Implementation and comparison of min-cut and max-cut algorithms

Michał Miziołek

Research paper
Supervisor: dr hab. Krzysztof Turowski

Jagiellonian University
Institute of Theoretical Computer Science
Cracow 2024

Contents

Chapter 1

Introduction

1.1 Problem definitions

Before diving into the intricacies of the min-cut and max-cut problems, it is essential to outline a common understanding of the fundamental graph theoretical concepts that underpin our analysis. This section introduces the necessary definitions, primarily derived from standard texts in graph theory such as Cormen et al.'s *"Introduction to Algorithms"* [Cor+22] and Bollobás's *"Modern Graph Theory"* [Bol98].

Definition 1.1.1 (undirected graph)**.** A *graph* $G = (V, E)$ consists of a set V, which is a nonempty collection of *vertices*, and a set E, comprised of *edges*. Each edge is a 2-element subset of V, denoted as $\{u, v\}$, indicating a bidirectional connection between vertices u and v.

Definition 1.1.2 (directed graph)**.** A *directed graph* $G = (V, E)$ consists of a set V of vertices and a set E of edges. Each edge in a *directed graph* is an ordered pair of vertices, denoted as (u, v), indicating a directed connection from vertex u to vertex v.

Additionally, edges can be *parallel* (multiple edges with the same ordered pair of vertices) and can also include *loops* (edges that connect a vertex to itself, i.e., (u, u)).

Definition 1.1.3 (adjacency)**.** Two vertices u and v in a graph $G = (V, E)$ are *adjacent* if there exists an edge $e \in E$ that connects u and v. This edge e is said to be *incident* to both u and v.

Definition 1.1.4 (degree)**.** The *degree* of a vertex v denoted by $\deg(v)$ in graph $G = (V, E)$ is the number of edges incident to v.

Definition 1.1.5 (subgraph)**.** A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

Definition 1.1.6 (weighted graph)**.** A *weighted graph* $G = (V, E, w)$ is a graph in which each edge $e \in E$ has an associated *weight* $w(e)$. The weight function $w : E \to \mathbb{R}^+$ assigns a positive real number to each edge.

Definition 1.1.7 (induced subgraph)**.** An *induced subgraph* $G[V']$ of a graph $G = (V, E)$ is defined by a subset $V' \subseteq V$. The subgraph $G[V']$ includes all the edges in E that have both endpoints in V'.

Since we will be discussing not only exact algorithms, but also approximate ones, it is useful to recall the definitions of approximation ratio and APX class:

Definition 1.1.8 (approximation ratio)**.** The *approximation ratio* of an algorithm $\mathcal{A}$ for an optimization problem Π is a constant $\alpha \leq 1$ such that for every instance I of Π, the solution $\mathcal{A}(I)$ produced by $\mathcal{A}$ satisfies the inequality $\mathcal{A}(I) \geq \alpha \cdot \mathrm{OPT}(I)$, where $\mathrm{OPT}(I)$ denotes the optimal solution value for instance I.

Definition 1.1.9 (APX class)**.** The *APX class* is the set of all optimization problems for which there exists a polynomial-time approximation algorithm with a bounded approximation ratio. Formally, a problem Π is in APX if there exists a polynomial-time algorithm $\mathcal{A}$ and a constant $\alpha \leq 1$ such that for every instance I of Π, the solution $\mathcal{A}(I)$ produced by $\mathcal{A}$ satisfies the inequality $\mathcal{A}(I) \geq \alpha \cdot \mathrm{OPT}(I)$.

The following definitions are specific to the cut problem:

Definition 1.1.10 (cut)**.** A *cut* in a graph $G = (V, E)$ is a partition of V into two non-empty subsets S and T such that $S \cup T = V$ and $S \cap T = \emptyset$. The *cut-set* of a cut (S, T) is the set of all edges that have one endpoint in S and the other in T.

Definition 1.1.11 (capacity of a cut)**.** The *capacity* of a cut (S, T) in a weighted graph $G = (V, E, w)$ is defined as the sum of the weights of the edges in the cut-set of (S, T), where $w : E \to \mathbb{R}^+$ is the function assigning weights to the edges.

Definition 1.1.12 (max-cut problem)**.** The *max-cut problem* involves finding a cut (S, T) in a weighted graph $G = (V, E, w)$ such that the capacity of the cut is maximized.

Definition 1.1.13 (s-t cut)**.** An *s-t cut* in a graph $G = (V, E)$ is a partition of V into two non-empty subsets S and T such that $S \cup T = V$ and $S \cap T = \emptyset$. The *cut-set* of a cut (S, T) is the set of all edges that $s \in S$ and $t \in T$.

Definition 1.1.14 (min-cut problem)**.** The *min-cut problem* seeks to determine a cut (S, T) in a weighted graph $G = (V, E, w)$ that minimizes the capacity of the cut. The value of the cut is equivalent to minimum out of all s-t cuts, considering all possible pairs of s and t.

The definitions provided above set the stage for a detailed exploration of max-cut and min-cut problems.

1.2 Problem statement

This paper is dedicated to the in-depth investigation of two fundamental problems in graph theory: the max-cut and min-cut problems as formalized in Definition 1.1.12 and Definition 1.1.14. Throughout most of this paper, we will work with *undirected graphs*, with a

primary focus on searching for global min-cut and max-cut values. It is worth mentioning that only in the sections on Equivalence (Section 1.4) and the Hao-Orlin algorithm (Section 3.1), we will consider *directed graphs*. Our main objective is to implement, analyze, and compare various algorithms designed to find optimal and approximate solutions to these problems, focusing on both their theoretical efficiency and practical applicability.

