynos ← dom_buckets
        selected_bonds ← dom_bonds


    # Strategy 4:
 If bucket_cost < bond_cost, try more bucket placements
    IF bucket_cost < bond_cost:
        [... strategic bucket placement logic ...]


    RETURN min_cost, bucket_dynos, selected_bonds
```

Ambra Boci

We are placing a bucket on every dyno (the worst-case scenario, with no bonds).

We are using one central bucket and connecting other dynos with MST bonds to share access.

We are applying the dominating set approach to reduce both the number of buckets and bonds.

If the bucket cost is less than the bond cost, we are also testing smarter bucket placements that use fewer bonds.

Finally, we are selecting the strategy that results in the lowest total cost.

Ambra Boci

Finding All Optimal Solutions

```
ALGORITHM : FindAllOptimalConfigurations(component, target_cost):
    optimal_configs ← empty list
    all_buckets_cost ← length of component × bucket_cost
    IF all_buckets_cost = target_cost:
        ADD (component.copy(), []) TO optimal_configs

    mst_bonds ← FindMST(component, graph, bond_cost)
    FOR EACH central_bucket IN component:
        centralized_cost ← bucket_cost + length of mst_bonds × bond_cost
        IF centralized_cost = target_cost:
            ADD ([central_bucket], mst_bonds) TO optimal_configs

