Documentation for Karger-Stein Algorithm Implementation

Rameez Wasif
Zain Hatim

Overview

This document provides detailed documentation for the Python implementation of the Karger-Stein algorithm for computing minimum k-cuts in weighted graphs.

Key Components

1. Brute-Force λ_k Estimation

Function: `_brute_force_lambda_k()`
Computes the exact minimum k-cut weight by evaluating all possible k-partitions. This is feasible for small graphs only.

2. Partition Generation

Function: `_generate_k_partitions(n, k)`
Recursively generates all possible k-partitions of a set of n nodes.

3. Minimum 2-Cut Estimation

Function: `_min_cut_weight()`
Uses basic Karger's algorithm to estimate the minimum 2-cut weight by repeated contraction and trial sampling.

4. Recursive Cut with Sparsification

Function: `find_min_k_cut_recursive()`
Runs the Karger-Stein algorithm with optional Nagamochi-Ibaraki sparsification to reduce edge count.

5. Sparsification

Function: `_nagamochi_ibaraki_sparsify()`
Preserves small cuts by iteratively constructing a sparse subgraph using max spanning forests.

6. Connectivity Computation

Function: `_compute_max_connectivity()`
Estimates maximum edge connectivity using exact computation (for small graphs) or random sampling (for larger graphs).

7. Cut Preservation Check

Function: `_preserve_min_k_cut()`
Checks whether the sparsified graph retains the minimum k-cut by comparing sampled cuts between original and sparsified graphs.

8. Exact Minimum k-Cut (for Validation)

Function: `_compute_min_k_cut()`
Brute-force approach for small graphs to validate k-cut correctness.

9. Random Partition Generator

Function: `_generate_random_partition()`
Randomly assigns nodes into k partitions for heuristic-based testing and preservation validation.

10. Refined λ_k Estimation

Function: `_estimate_lambda_k_from_trials()`
Improved estimation based on multiple observed cut weights from sampled trials. Computes average and variance, logs them.

Code Snippets

Code for all methods is implemented using Python 3 and makes use of NetworkX for graph operations. Below is an excerpt showing the start of the implementation:

```
def _brute_force_lambda_k(self) -> float:
    from itertools import combinations
    n = self.original_graph.number_of_nodes()
    min_weight = float('inf')
    for partition in self._generate_k_partitions(n, self.k):
        cut_weight = self._calculate_cut_weight(self.original_graph, partition)
        if cut_weight < min_weight:
            min_weight = cut_weight
    return min_weight
```

Listing 1: Sample Function - Brute Force Lambda k

Dependencies

- Python 3.7+
- NetworkX
- Numpy
- itertools (standard library)
- custom `contract_edge` function
- custom `PerformanceLogger`