The techniques to be utilized in this research include the implementation of various algorithms—ranging from straightforward naive approaches and branch and bound strategies to more sophisticated techniques such as the Goemans-Williamson algorithm for max-cut and the Stoer-Wagner, Karger's, and Hao-Orlin algorithms for min-cut. We will analyze these algorithms for correctness, time complexity, and their practical utility across different graph structures characterized by varying size, density, and weight distributions.

Current algorithms allows min-cut can to be solved in polynomial time using among others the Ford-Fulkerson algorithm [FF56]. In contrast, the max-cut problem is known to be NP-hard [Kar72], making it a more challenging problem from a computational complexity perspective.

To be more specific, the max-cut Problem is APX-hard, meaning that there is no polynomial-time approximation scheme (PTAS), arbitrarily close to the optimal solution, for it, unless P = NP [PY91]. Thus, there is $\epsilon > 0$, such no $(1 + \epsilon)$-approximation or $(1 - \epsilon)$-approximation algorithm exists for max-cut. If we assume that the unique games conjecture (UGC) is correct, then the best known approximation ratio for max-cut would be $0.87856 - \epsilon$ for any $\epsilon > 0$ [Kho+07]. However, if it is not true, then it is proven that this problem is NP-hard with approximation ratio better than $\frac{16}{17}$ [Hås01]. Thus, every known polynomial-time approximation algorithm achieves an approximation ratio strictly less than one.

In certain graph classes, the min-cut problem can be solved much faster. For example, in planar graphs, an optimal min-cut can be found in $O(|V| \log^2 |V|)$ using algorithm by Łącki and Sankowski [ŁS11]. Similarly, the max-cut problem, while generally NP-hard, can be tractable in specific cases [GJ79]. For instance, in planar graphs, the max-cut problem can be solved in polynomial time, specifically $O(|V|^{1.5} \log V)$, by transforming the problem into a minimum-weight perfect matching problem on an associated graph [LP12]. In the context of parameterized complexity, the max-cut problem parameterized by the size of the cut k is fixed-parameter tractable (FPT) [DF13]. This means that there exists an algorithm that solves the max-cut problem in $f(k) \cdot |I|^{O(1)}$ time, where f is a function depending only on k and $|I|$ is the size of the input. However, certain parameterizations of the max-cut problem, such as parameterizing by the number of vertices, lead to W[1]-hard problems, indicating that no FPT algorithm is likely to exist for these cases unless FPT= W[1] [DF13].

Further, this paper will undertake a comparative evaluation based on experimental results to determine under what conditions each algorithm exhibits optimal performance. The duality between the max-flow and min-cut problems will also be explored to demonstrate how solutions to one can inform solutions to the other, enriching both the theoretical understanding and practical application perspectives.

1.3 Historical background

The exploration of the max-cut and min-cut problems within graph theory presents a historical journey through computational complexity and algorithmic development.

Origins and Early Developments: The study of cuts in graphs initially arose from the fields of network design and operations research in the mid-20th century. The min-cut problem, in particular, gained prominence through its association with network reliability. Early solutions to the min-cut problem leveraged the duality between minimum cuts and maximum flows, a relationship formalized by the Ford and Fulkerson's max-flow min-cut theorem in 1956 [FF56]. This theorem provided not only a foundational theoretical result but also efficient polynomial-time algorithms for finding minimum cuts in networks, such as the Edmonds-Karp algorithm [EK72], which emerged as an implementation of the Ford-Fulkerson method.

The Rise of NP-Completeness: The concept of NP-completeness introduced by Stephen Cook in 1971 and independently by Leonid Levin in 1973 reshaped the landscape of computational theory. The max-cut problem was among the first problems to be classified as NP-complete by Richard Karp in his seminal 1972 paper [Kar72]. This classification underlined the computational difficulty of the problem and sparked an extensive search for efficient heuristic and approximation algorithms, given the lack of polynomial-time solutions for NP-complete problems in general.

Seminal Approximations and Innovations: A significant milestone in the study of the max-cut problem was the introduction of the Goemans-Williamson algorithm in 1995 [GW95]. This algorithm marked a breakthrough in approximation techniques, utilizing semidefinite programming to achieve a performance guarantee significantly better than any previous approach. The Goemans-Williamson algorithm demonstrated that it was possible to approximate NP-hard problems like max-cut to within about 0.878 of the optimal solution, assuming the Unique Games Conjecture [Kho+07]. This result not only advanced theoretical understanding but also influenced subsequent research in algorithmic approximations for other NP-hard problems.

Development of Min-Cut Algorithms: Parallel to the developments in the max-cut problem, significant advancements were made in algorithms for the min-cut problem. The Stoer-Wagner algorithm, introduced in 1997 [SW97], is a notable example that offers an efficient and practical method for finding the minimum cut in undirected graphs. Its simplicity and effectiveness have made it a standard approach in various applications.

On the more randomized side of algorithm design, Karger's algorithm and its refined version, Karger-Stein algorithm, introduced in the late 1990s [KS96], provided a probabilistic method to compute minimum cuts with high probability, enhancing the understanding of how randomness can be leveraged to solve deterministic problems effectively.

Further Algorithmic Contributions: Algorithms like the Hao-Orlin algorithm [HO94] further refined the ability to compute minimum cuts, offering more efficient ways to handle larger and more complex network structures. On the heuristic side, algorithms such as the Burer algorithm [BMZ02] contributed to the toolbox available for tackling the max-cut problem, particularly in specific graph classes where approximation could be more

effectively achieved.

Authors	Year	Time	Approx. ratio	Refs
Sahni & Gonzales	**1976**	$O(\|V\| \cdot \|E\|^2)$	**0.5**	[SG76]
Branch and Bound	**N/A**	$O(2^{\|V\|} \cdot \|E\|))$	**exact**	**folklore**
Burer et al.	**1990**	$O((T+k) \cdot \|V\|^2)$	**heuristic**	[BMZ02]
Goemans & Williamson	**1995**	*polynomial*	**0.878**	[GW95]

Table 1.1: Table of significant max-cut algorithms (implemented algorithms in bold).