    RETURN unique_configs
```

Try each strategy again (all buckets, MST, centralized buckets).

For every one that matches the minimum total cost, save it.

Try different combinations of bucket placements and bond arrangements that give the same total cost.

Eliminate duplicates.

Ambra Boci

Correctness Proof

1.FindConnectedComponents Algorithm

The FindConnectedComponents algorithm correctly identifies all connected components in the graph.

Base Case: Graph with 1 node. When the graph has only one node (dyno), the algorithm will:

1. Add the node to the visited set
2. Add the node to a new component list

Since there are no neighbors, the queue becomes empty

The component containing the single node is added to components

Return components containing one component with one node

This is correct because a graph with one node has exactly one connected component containing that node.

Inductive Hypothesis:

Assume the algorithm correctly identifies all connected components in any graph with k nodes, where k ≥ 1.

Inductive Step: Consider a graph G with k+1 nodes.

There are two cases to consider:

The (k+1)th node forms its own connected component

The (k+1)th node is connected to one or more of the first k nodes

Case 1: If the (k+1)th node forms its own connected component, then by the inductive hypothesis, the algorithm correctly identifies all connected components among the first k nodes. When it processes the (k+1)th node, it will identify it as a new unvisited node, perform BFS (which will only visit this single node since it's not connected to others), and add a new component containing only this node to the components list. Thus, all connected components are correctly identified.

Case 2: If the (k+1)th node is connected to some of the first k nodes, let C be the connected component among the first k nodes that the (k+1)th node connects to. By the inductive hypothesis, the algorithm correctly identifies all connected components among the first k nodes.

Now, if the algorithm first processes a node in C, the BFS will also visit the (k+1)th node (since it's connected to C), and it will be included in the component C.

If the algorithm first processes the (k+1)th node, the BFS will visit all nodes in C (since they're connected to the (k+1)th node), and C plus the (k+1)th node will form a component.

In either case, the algorithm correctly identifies the connected component containing the (k+1)th node and all connected components in the graph.

At the beginning of each iteration of the outer FOR loop, all nodes that have been visited so far have been correctly assigned to their respective connected components.

This invariant is maintained because:

1. Initially, no nodes have been visited, and components is empty (trivially true).
2. In each iteration, if a node has not been visited, the BFS starting from that node visits exactly all nodes in its connected component.
3. Each node is visited exactly once due to the visited set.
4. At the end of the iteration, all visited nodes have been assigned to their components.

Therefore, by induction, the FindConnectedComponents algorithm correctly identifies all connected components in any graph.

2.FindMST Algorithm (Prim's Algorithm) – Correctness Proof

The FindMST algorithm correctly computes a minimum spanning tree for the given connected component.

Base Case: |visited| = 1.
Initially, visited contains only the start node. Since a minimum spanning tree for a graph with one node has no edges, mst_edges is correctly initialized as empty.

Inductive Hypothesis:
Assume that after k iterations of the WHILE loop ($1 \leq k <$ |component|), the edges in mst_edges form a minimum spanning tree for the nodes in visited, and |visited| = k+1.

Inductive Step:
Let's consider the (k+1)th iteration of the WHILE loop.

Let T_k be the tree formed by the edges in mst_edges after k iterations. By the inductive hypothesis, T_k is a minimum spanning tree for the nodes in visited.

In the (k+1)th iteration, the algorithm:

1. Extracts the minimum weight edge (frm, to) from the priority queue where frm is in visited and to is not
2. Adds to to visited
3. Adds edge (frm, to) to mst_edges
4. Adds edges from to to unvisited neighbors to the priority queue

Let's define a cut (visited, V–visited) in the graph, where V is the set of all nodes in the component.
The algorithm selects the minimum weight edge (frm, to) crossing this cut.

There MUST exists a minimum spanning tree of the entire graph that includes the minimum weight edge crossing any cut.

Proof:
Suppose e = (u, v) is the minimum weight edge crossing a cut (S, V–S), and T is a minimum spanning tree that does not include e.
Adding e to T creates a cycle. This cycle must contain another edge e′ that crosses the same cut (S, V–S).
Removing e′ and adding e results in a spanning tree T′ with weight w(T′) = w(T) – w(e′) + w(e).
Since e is the minimum weight edge crossing the cut, w(e) ≤ w(e′), which means w(T′) ≤ w(T).

Ambra Boci

Since T is a minimum spanning tree, $w(T') = w(T)$, and T' is also a minimum spanning tree.
Therefore, there exists a minimum spanning tree that includes e.

By this claim, there exists a minimum spanning tree of the entire graph that includes the edge (frm, to) selected by the algorithm.
Since T_k is a minimum spanning tree for the nodes in visited, and (frm, to) is the minimum weight edge connecting a node in visited to a node outside visited,
$T_{k+1} = T_k \cup \{(\text{frm}, \text{to})\}$ is a minimum spanning tree for the new set of visited nodes.

Termination**:**
The WHILE loop terminates when either the priority queue is empty or all nodes in the component have been visited.
Since the component is connected, there is always an edge crossing the cut (visited, V–visited) until all nodes are visited.
Therefore, the algorithm will visit all nodes and terminate with a minimum spanning tree for the entire component.

Therefore, by induction, the FindMST algorithm correctly computes a minimum spanning tree for the given connected component.

3.MinimumDominatingSet Algorithm
The MinimumDominatingSet algorithm produces a valid dominating set where every node either hosts a bucket or is connected via a bond to a node hosting a bucket.

Proof:

We will prove this using loop invariants and induction on the number of iterations of the WHILE loop.

Invariant 1:
At the start of each iteration of the WHILE loop, uncovered contains exactly those nodes that are neither in bucket_dynos nor adjacent to a node in bucket_dynos.

Invariant 2:
At the start of each iteration of the WHILE loop, bonds contains exactly the edges connecting nodes not in bucket_dynos to nodes in bucket_dynos.

Base Case**:** Before the first iteration of the WHILE loop:

- uncovered contains all nodes in the component
- bucket_dynos is empty
- bonds is empty
  Both invariants hold trivially.

Inductive Hypothesis:
Assume that after *k* iterations of the WHILE loop, both invariants hold.

Inductive Step:
Consider the (*k*+1)th iteration of the WHILE loop.

In this iteration, the algorithm:

1. Selects best_node with the highest coverage (number of uncovered nodes it can cover)

2. Adds best_node to bucket_dynos
3. Removes best_node from uncovered
4. For each neighbor of best_node:
   a. Removes the neighbor from uncovered
   b. If the neighbor is not in bucket_dynos, adds a bond between best_node and the neighbor

After these steps:

- best_node is in bucket_dynos and not in uncovered (maintaining Invariant 1)
- All neighbors of best_node are removed from uncovered (maintaining Invariant 1)
- Bonds are added between best_node and its neighbors that are not in bucket_dynos (maintaining Invariant 2)

Therefore, both invariants are maintained after the (k+1)th iteration.

<u>Termination:</u>
The algorithm terminates when uncovered is empty.
By Invariant 1, this means all nodes are either in bucket_dynos or adjacent to a node in bucket_dynos, which is the definition of a dominating set.
By Invariant 2, bonds contains exactly the edges connecting nodes not in bucket_dynos to nodes in bucket_dynos, ensuring that every node not hosting a bucket is connected to a node hosting a bucket.

Therefore, the algorithm produces a valid dominating set where every node either hosts a bucket or is connected via a bond to a node hosting a bucket.

4.OptimizeComponent Algorithm

The OptimizeComponent algorithm correctly finds the minimum cost configuration among the strategies it considers, ensuring that every dyno either hosts a bucket or has access to a bucket through bonds.

<u>Strategy 1:</u>

In this strategy, every node in the component hosts a bucket.
Therefore, every dyno has direct access to a bucket, making this a valid configuration.

<u>Strategy 2 (MST-Based):</u>

In this strategy:

1. One node (central_bucket) hosts a bucket
2. All other nodes are connected to central_bucket via the minimum spanning tree edges (mst_bonds)

Since a spanning tree by definition connects all nodes in the component, every node not hosting a bucket has a path to the central_bucket through the spanning tree edges.
Given that bonds are bidirectional, this ensures every dyno has access to a bucket, making this a valid configuration.

<u>Strategy 3 (Dominating Set):</u>

By the correctness of the MinimumDominatingSet algorithm (proved earlier), this strategy produces a dominating set (dom_buckets) where every node either hosts a bucket or is adjacent to a node hosting a bucket. The bonds (dom_bonds) connect nodes not hosting buckets to nodes hosting buckets.
Therefore, every dyno has access to a bucket, making this a valid configuration.

Strategy 4 (Bucket-Focused):

This strategy applies when bucket_cost < bond_cost, and it focuses on strategic bucket placement to minimize the number of bonds needed.
Without going into the detailed implementation, we can assert that any correct implementation of this strategy must ensure that every dyno has access to a bucket, making it a valid configuration.

Selection of Minimum Cost Strategy:

The algorithm initializes min_cost with the cost of Strategy 1.
For each subsequent strategy, it calculates the cost and updates min_cost, bucket_dynos, and selected_bonds if the new strategy has a lower cost.
After considering all strategies, min_cost will contain the minimum cost among all strategies, and bucket_dynos and selected_bonds will represent the corresponding configuration.

5. FindAllOptimalConfigurations Algorithm
The FindAllOptimalConfigurations algorithm correctly identifies all distinct optimal solutions for a connected component with the given target cost, among the configurations it considers.

The algorithm considers three types of configurations:

1. All-buckets configuration: Every node hosts a bucket, no bonds
2. MST-based configurations: One node hosts a bucket, connected to others via MST
3. Various bucket combinations: Different numbers of buckets placed on different nodes

.For each type of configuration, the algorithm:

1. Calculates the cost of the configuration
2. Compares it with the target cost
3. Adds the configuration to optimal_configs if the costs match

This correctly identifies configurations that have the target cost.

Bucket Combinations Generation:

The GenerateBucketCombinations function recursively generates all combinations of placing buckets on different sets of nodes. For each combination, it:

1. Calculates the necessary bonds
2. Computes the cost
3. Adds the configuration to optimal_configs if the cost matches the target cost

Ambra Boci

Removal of Duplicates:

Two configurations are considered the same if they have the same set of bucket dynos and the same set of bonds. The algorithm:

1. Iterates through all configurations in optimal_configs
2. For each configuration, checks if it's the same as any configuration already in unique_configs
3. Adds the configuration to unique_configs only if it's unique

This ensures that only distinct configurations are counted.

Ambra Boci

Time complexity analysis

1. Building the Graph

Complexity: $O(V + E)$

Constructing the graph involves creating an adjacency list for each dyno (node) and inserting each bond (edge) into the list. This operation is linear in the number of dynos and bonds.

2. Finding Connected Components

Complexity: $O(V + E)$

This step uses Breadth-First Search (BFS) to identify all connected components in the graph. Each node and edge is visited exactly once, leading to linear time complexity.

3. Minimum Spanning Tree (MST) Construction

With Priority Queue: $O(E \log V)$

With Simple Implementation: $O(V^2)$

MST is computed for each component using Prim's Algorithm. If a priority queue is used, it efficiently retrieves the minimum cost edge. The simpler implementation (using nested loops) is less efficient but easier to implement.

4. Minimum Dominating Set (Greedy Approximation)

Complexity: $O(V^2)$

The greedy approach selects dynos that cover the most uncovered neighbors. In the worst case, every node is checked against every other node, leading to quadratic time.

5. Component Optimization

Complexity: $O(V^2)$

For each component, multiple strategies are evaluated:

- All dynos hosting buckets
- One central bucket with MST
- Greedy dominating set
- Special case for low bucket costs

Each strategy may include full passes over all nodes, hence a complexity of $O(V^2)$ per component.

6. Finding All Optimal Configurations

Complexity: O(2^V) (Potentially exponential)

Enumerating all possible bucket placements and bond combinations that result in the same minimum cost can grow exponentially with the number of dynos.

OVERALL TIME COMPLEXITY:

$O(C \times (V^2 + E \log V))$

Ambra Boci

References

1.Connected components in undirected graphs.

Dasgupta et al., Section 3.2 (pages 89–91) Chapter 3 – Decompositions of Graphs, Section 3.2 (Depth-first search in undirected graphs).

2. Finding MST using greedy algorithms like Prim's and Kruskal's.

Dasgupta et al., Section 5.3 (pages 131–134) Chapter 5 – Minimum Spanning Trees.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. Section 23.2: The algorithms of Kruskal and Prim.

3. Priority Queue ,Data structure supporting efficient min/max operations.

Dasgupta et al., Section 5.3 Discussed within Prim's algorithm in Chapter 5.

4. Minimum Dominating Set

NP-complete problem where a minimal subset of vertices dominates the graph.

Dasgupta et al., Section 8.2 ,Chapter 8 – NP-Complete Problems

5. Greedy Approximation for Dominating Set

Dasgupta et al., Section 9.2 Chapter 9 – Approximation Algorithms.

Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson. Chapter 4: Greedy Algorithms

6. Component-Based Decomposition

Dasgupta et al., Section 3.2 (pages 89–91) Connected Components.

7. Multi-Strategy Optimization

Dasgupta et al., Chapters 8 and 9 (general strategy discussion)