BigWeather Report

Algorithms and Complexity Assignment

by Frenki Selmani, Kristi Shehaj

1. Introduction

The goal of this project was to solve a resource allocation problem for BigWeather, a weather forecasting company that needs to ensure all compute units (dynos) have access to a data bucket. A dyno can either host a bucket or connect to another dyno via a bond to access its bucket. Each bucket and bond has a cost, and the objective is to minimize the total cost across the system.

Initial solutions assumed full connectivity and were later found not suitable for general graphs. A brute-force approach was implemented later on to correctly handle arbitrary graph topologies and meet all requirements, including the bonus features.

2. Data Structures Used

2.1 Graph (Adjacency List)

Implemented using a list of lists (self.adj), where adj[i] contains all neighbors of node i. This structure enables efficient traversal and is optimal for sparse graphs.

2.2 Queue (Custom Implementation)

Used in BFS to find connected components. Implemented as a list with an index pointer to simulate efficient dequeuing.

These data structures ensure clarity and efficiency while obeying the constraint of not using built-in libraries for core logic.

3. Algorithms

The core logic of the program revolves around two primary computational processes:

3.1 Component Detection (BFS)

To efficiently divide the graph into manageable subproblems, a breadth-first search (BFS) algorithm is used to extract connected components. This is essential because the bucket assignment problem can be solved independently within each component.

3.2 Bucket Placement via Brute-Force

For each component, the brute-force approach systematically evaluates all possible subsets of nodes that could host a bucket. It verifies whether each subset allows every dyno in the component to access a bucket, either directly or via a bond to a host. Among all valid configurations, it selects the one(s) with the minimum total cost.

This method, although exponential in the number of nodes per component, guarantees optimality and correctness, making it appropriate for relatively small components often found in sparse graphs.

4. Pseudocode for Algorithms

4.1 BFS to Identify Components

```
BFS(start):
    Initialize empty queue and add start node
    Mark start as visited
    Initialize component list
    While queue not empty:
        Dequeue node and add to component list
        For each neighbor of node:
            If not visited:
                Mark as visited
     Return the component list
```

4.2 Brute-Force Bucket Assignment

```
function count_brute_force(component):
    n = len(component)
    min_cost = ∞
    count = 0
    best_hosts = []

    for each non-empty subset of component as hosts:
        if not is_valid(hosts):
            continue

        bond_count = 0
        for each node in component:
            if node not in hosts:
                if there exists a neighbor that is a hosts:
                    bond_count += 1

        cost ← |hosts| × bucket_cost + bond_count × bond_cost

        if cost < min_cost:
            min_cost ← cost
            count ← 1
            best_config ← hosts
        else if cost == min_cost:
            count += 1

        return (min_cost, count, best_config)
```

4.3 Validity Check for a Configuration

```
function is_valid(hosts):
    for each node in component:
        if node is a host: (skip)
            if no neighbor of node is in hosts:
                return False
     return True
```

5. Correctness Argument

- **Component Detection**: BFS correctly marks all nodes reachable from the starting node, forming a connected component.
- **Validity**: Every brute-force configuration is tested against the condition that all nodes must have access to and are neighbors to a bucket.
- **Exhaustiveness**: All possible subsets of hosts are tested, guaranteeing the globally optimal cost is found.
- **Verification**: Correct results observed on edge cases (line graphs, trees, cycles, and stars), proving generality.

Thus, the brute-force approach is correct for any graph input.

6. Time Complexity Analysis

Greedy Heuristic (Prototype)

The greedy method tries to place one bucket in a node that connects to all others in a component (like in a star graph).

- For each component with c nodes:
  - It checks each node to see if it connects to all others → $O(c^2)$
- For the whole graph:
  - Graph creation and BFS: $O(n + k)$
  - Greedy checks: up to $O(n^2)$ in worst case

Pros: Fast on simple graphs like stars

Cons: Not guaranteed to find the correct or cheapest solution for all graphs

Brute Force Algorithm (Final Submission)

This method tries every possible combination of bucket placements in each component.

- For a component with c nodes:
- There are 2^c possible subsets to try
- For each subset:
- Check if it's valid ($O(c^2)$)
- Count bonds and cost ($O(c)$)

So, total time per component: $O(2^c \times c^2)$

- Graph building and BFS: $O(n + k)$
- Slower on large components, but works for any graph

Pros: Always finds the correct, cheapest solution

Cons: Slower on large components (but acceptable for small to medium graphs)

7. The Process

1)The first step was creating the Graph Class. Then, self.adj was defined, which is a list of lists representing adjacent to the node.

```python
#Graph Class
class Graph:
    def __init__(self, n):
        self.n = n
        self.adj = [[] for i in range(n + 1)]

    def add_pair(self, u, v):
        self.adj[u].append(v)
        self.adj[v].append(u)

    def get_neighbors(self, node):
        return self.adj[node]
```

2)Then Queue Class was created. We defined self.items (a list to store items in the queue) and self.start (a start index for dequeueing).

```python
# Queue Class
class Queue:
    def __init__(self):
        self.items = []
        self.start = 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.start < len(self.items):
            item = self.items[self.start]
            self.start += 1
            return item
        return None

    def is_empty(self):
        return self.start >= len(self.items)
```