Authors	Year	Time	Refs
Ford & Fulkerson	1956	$O(\|E\| \cdot f)$	[FF56]
Edmonds & Karp	1972	$O(\|V\| \cdot \|E\|^2)$	[Kar72]
Karger	**1993**	$O(\|V\|^2 \cdot \|E\|)$	[Kar93]
Hao & Orlin	**1994**	$O(\|V\|^2 \cdot \|E\|)$	[HO94]
Karger & Stein	**1996**	$O(\|V\|^2 \cdot \log^3 \|V\|)$	[KS96]
Stoer & Wagner	**1997**	$O(\|V\| \cdot \|E\| + \|V\|^2 \cdot \log \|V\|)$	[SW97]

Table 1.2: Table of significant min-cut algorithms (implemented algorithms in bold).

1.4 Equivalence of min-cut and max-flow

1.4.1 Definitions

Suppose we are given a directed graph G (Definition 1.1.2), with two distinguished types of vertices: the *source* and the *sink*. The source is a designated vertex from which edges solely emanate, meaning all outgoing edges originate from this vertex. Conversely, the sink is a designated vertex into which edges solely converge, indicating that all incoming edges terminate at this vertex.

Definition 1.4.1 (Capacity function)**.** Denoted by c, this function maps each edge to a positive integer, representing the capacity of the respective edge, $c : E \to \mathbb{N}^+$.

Definition 1.4.2 (Integer flow)**.** A function f that assigns a non-negative integer to each edge, $f : E \to \mathbb{N}$, such that $f(u, v) \leq c(u, v)$ for all $(u, v) \in E$. For any vertex v that is neither the source nor the sink, the sum of the flows into v equals the sum of the flows out of v:

$$\sum_{u \in N(v)} f(u, v) = \sum_{u \in N(v)} f(v, u),$$

Definition 1.4.3 (Cut capacity)**.** Denoted by $c(S,T)$, the cut capacity is the sum of the capacities of the edges from S to T:

$$c(S,T) = \sum_{(u,v)\in E: u\in S, v\in T} c(u,v).$$

A minimum s-t cut is a cut with the minimum capacity among all possible cuts.

Definition 1.4.4 (Flow through a cut)**.** Defined as the sum of the flows of the edges from S to T minus the sum of the flows of the edges from T to S:

$$f(S,T) = \sum_{(u,v)\in E: u\in S, v\in T} f(u,v) - \sum_{(u,v)\in E: u\in T, v\in S} f(u,v).$$

Definition 1.4.5 (Residual graph)**.** Given a flow f on a graph G, the *residual graph* G_f is constructed by assigning each edge (u,v) a residual capacity $c_f(u,v) = c(u,v) - f(u,v)$. If $f(u,v) > 0$, the edge (v,u) is also included in the residual graph with a residual capacity $c_f(v,u) = f(u,v)$.

Definition 1.4.6 (Augmenting path)**.** A sequence of edges forming a path from the source to the sink in the residual graph. In an augmenting path, a *forward'* edge must have residual capacity (i.e., $f(u,v) < c(u,v)$), and a *backward* edge must have flow (i.e., $f(u,v) > 0$). By increasing the flow along forward edges and decreasing it along backward edges, the overall flow from the source to the sink can be increased.

Sometimes, the value of a flow f is denoted as $\text{val}(f) = \sum_{(v,t)\in E} f(v,t)$, representing the total flow into the sink, which we aim to maximize.

1.4.2 Proof

Lemma 1.4.1 (First Lemma about Cuts)**.** For any flow f in a flow network and for any cut (S,T), the flow through the cut is less than or equal to its capacity, i.e., $f(S,T) \leq c(S,T)$.

Proof. Consider the cut (S,T) in a flow network. The flow through the cut (S,T) is given by:

$$f(S,T) = \sum_{(u,v)\in E: u\in S, v\in T} f(u,v) - \sum_{(u,v)\in E: u\in S, v\in T} f(v,u).$$

By definition, the capacity of the cut (S,T) is:

$$c(S,T) = \sum_{(u,v)\in E: u\in S, v\in T} c(u,v).$$

Since $f(u,v) \leq c(u,v)$ for all edges $(u,v) \in E$, we have:

$$\sum_{(u,v)\in E: u\in S, v\in T} f(u,v) \leq \sum_{(u,v)\in E: u\in S, v\in T} c(u,v).$$

Additionally, since $f(u, v) \geq 0$ for all edges $(u, v) \in E$, we have:

$$\sum_{(u,v)\in E: u\in S, v\in T} f(v, u) \geq 0.$$

Therefore, combining these inequalities, we obtain:

$$\begin{aligned} f(S, T) &= \sum_{(u,v)\in E: u\in S, v\in T} f(u, v) - \sum_{(u,v)\in E: u\in S, v\in T} f(v, u) \\ &\leq \sum_{(u,v)\in E: u\in S, v\in T} c(u, v) = c(S, T). \end{aligned}$$

Hence, the flow through the cut (S, T) is less than or equal to its capacity, which completes the proof. □

Lemma 1.4.2 (Second Lemma about Cuts)**.** The flow through each cut of the network is the same and equals val(f).

Proof. We proceed by induction on the number of vertices in the set S of the cut (S, T) with $s \in S$.
Base Case: When $|S| = 1$, then $S = \{s\}$. Since there are no edges from the source to itself, the flow through this trivial cut is zero, which equals val(f).
Inductive Step: Assume that for any cut (S', T') where $|S'| < |S|$, the flow through the cut is exact val(f).

Consider a cut (S, T) with $|S| = k$. Select a vertex $x \in S$ such that x is not the source. Construct a new cut (S', T') by moving x from S to T, i.e., $S' = S \setminus \{x\}$ and $T' = T \cup \{x\}$. By the inductive hypothesis, the flow through (S', T') is val(f).

