Karger's Algorithm for Minimum Cut Problem

Students Charels Hugo
Course INFO-F413
Instructors Jean Cardinal

Abstract

This report explores Karger's algorithms for the minimum cut problem, focusing on the Contract and FastCut variations. Implementations are presented, success probabilities evaluated under constrained time budgets, and theoretical results verified through experiments. Complete and planar graphs are analyzed due to their contrasting topological properties and importance in graph algorithm studies. Experimental findings confirm theoretical predictions and provide insights into algorithm performance and efficiency.

1 Introduction

The minimum cut problem involves partitioning the vertex set of a graph $G = (V, E)$ into two non-empty subsets such that the number of edges crossing the subsets is minimized. Karger's randomized contraction algorithm offers an innovative probabilistic solution, combining simplicity with strong theoretical guarantees.

This study focuses on implementing and comparing the Contract and FastCut algorithms, with particular attention to verifying the following theorem:

Theorem 1 (FastCut Success Probability)**.** *The FastCut algorithm succeeds in finding a minimum cut with probability* $\Omega(1/\log n)$.

Complete and planar graphs are chosen as test cases due to their theoretical and practical significance. Complete graphs, characterized by maximum edge density, provide a challenging scenario for edge-based algorithms, while planar graphs, constrained by structural properties, often exhibit unique algorithmic behavior. By examining these graph families, we aim to connect theoretical insights with empirical performance.

2 Algorithm Description

2.1 The Contract Algorithm

The Contract algorithm iteratively contracts randomly chosen edges until only two vertices remain. The resulting cut corresponds to the partition of these vertices. The algorithm's success probability is at least $\frac{2}{n(n-1)}$ for an input graph with n vertices.

Implementation The algorithm is implemented in Rust (see Appendix A.1).

2.2 The FastCut Algorithm

FastCut enhances the Contract algorithm by recursively applying it to smaller subgraphs. The steps are as follows:

1. If $n \leq 6$, compute the minimum cut via brute force.
2. Otherwise, set $t = \lceil 1 + n/\sqrt{2} \rceil$.
3. Perform two independent Contract algorithm until t vertices remain, yielding subgraphs H_1 and H_2.
4. Recursively compute the minimum cuts of H_1 and H_2.
5. Return the smaller of the two cuts.

FastCut improves the success probability to $\Omega(1/\log n)$ while maintaining a runtime complexity of $O(n^2 \log n)$.

Implementation The algorithm is implemented in Rust (see Appendix A.2).

3 Experimental Plan

Our experiments focus on complete and planar graphs to evaluate the performance of Karger's algorithms. For planar graphs, instances were constructed with the maximum number of edges, $|E| = 3|V| - 6$, as dictated by Euler's formula, ensuring challenging test cases.

The experimental goals are:

1. Verify theoretical success probability bounds:
   (a) Run each algorithm 100 times on graphs of varying sizes.
   (b) Compare observed success rates with theoretical bounds.
2. Compare performance under equal time constraints:
   (a) Measure the runtime of a single FastCut execution.
   (b) Run Contract repeatedly within the same time frame.
   (c) Compare the success rates of both approaches.

Metrics include success probability, runtime, and the impact of graph topology on performance. Each experiment is repeated 100 times for statistical robustness.

4 Experimental Results

4.1 Verification of Probability Bounds

For complete graphs, FastCut consistently achieved a 100% success rate, while the Contract algorithm demonstrated slightly lower probabilities, as shown in Figure 1.

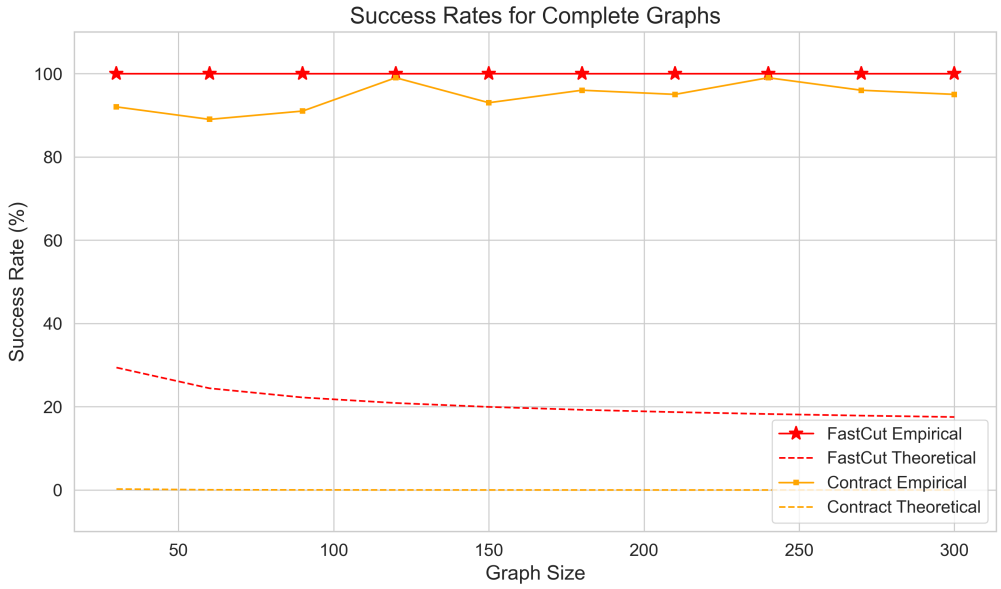

Figure 1: Success probability of FastCut and Contract algorithms on complete graphs.