3)We set up a way to get inputs from a text file. “lines” reads lines from the file and remove empty ones. “Map” turns the stings into integers. “Graph.add_pair” adds the pairs to the already initialized graph. Then, “visited” creates a list of visited nodes where all are initially false.

```
# Read input
file_path = input("Enter absolute path to input file: ")
with open(file_path, 'r') as file:
    lines = [line.strip() for line in file.readlines() if line.strip()]

n, k, bucket_cost, bond_cost = map(int, lines[0].split())
graph = Graph(n) h

for line in lines[1:]:
    u, v = map(int, line.split())
    graph.add_pair(u, v)

visited = [False] * (n + 1)
```

4)Implementation of **BFS** (*Breadth First Search*) to find connected components. “Component” is a list to store the component nodes. “Visited[start]” marks the start of the node as visited. The **while** loop dequeues nodes until the queue is empty. Furthermore, “component.append(node)” adds the node to the component list. The **for** loop gets the neighbors of the current node, and then depending on whether the node is visited or not, the loop takes it into account.

```
# BFS
def bfs(start): # BFS function to find connected components
    queue = Queue()
    queue.enqueue(start)
    component = [] # List to store the component nodes
    visited[start] = True # Mark the start node as visited

    while not queue.is_empty(): # Dequeue nodes until the queue is empty
        node = queue.dequeue()
        component.append(node) # Add the node to the component list

        for neighbor in graph.get_neighbors(node): # Get neighbors of the current node
            if not visited[neighbor]:   # If the neighbor is not visited
                visited[neighbor] = True #  Mark it as visited
                queue.enqueue(neighbor)  # Enqueue the neighbor for further exploration
    return component
```

5)Implementation of Brute-force function to find the minimum cost and configurations. Is_valid checks whether the selected hosts are valid. Then, the list is converted to set for a faster lookup. The for loop checks each node in the component, and if the node is a host, we skip it. If no neighbor is a host, it is marked as invalid.

```python
# Brute-force
def count_brute_force(component, graph, bucket_cost, bond_cost):
    def is_valid(hosts): # Check if the selected hosts are valid
        host_set = set(hosts) # Convert list to set for faster lookup
        for node in component: # Check each node in the component
            if node in host_set: # If the node is a host, skip it
                continue
            neighbors = graph.get_neighbors(node)
            if not any(neigh in host_set for neigh in neighbors): # If no neighbor is a host, it's invalid
                return False
        return True

    n = len(component) # Number of nodes in the component
    min_cost = float('inf') # Initialize minimum cost to infinity as default vaule
    count = 0 # Count of configurations
    best_hosts = [] # List to store the best hosts
```

The for loop below iterates using bitmasking. It checks each beat and if the j-th bit is set, it is included in the corresponding code. Then, the “if not”, checks whether the selected hosts aren’t valid, with the intention of skipping them.

Furthermore, bond_count is initialized in order to calculate the cost. If the cost is less than the minimum, we update it and reset the count to 1, while also updating the best hosts. But if the cost is equal to the minimum, we increment the count. If no proper configuration is found, return the cost of using only buckets.

```python
    for i in range(1, 2 ** n): #Iterates using bitmasking
        hosts = []
        for j in range(n): # Check each bit
            if (i >> j) & 1: # If the j-th bit is set, include the corresponding node
                hosts.append(component[j])

        if not is_valid(hosts): # If the selected hosts are not valid, skip
            continue

        bond_count = 0 #Initialize bond count in oreder to calculate the cost
        for node in component:
            if node in hosts:
                continue
            for neigh in graph.get_neighbors(node):
                if neigh in hosts: # If the neighbor is a host, count it as a bond
                    bond_count += 1
                    break

        cost = len(hosts) * bucket_cost + bond_count * bond_cost # Calculate the total cost

        if cost < min_cost: # If the cost is less than the minimum cost found so far
            min_cost = cost # Update the minimum cost
            count = 1 # Reset the count to 1
            best_hosts = hosts # Update the best hosts
        elif cost == min_cost: # If the cost is equal to the minimum cost
            count += 1 # Increment the count

    if count == 0: # If no valid configuration is found, return the cost of using only buckets
        return len(component) * bucket_cost
    return min_cost, count, best_hosts
```

8. Algorithm Design Evolution

The development of this solution followed a progressive exploration of possible strategies, each one aimed at minimizing the total configuration cost while maintaining correctness.

Phase 1: Greedy Heuristic (Prototype)

Initially, we used a greedy strategy where we attempted to assign a single bucket node to each component. The remaining nodes were then connected to said bucket node via bonds. This approach was effective in providing the required output.

Phase 2: Optimized Heuristic

We extended the greedy method to consider the Bonus Considerations. It now checks for valid bucket placements, gives the count of valid configurations and also a visualization of one of minimal cost configurations.

However, during testing we found out that the algorithm only works with complete/star graphs since the logic assumes that all nodes are neighbors/connected.

Phase 3: Brute-Force Algorithm (Final Version/Stage)

Ultimately, we adopted a brute-force approach to ensure correctness. It exhaustively checks all possible subsets of bucket placements within each component, verifying that every node is either a host or connected to a host. Although exponential in nature, it guarantees that the optimal configuration is found.

9. Conclusion

The final brute-force implementation meets all problem and bonus requirements:

- Calculates the minimal total cost
- Counts the number of cheapest configurations
- Visualizes one valid optimal solution

Early prototype versions explored simple assumptions but were replaced to handle other types of graphs.