We now express the flows through (S, T) and (S', T'):

$$\begin{aligned} f(S, T) &= \sum_{(u,v)\in E: u\in S, v\in T} f(u, v) - \sum_{(v,u)\in E: u\in S, v\in T} f(v, u), \\ f(S', T') &= \sum_{(u,v)\in E: u\in S', v\in T'} f(u, v) - \sum_{(v,u)\in E: u\in S', v\in T'} f(v, u). \end{aligned}$$

Since $S' = S \setminus \{x\}$ and $T' = T \cup \{x\}$, the sums can be rewritten as:

$$\begin{aligned} f(S, T) = &\sum_{(u,v)\in E: u\neq x, u\in S, v\in T} f(u, v) + \sum_{(x,v)\in E: v\in T} f(x, v) \\ &- \sum_{(v,u)\in E: u\neq x, u\in S, v\in T} f(v, u) - \sum_{(v,x)\in E: v\in T} f(v, x), \\ f(S', T') = &\sum_{(u,v)\in E: u\in S', v\neq x, v\in T'} f(u, v) + \sum_{(u,x)\in E: u\in S'} f(u, x) \\ &- \sum_{(v,u)\in E: u\in S', v\neq x, v\in T'} f(v, u) - \sum_{(x,u)\in E: u\in S'} f(x, u). \end{aligned}$$

Notice that for any vertex $v \neq x$, $v \in T'$ if and only if $v \in T$, and for any vertex $u \neq x$, $u \in S'$ if and only if $u \in S$. Thus, we can simplify:

$$\begin{aligned} f(S', T') = &\sum_{(u,v) \in E : u \in S, v \neq x, v \in T} f(u, v) + \sum_{(u,x) \in E : u \in S} f(u, x) \\ &- \sum_{(v,u) \in E : u \in S, v \neq x, v \in T} f(v, u) - \sum_{(x,u) \in E : u \in S} f(x, u). \end{aligned}$$

Thus, the difference between $f(S, T)$ and $f(S', T')$ is:

$$\begin{aligned} f(S, T) - f(S', T') = &\left(\sum_{(x,v) \in E : v \in T} f(x, v) - \sum_{(v,x) \in E : v \in T} f(v, x) \right) \\ &- \left(\sum_{(u,x) \in E : u \in S} f(u, x) - \sum_{(x,u) \in E : u \in S} f(x, u) \right). \end{aligned}$$

Since the total flow into x equals the total flow out of x:

$$\sum_{(u,x) \in E} f(u, x) = \sum_{(x,v) \in E} f(x, v),$$

we have:

$$f(S, T) - f(S', T') = 0.$$

Hence, $f(S, T) = f(S', T')$. By the inductive hypothesis, $f(S', T') = \text{val}(f)$. Therefore, $f(S, T) = \text{val}(f)$, completing the proof. □

Now we will try to proof the max-flow minimum cut theorem, where the value of the maximum flow from the source to the sink is equal to the capacity of the minimum cut. To be more specific:

Theorem 1.4.1. The following conditions are equivalent:

1. f is a maximum flow.
2. There is no augmenting path from the source to the sink in the residual graph.
3. The cut (S, T), where S includes all vertices that can be reached by an augmenting path from the source, is a well-defined cut, satisfying the condition $f(S, T) = c(S, T)$.

Proof. The third condition might sound daunting, but it really isn't. Let's address the individual implications to prove the equivalence:

(1) $\implies$ (2) If f were a maximum flow and there existed an augmenting path from the source to the sink, then the flow value could be increased by 1 using this path, which would mean that the flow was not maximum.

(2) $\implies$ (3) The idea is that the correctness of the cut stated in (3) follows from (2) because there is no augmenting path from the source to the sink, hence the sink must definitely be in T. The source will obviously be in S. The remaining conditions for the cut will naturally be met, so we have that (S, T) is a well-defined cut. Now, looking at all edges entering and leaving S, we discover that since there is no augmenting path that could extend beyond S, all outgoing edges must be fully saturated and all incoming ones depleted (otherwise, we could add a vertex from T to S, according to our cut definition). Thus, we have that $f(S, T) = c(S, T)$; in other words, the flow through the cut equals its capacity.

(3) $\implies$ (1) Since $\text{val}(f)$ must be less than the capacity of any cut, as we noticed at the stage of defining the terms (and formalists hopefully have proven, unless they are still pondering what an even number is), and from (3) we have that the cut (S, T) satisfies $f(S, T) = c(S, T)$, then we have that this flow is maximum (i.e., it is not possible to achieve a greater flow since we have just reached the limit).

From the above proof, it also follows that the capacity of the minimum cut equals $\text{val}(f)$. □

1.5 Practical applications

The min-cut and max-cut algorithms have many practical applications that illustrates their crucial role and showcasing their utility in solving complex real-world problems.

In network design, especially within telecommunications and data centers, ensuring uninterrupted service is a major concern. The use of min-cut algorithms such as the Stoer-Wagner algorithm is fundamental in these scenarios. These algorithms are applied to model the network as a graph where nodes represent switches, routers, and hubs, while edges represent the communication links. The min-cut algorithm calculates the minimum set of edges that, if removed, would disconnect the network, allowing engineers to strategically place redundancies or increase capacities in critical areas to enhance fault tolerance [SW97].

In VLSI design, max-cut algorithms are employed to manage the circuit layout on silicon chips efficiently. The Goemans-Williamson algorithm, for example, is used to partition the graph representing the circuit, optimizing the placement of circuit components to minimize interconnect lengths and delay, crucial for improving the chip's performance [GW95].