For planar graphs, FastCut maintained near-perfect performance, with success rates close to 100% in all trials. In contrast, Contract often fell below 40%, as depicted in Figure 2.

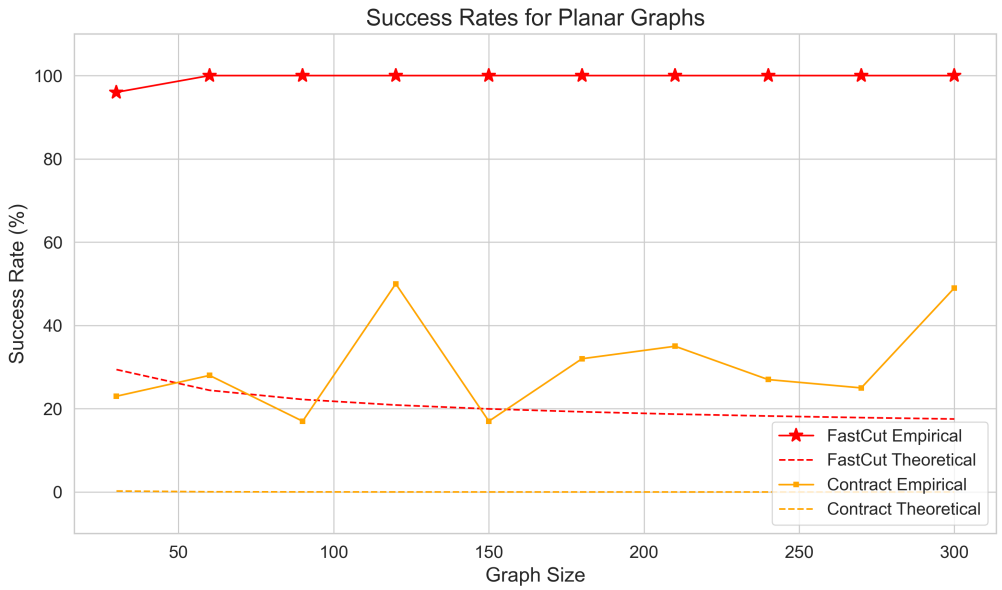

Figure 2: Success probability of FastCut and Contract algorithms on planar graphs.

4.2 Performance Under Equal Time Constraints

Under identical time budgets, FastCut achieved a 100% success rate on complete graphs in a single run, while Contract required multiple runs to reach comparable rates (Figure 3).

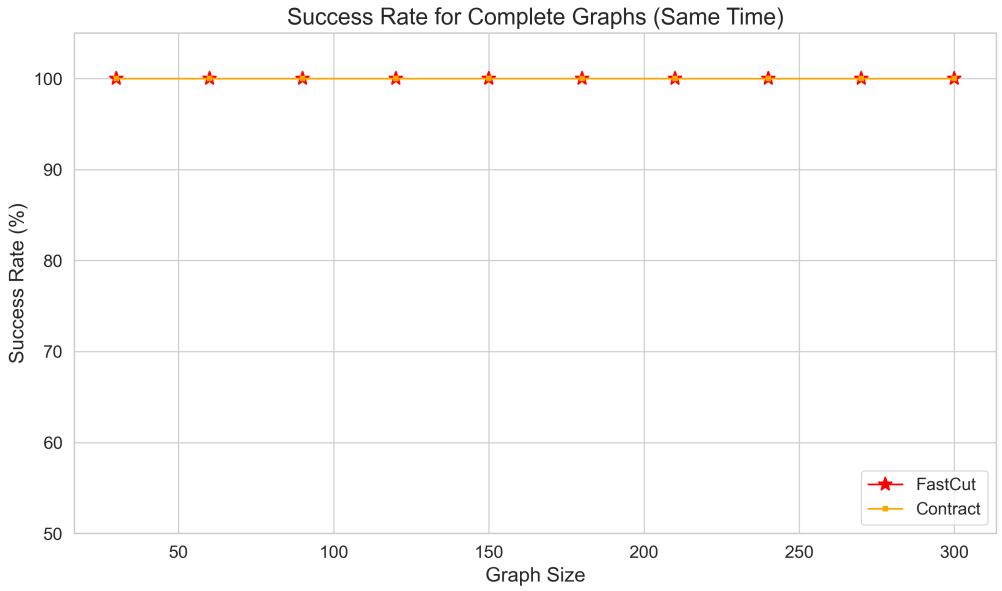

Figure 3: Comparison of success rates for FastCut and Contract on complete graphs under equal time constraints.