In machine learning, specifically in unsupervised data clustering, min-cut algorithms like Karger's algorithm optimize data groupings. This method is particularly beneficial in genomic research, where accurately grouping genetic markers can lead to better understanding of genetic traits and disease markers [Kar93].

Financial markets also leverage max-cut algorithms for portfolio optimization, identifying asset groups with minimal cross-correlations to diversify risk effectively. This technical application helps construct portfolios that are resilient to market volatilities, maximizing returns by managing the correlations between different investment categories [CR93].

Lastly, in power grid management, min-cut algorithms provide insights into the vulnerability of electrical grids. These algorithms enhance the resilience of power systems against failures and natural disasters, ensuring stable and reliable energy distribution [Bie10].

This are only few examples of technical implementations of min-cut and max-cut algorithms that showcases their significant impact on optimizing and safeguarding critical systems in various industries.

Chapter 2

Max-Cut

2.1 Greedy algorithm

2.1.1 Idea and Correctness

The greedy algorithm for the max-cut problem operates under a straightforward yet powerful premise: explore local vertex partition improvements to maximize the sum of weights across the cut edges. The algorithm iteratively evaluates the cut-value of different vertex subsets, systematically flipping the set membership of each vertex to explore potential improvements in the cut value. Therefore we will adjust the partition by moving one vertex at a time from one set to another, checking whether such moves increase the total edge weights that cross from one set to the other.

2.1.2 Implementation

The implementation of the Greedy max-cut algorithm is structured to continuously evaluate and adjust vertex partitions. Initially, all vertices are assigned to one subset, and the algorithm iteratively flips the membership of each vertex to see if the move results in a higher cut value. This is done by the function `calculateCutValue`, which computes the total weight of the edges crossing between the two subsets for a given partition.

The core of the implementation resides in the `run` method, where the algorithm uses a greedy approach:

- It starts with all vertices in one subset.
- For each vertex, it tests the effect of moving it to the opposite subset.
- If the move increases the total weight of the cut edges, the move is kept; otherwise, it is reverted.

This process repeats until no further improvements can be made, ensuring that each vertex placement is locally optimal with respect to the cut value.

```
1 void NaiveMaxCut::run() {
2     maxCutValue = 0;
```

```
    std::vector<bool> set(graph->numberOfNodes(), false);
    bool improved = true;
    while (improved) {
        improved = false;
        for (int i = 0; i < graph->numberOfNodes(); i++) {
            set[i] = !set[i];
            double newCut = calculateCutValue(set);
            if (newCut > maxCutValue) {
                maxCutValue = newCut;
                maxCutSet = set;
                improved = true;
            } else {
                set[i] = !set[i];
            }
        }
    }
}
```

2.1.3 Correctness and time complexity

Theorem 2.1.1 (Lemma A1 [SG76])**.** The greedy algorithm terminates when at least half of the edges belong to the cut ensuring it is an 0.5-approximation algorithm.

Proof. Let $G = (V, E)$ be an input graph. Consider the state of the graph when the algorithm terminates.

For any vertex $v \in V$, let $\deg(v)$ denote the degree of v, which is the number of edges incident to v. Let S and T be a cut of G.

Assume for contradiction that there exists a vertex $v \in S$ (without loss of generality) for which fewer than half of the edges incident to v cross the cut. Formally:

$$\sum_{u \in T} w(v, u) < \frac{1}{2} \deg(v)$$

where $\forall_{(u,v)\in E}, : w(v, u) = 1$.

If vertex v were moved to the other subset T, the number of edges crossing the cut would change. Specifically, each edge (v, u) for $u \in T$ would no longer cross the cut, and each edge (v, u) for $u \in S$ would begin to cross the cut. The net effect would be:

$$c(S', T') = c(S, T) + \left(\deg(v) - 2 \sum_{u \in T} w(v, u) \right)$$

where $S' = S \setminus \{v\}$ and $T' = T \cup \{v\}$. Since $\sum_{u \in T} w(v, u) < \frac{1}{2} \deg(v)$, we have:

$$\deg(v) - 2 \sum_{u \in T} w(v, u) > 0$$

Therefore, moving v would increase the cut value, contradicting the assumption that the algorithm has terminated and no further improvement is possible.

Consequently, at least half of the edges incident to each vertex must belong to the cut, ensuring that the cut includes at least $|E|/2$ edges. This establishes the 0.5-approximation guarantee. □

Theorem 2.1.2. The time complexity of the Naive max-cut algorithm is $O(|V| \cdot |E|^2)$.

Proof. The time complexity of the greedy max-cut algorithm is determined by the number of iterations and the operations performed within each iteration. The algorithm improves the cut value by at least one edge per iteration, and since there are $|E|$ edges, it iterates at most $|E|$ times (see Lemma A2 [SG76]). Within each iteration, the algorithm evaluates each vertex, flipping its subset membership and recalculating the cut value, which involves inspecting all incident edges. Calculating the cut value requires $O(|E|)$ operations, and evaluating all $|V|$ vertices results in $O(|V| \cdot |E|)$ work per iteration. Therefore, the total time complexity is $O(|E| \cdot (|V| \cdot |E|)) = O(|V| \cdot |E|^2)$. □

2.2 Branch and Bound algorithm

2.2.1 Idea

The Branch and Bound algorithm is an optimization technique that systematically explores the solution space for combinatorial problems like the max-cut problem. The core idea is to iteratively partition the problem into smaller subproblems (branching) and compute bounds to eliminate subproblems that cannot yield better solutions than the best one found so far (bounding).

To visualize this, consider a binary tree where each node represents a decision to include a vertex in either subset S or T. The root node represents the initial state with no vertices assigned. Each level of the tree corresponds to a decision about one vertex, and the bounding step helps in cutting off large portions of the tree that cannot contain the optimal solution, thus saving computation time.

Formally, for a graph $G = (V, E)$ with vertices $V = \{v_1, v_2, \ldots, v_n\}$, we seek to partition V into two disjoint subsets S and T such that the sum of the weights of the edges between S and T is maximized. The algorithm can be described as follows:

At each step, decide whether to include a vertex v_i in subset S or subset T. This decision creates a branching tree of possible subsets. After that, for each partial partition, calculate an upper bound on the possible maximum cut value that can be achieved if the partition is extended. If this bound is less than the current best known cut value, prune that branch.

This method effectively narrows down the solution space, focusing only on promising subproblems and thereby improving efficiency over naive exhaustive search methods.

2.2.2 Implementation

The implementation of the Branch and Bound algorithm involves several key components

The `bound` function calculates an upper bound on the maximum cut value for a given node. It adds the maximum possible contributions from unassigned vertices to the current cut value. This ensures that the calculated bound is as tight as possible, facilitating effective pruning of the search tree.

```
int BranchAndBoundMaxCut::bound(Node u) {
    int result = calculateCutValue(u.set);
    graph->forEdges([&](node j, node k, edgeweight w) {
        if (j >= u.level) {
            result += w;
        }
    });
    return result;
}
```

The `branchAndBound` function is the core of the algorithm: at the beginning, the stack is initialized with the root node, representing an empty partition. While the stack is not empty, nodes are removed from it and explored. If a node represents assignment of the last vertex, its cut value is calculated and compared to the best known value. For non-leaf nodes, two new nodes are created by assigning the current vertex to either subset S or T, and their bounds are calculated. New nodes are pushed onto the stack only if their bounds indicate potential for a better solution.

```
void BranchAndBoundMaxCut::branchAndBound() {
    maxCutValue = 0;
    std::vector<Node> stack;
    Node root;
    root.level = 0;
    root.set.resize(graph->numberOfNodes(), false);
    root.bound = bound(root);
    stack.push_back(root);
    while (!stack.empty()) {
        Node u = stack.back();
        stack.pop_back();
        if (u.level == graph->numberOfNodes()) {
            double currentCutValue = calculateCutValue(u.set);
            if (currentCutValue > maxCutValue) {
                maxCutValue = currentCutValue;
                maxCutSet = u.set;
            }
        } else {
            for (int i = 0; i < 2; ++i) {
                Node v = u;
                v.level = u.level + 1;
```

```
22                  v.set[u.level] = i;
23                  v.bound = bound(v);
24                  if (v.bound > maxCutValue) {
25                      stack.push_back(v);
26                  }
27              }
28          }
29      }
30  }
```

2.2.3 Correctness

The correctness of the Branch and Bound algorithm for max-cut is ensured by its approach to exploring the solution space and its use of bounding to non-promising subproblems. The algorithm maintains a global best solution and prunes branches where the upper bound indicates no possible improvement over this best solution.

The bounding function ensures that if a branch is abandoned, no better solution exists within that branch. This is because the bound represents the maximum possible cut value that can be achieved from that node onward.This guarantees that the algorithm does not miss any potential solutions.

Whenever a leaf node representing some cut (S, T) is reached, the actual cut value is compared against the current best found value, ensuring that the best known solution is always updated.

Thus, the Branch and Bound algorithm correctly finds the maximum cut by systematically exploring feasible partitions and using bounds to eliminate non-promising ones.

2.2.4 Time complexity

The time complexity of the Branch and Bound algorithm is influenced by the effectiveness of the bounding function and the graph's structure. In the worst case, the algorithm explores an exponential number of nodes, similar to a naive exhaustive search, leading to a time complexity of $O(2^{|V|} \cdot |E|)$. However, in practice, the bounding step significantly reduces the number of nodes to be explored, making the algorithm more efficient for many practical instances.

2.3 Burer algorithm

2.3.1 Idea and correctness

The Burer algorithm introduced in [BMZ02] is a heuristic method designed to find an approximate solution to the max-cut problem.

Formulation and Relaxation

The Max-Cut problem can be formulated as the following binary quadratic program:

$$\text{maximize} \quad \frac{1}{2}\sum_{i<j} w(i,j)\cdot(1-x_i x_j) \quad \text{subject to} \quad x_i \in \{-1,1\}, \text{ for all } i,$$

where $w(i,j)$ represents the weight of the edge between vertices i and j, and x_i is a binary variable indicating the subset to which vertex i belongs.

This problem can be equivalently written as:

$$\text{minimize} \quad \sum_{i<j} w(i,j)\cdot x_i x_j \quad \text{subject to} \quad x_i \in \{-1,1\}, \text{ for all } i,$$

Further, this can be relaxed into a semidefinite programming (SDP) problem:

$$\text{minimize} \quad \frac{1}{2}W\cdot X \quad \text{subject to} \quad X_{ii}=1 \text{ for } i=1,\ldots,n, \quad X\succeq 0$$

where W is the matrix of weights $w(i,j)$, X is a positive semidefinite matrix, which means that it can be represented as $X=V^TV$ for some matrix V. This property ensures that all eigenvalues of X are non-negative and implies that X does not have any negative quadratic forms.

Rank-Two Relaxation

Instead of solving the full SDP relaxation, Burer et al. propose a rank-two relaxation:

$$\text{minimize} \quad f(\theta)=\frac{1}{2}W\cdot\cos(T(\theta)) \quad \text{where} \quad T_{ij}(\theta)=\theta_i-\theta_j$$

In this formulation, each variable x_i is represented by an angle θ_i on the unit circle. The matrix $T(\theta)$ contains the pairwise differences of these angles.

Using polar coordinates, each x_i is mapped to θ_i such that:

$$v_i=\begin{pmatrix}\cos(\theta_i)\\ \sin(\theta_i)\end{pmatrix}$$

and the inner product $x_i x_j$ is represented as $\cos(\theta_i-\theta_j)$.