For planar graphs, Contract consistently achieved 100% success rates, while FastCut remained above 90% (Figure 4).

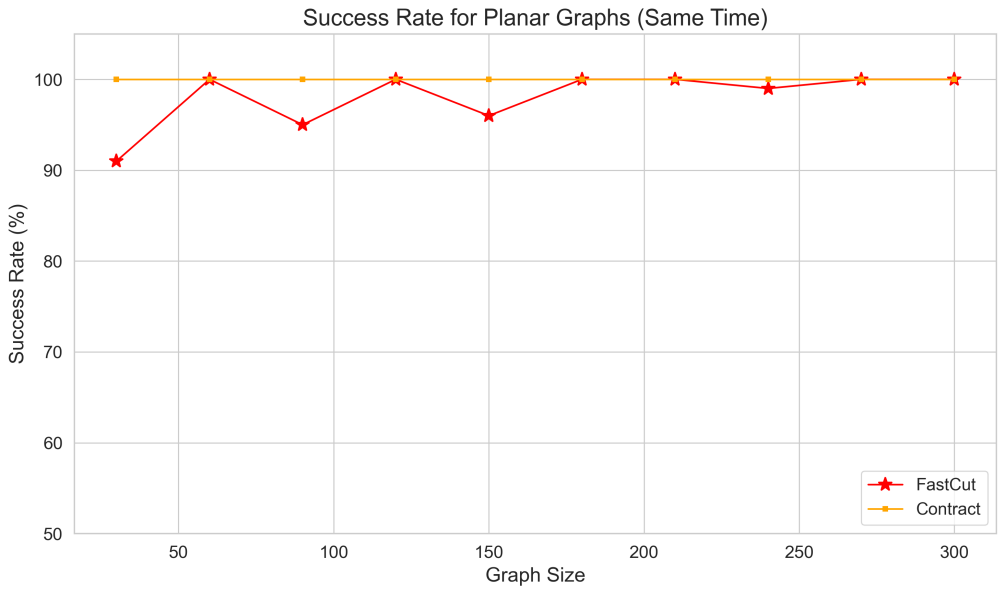

Figure 4: Comparison of success rates for FastCut and Contract on planar graphs under equal time constraints.

5 Discussion

The higher success rate of the Contract algorithm on complete graphs can be attributed to their dense edge structure. With $\binom{n}{2}$ edges, the likelihood of contracting a non-minimum cut edge is lower, enhancing the probability of preserving a minimum cut. Conversely, planar graphs, with their edge count limited by Euler's formula, increase the risk of contracting a minimum cut edge, reducing success rates.

Comparisons under equal time constraints highlight the tradeoff between algorithm complexity and runtime efficiency. FastCut's recursive nature ensures high single-run success probabilities but limits the number of runs possible within a fixed time. Repeated runs of Contract compensate for its lower single-run probability, achieving higher aggregate success rates.

6 Conclusion

This study evaluated Karger's Contract and FastCut algorithms on complete and planar graphs, validating theoretical predictions with empirical data. Results demonstrate that Contract performs better on complete graphs due to their dense edge structure, while planar graphs pose challenges due to sparsity. FastCut, though slower, consistently achieves high success rates.

For future work, we propose extending this analysis to other graph classes, and experimenting with larger graph sizes. These directions could provide deeper insights into the scalability and adaptability of Karger's algorithms. Unfortunately, the scope of this study was limited by the computational resources and time available, which constrained the size and variety of graphs that could be tested.

Disclaimer

This document was co-written with the assistance of generative AI, primarily for improving grammar, style, and clarity. The ideas, results, and analyses are the work of the authors.

A Source Code

A.1 Contract

```rust
fn contract(mut graph: impl UnMulGraph) -> usize {
        while graph.len_vertices() > 2 {
                let (u, v) = graph.get_random_edge();
                graph.contract_edge(u, v);
        }
        graph.len_edges()
}
```