Algorithmic Insight

The core of the Burer algorithm involves the following steps:

1. Start with an initial guess for θ.
2. Use a gradient descent method to minimize the function $f(\theta)$. The gradient $\nabla f(\theta)$

is given by:

$$\frac{\partial f(\theta)}{\partial \theta_j} = \sum_k w(k,j) \cdot \sin(\theta_k - \theta_j).$$

3. Once a local minimum θ is found, generate a cut using the `procedureCut`. This involves partitioning the unit circle at various angles to find the best cut.

ProcedureCut

The `procedureCut` is an efficient method to generate the best possible cut from a given angle configuration θ (see Section 4 in [BMZ02]). Here's how it works:

1. Given θ, sort the angles θ_i such that $\theta_1 \leq \theta_2 \leq \ldots \leq \theta_n$.
2. Initialize $\alpha = 0$, $\Gamma = -\infty$, and set $i = 1$.
3. Let j be the smallest index such that $\theta_j > \pi$. If there is no such j, set $j = n + 1$.
4. While $\alpha \leq \pi$:
   (a) Generate a cut x by setting:

$$x_i = \begin{cases} +1 & \text{if } \theta_i \in [\alpha, \alpha + \pi) \\ -1 & \text{otherwise} \end{cases}$$

   (b) Compute the cut value $\gamma(x)$.
   (c) If $\gamma(x) > \Gamma$, update $\Gamma = \gamma(x)$ and $x^* = x$.
   (d) Adjust α:

$$\alpha = \begin{cases} \theta_i & \text{if } \theta_i \leq \theta_j - \pi \\ \theta_j - \pi & \text{otherwise} \end{cases}$$

   (e) Increment i or j accordingly - in the first case above increment j, otherwise i.
5. Return the best cut x^*.

.

2.3.2 Implementation

The `run` method orchestrates the entire process of finding the maximum cut. First we call `perturbTheta` which is responsible for distributing angle values uniformly and then performing gradient descent.

After that we calculate `procedureCut` which we discussed in Section 2.3.1, which calculates the minimum cut.

```cpp
void RankTwoRelaxationMaxCut::run() {
    theta.resize(graph->numberOfNodes());
    distributeThetaEvenly();
    perturbTheta();
    maxCutSet = procedureCut();
    maxCutValue = calculateCutValue(maxCutSet);
    if (maxCutSet[graph->numberOfNodes() - 1] == true) {
        maxCutSet[graph->numberOfNodes() - 1] = false;
        double candidateValue = calculateCutValue(maxCutSet);
        if (candidateValue > maxCutValue) {
            maxCutValue = candidateValue;
        }
    }
}
```

`procedureCut` implementation:

```cpp
std::vector<bool> RankTwoRelaxationMaxCut::procedureCut() {
    double bestValue = -std::numeric_limits<double>::infinity();
    std::vector<bool> bestCut(graph->numberOfNodes()), x(graph->numberOfNodes());
    for (double alpha = 0; alpha <= M_PI; alpha += 0.01) {
        for (int i = 0; i < graph->numberOfNodes(); ++i) {
            x[i] = (theta[i] >= alpha && theta[i] < alpha + M_PI) ? true : false;
        }
        double value = calculateCutValue(x);
        if (value > bestValue) {
            bestValue = value;
            bestCut = x;
        }
    }
    return bestCut;
}
```

2.3.3 Time complexity

Let T be the number of gradient descent iterations and k the number of partitions in `procedureCut`.

Combining all components, the overall time complexity of the Burer Algorithm can be summarized as follows:

- **Initialization:** $\mathcal{O}(|V|)$
- **Gradient Descent:** $\mathcal{O}(T \cdot |V|^2)$
- **Cut Evaluation:** $\mathcal{O}(|V|^2)$

- **Procedure-CUT:** $\mathcal{O}(k \cdot |V|^2)$, where k is the number of partitions.

Thus, the total time complexity of the Burer Algorithm is dominated by the gradient descent and `procedureCut` steps, yielding:

$$\mathcal{O}(T \cdot |V|^2 + k \cdot |V|^2) = \mathcal{O}((T + k) \cdot |V|^2)$$

In practical scenarios, T and k are constants or logarithmic relative to $|V|$, making the algorithm efficient for large graphs.

2.4 Goemans-Williamson algorithm

2.4.1 Idea

The Goemans-Williamson algorithm is one of the approximation algorithms, utilizing semidefinite programming to achieve an approximation ratio of $\alpha = \min\limits_{0 \leq \theta \leq \pi} \frac{2}{\pi} \frac{\theta}{1-\cos\theta} \approx 0.87856$. Instead of assigning a binary value to each vertex (indicating which set it belongs to), the algorithm assigns a vector on the unit sphere. This relaxation allows the problem to be represented in a higher-dimensional space.

To comprehend the fundamental aspects of the algorithm, it is essential to discuss the principles of semidefinite programming:

Semidefinite Programming (SDP)

Semidefinite programming (SDP) is a subfield of convex optimization that generalizes linear programming [GM12]. In SDP, we optimize a linear objective function subject to the constraint that an affine combination of symmetric matrices is positive semidefinite. Mathematically, an SDP problem can be formulated as:

$$\begin{aligned} \text{maximize} \quad & \operatorname{Tr}(C^T X) \\ \text{subject to} \quad & \operatorname{Tr}(A_i^T X) = b_i, \quad i = 1, \ldots, m \\ & X \succeq 0, \end{aligned}$$

where X is a symmetric matrix variable, C and A_i are given symmetric matrices, b_i are given scalars, and $X \succeq 0$ means that X is positive semidefinite.

In the context of the Goemans-Williamson algorithm for the max-cut problem, the semidefinite programming (SDP) model aims to maximize an objective function that is a relaxation of the combinatorial max-cut problem. This SDP relaxation provides a framework where the combinatorial problem's constraints are captured in a way that can be efficiently solved, yielding a solution that is a bound on the actual integer problem.