A.2 FastCut

```rust
fn fast_cut(graph: impl UnMulGraph + Clone) -> usize {
        if graph.len_vertices() <= 6 {
                min_cut(graph)
        } else {
                let t = (1.0 + graph.len_vertices() as f64 / 2.0_f64.sqrt()).ceil() as usize;
                let g1 = contract_t(graph.clone(), t);
                let g2 = contract_t(graph.clone(), t);
                min(fast_cut(g1), fast_cut(g2))
        }
}

fn contract_t(mut graph: impl UnMulGraph + Clone, t: usize) -> impl UnMulGraph + Clone {
        while graph.len_vertices() > t {
                let (u, v) = graph.get_random_edge();
                graph.contract_edge(u, v);
        }
        graph.clone()
}
```

```rust
// Brute force algorithm for finding the minimum cut of a graph
fn min_cut(graph: impl UnMulGraph + Clone) -> usize {
        let mut min_cut = usize::MAX;
        for (set_a, set_b) in bipartitions(graph.vertex_set()) {
                // Count the edges crossing the partition
                let mut crossing_edges = 0;
                for u in &set_a {
                        for v in &set_b {
                                // Add the number of edges u, v
                                crossing_edges += graph.get_num_edges(*u, *v);
                        }
                }
                assert_ne!(crossing_edges, 0, "Crossing edges can't be 0");
                // Update the minimum cut
                if crossing_edges < min_cut {
                        min_cut = crossing_edges;
                }
        }
        min_cut
}

// Helper function to generate all possible bipartitions of a set
fn bipartitions(set: &HashSet<usize>) -> impl Iterator<Item=(HashSet<usize>, HashSet<usize>)> {
        let elements: Vec<_> = set.iter().cloned().collect();

        (1..(1 << elements.len())).filter_map(move |mask| {
```

```
                let mut part1 = HashSet::new();
                let mut part2 = HashSet::new();

                for (i, &elem) in elements.iter().enumerate() {
                        if (mask & (1 << i)) != 0 {
                                part1.insert(elem);
                        } else {
                                part2.insert(elem);
                        }
                }

                // Ensure that only one of the symmetric pairs is returned
                if !part1.is_empty() && !part2.is_empty() && part1.iter().min() <= part2.iter().min() {
                        Some((part1, part2))
                } else {
                        None
                }
        })
}
```

A.3 Graph

```
// Undirected MultiGraph Interface
trait UnMulGraph: Debug + Clone {
        fn add_edge(&mut self, u: usize, v: usize); // Add edge (u, v)
        fn contract_edge(&mut self, u: usize, v: usize); // Contract edge (u, v)
        fn get_num_edges(&self, u: usize, v: usize) -> usize; // Return the number of edges between u, v
        fn get_random_edge(&self) -> (usize, usize); // Return a random edge
        fn edge_exists(&self, u: usize, v: usize) -> bool; // Return if the edge belongs to the graph
        fn len_vertices(&self) -> usize; // Return the number of vertices
        fn len_edges(&self) -> usize; // Return the number of edges
}
```

```
// Implementation of the Undirected MultiGraph Interface
#[derive(Debug, Clone, Default)]
pub struct Graph {
        edge_list: Vec<(usize, usize)>, // [(u, v), ...]
        vertex_set: HashSet<usize>, // {u, v, ...}
}

impl UnMulGraph for Graph {
        fn add_edge(&mut self, u: usize, v: usize) {
                if u == v {
                        panic!("Self-loop is not allowed");
                }
                // Insert vertices into the HashSet
                self.vertex_set.insert(u);
                self.vertex_set.insert(v);
                // Add edge to the edge list
                self.edge_list.push((u.min(v), v.max(u)));
        }

        fn contract_edge(&mut self, u: usize, v: usize) {
                let (u, v) = (u.min(v), u.max(v));

                // Remove the edge (u, v) from the edge list
                self.edge_list.retain(|&e| e != (u, v));

                // Remove vertex `v` from the vertex set
                self.vertex_set.remove(&v);
```

```rust
                self.edge_list.iter_mut().for_each(|edge| {
                        if edge.0 == v { edge.0 = u; }
                        if edge.1 == v { edge.1 = u; }
                        if edge.0 > edge.1 {
                                *edge = (edge.1, edge.0);
                        }
                });
                self.edge_list.retain(|&(a, b)| a != b);

                // Remove vertex `u` if there are no more edges connected to it
                if !self.edge_list.iter().any(|&(a, b)| a == u || b == u) {
                        self.vertex_set.remove(&u);
                }
        }

        fn get_num_edges(&self, u: usize, v: usize) -> usize {
                self.edge_list.iter().filter(|&&(a, b)| (a == u && b == v) || (a == v && b == u)).count()
        }

        fn get_random_edge(&self) -> (usize, usize) {
                self.edge_list.iter().cloned().choose(&mut rand::thread_rng()).expect("No edges available")
        }

        fn edge_exists(&self, u: usize, v: usize) -> bool {
                self.edge_list.contains(&(u.min(v), v.max(u)))
        }

        fn len_vertices(&self) -> usize {
                self.vertex_set.len()
        }

        fn len_edges(&self) -> usize {
                self.edge_list.len()
        }

        fn vertex_set(&self) -> &HashSet<usize> {
                &self.vertex_set
        }
}
```