The objective function of the SDP is designed to maximize the sum of the weights of edges between different sets in a partition of the vertices. For an undirected graph $G = (V, E)$ with edge weights $w(i, j)$ between vertices $i, j \in V$, the SDP maximizes:

$$\text{maximize} \quad \frac{1}{2} \sum_{i,j \in V} w(i,j) \cdot (1 - X_{ij})$$

where X is a positive semidefinite matrix, and $w(i,j)$ denotes the entry of X corresponding to vertices i and j. The term $(1 - X_{ij})$ effectively captures the contribution to the cut value from edge (i,j) if i and j are in different sets of the partition.

Transformation to Trace Form

The objective function can be transformed into a trace form, which is more convenient for semidefinite programming:

$$\frac{1}{2} \sum_{i,j \in V} w(i,j) \cdot (1 - X_{ij}) = \frac{1}{2} \left(\sum_{i,j \in V} w(i,j) - \sum_{i,j \in V} w(i,j) \cdot X_{ij} \right)$$

Since $\sum_{i,j \in V} w(i,j)$ is a constant, maximizing the above expression is equivalent to maximizing:

$$\sum_{i,j \in V} w(i,j) \cdot (1 - X_{ij}) = \text{Tr}(W) - \text{Tr}(WX)$$

where W is the matrix of edge weights $w(i,j)$. Therefore, the problem reduces to:

$$\text{maximize} \quad \text{-Tr}(WX)$$

subject to the constraints $X \succeq 0$ and $X_{ii} = 1$ for all i.

Performance Guarantee

The performance guarantee of the Goemans-Williamson algorithm is derived from the above analysis. The algorithm achieves an approximation ratio of (Theorem 2.4.1):

$$\alpha = \min_{0 \leq \theta \leq \pi} \frac{2}{\pi} \frac{\theta}{1 - \cos\theta} \approx 0.87856$$

This is a significant improvement over previous approximation algorithms and represents a major advance in the field of combinatorial optimization.

2.4.2 Implementation

1. **Semidefinite Relaxation**: The max-cut problem is first relaxed to a semidefinite programming problem. This involves replacing the binary variables with vectors of

unit length. The objective function is defined as:

$$\begin{aligned} \text{maximize} \quad & \operatorname{Tr}(C^T X) \\ \text{subject to} \quad & \operatorname{Tr}(A_i^T X) = b_i, \quad i = 1, \ldots, m \\ & X \succeq 0, \end{aligned}$$

where:

(a) **Block Matrix C:** This matrix represents the objective function of the SDP. Each entry C_{ij} is set to $-w(i,j)$ where $w(i,j)$ is the weight of the edge (i,j) in the graph. The matrix is symmetric and the diagonal entries are zero.

(b) **Linear Constraints b:** The constraints ensure that the resulting vectors $\mathbf{u}_i$ lie on the unit sphere. Each constraint b_i is set to 1, indicating that the diagonal entries of the SDP matrix should be 1.

(c) **Constraint Matrices A_i:** The constraints are $\|\mathbf{u}_i\|^2 = 1$, which translate to $X_{ii} = 1$ in the matrix X, where $X_{ij} = \mathbf{u}_i \cdot \mathbf{u}_j$. Therefore, A_i are diagonal matrices where the i-th diagonal entry is 1 and all other entries are 0.

2. **Solving the SDP**: This semidefinite program is then solved in polynomial time to obtain a set of vectors $\mathbf{u}_i$.

3. **Random Hyperplane Rounding**: After obtaining the vectors, a random hyperplane passing through the origin is chosen. This hyperplane is used to partition the vertices. Specifically, each vertex i is assigned to set S if $\mathbf{u}_i$ lies on one side of the hyperplane and to $\overline{S}$ if it lies on the other side. A random vector $\mathbf{r}$ is chosen uniformly from the unit sphere, and the partition is determined by the sign of the scalar product $\mathbf{u}_i \cdot \mathbf{r}$.

$$S = \{i \colon \mathbf{u}_i \cdot \mathbf{r} \geq 0\}, \quad \overline{S} = \{i \colon \mathbf{u}_i \cdot \mathbf{r} < 0\}$$

2.4.3 Correctness

Theorem 2.4.1. The Goemans-Williamson algorithm is a 0.878-approximation algorithm for the max-cut problem.

Proof. We analyze the expected value of the cut produced by the algorithm compared to the optimal cut value.

Let $\mathbf{u}_i$ be the unit vectors obtained by solving the SDP. Consider a random vector $\mathbf{r}$ uniformly distributed on the unit sphere. Define the cut S by:

$$S = \{i \colon \mathbf{u}_i \cdot \mathbf{r} \geq 0\}, \quad \overline{S} = \{i \colon \mathbf{u}_i \cdot \mathbf{r} < 0\}$$

For an edge (i,j), let X_{ij} be an indicator random variable that is 1 if i and j are on different sides of the cut and 0 otherwise. The probability that i and j are on different

sides is given by:

$$\mathbb{P}(X_{ij} = 1) = \mathbb{P}((\mathbf{u}_i \cdot \mathbf{r})(\mathbf{u}_j \cdot \mathbf{r}) < 0)$$

This probability can be computed using the fact that $\mathbf{r}$ is uniformly distributed over the unit sphere. The angle θ between $\mathbf{u}_i$ and $\mathbf{u}_j$ determines this probability:

$$\mathbb{P}(X_{ij} = 1) = \frac{\theta}{\pi}$$

where θ is the angle between $\mathbf{u}_i$ and $\mathbf{u}_j$.

Since $\cos(\theta) = \mathbf{u}_i \cdot \mathbf{u}_j$, we have $\theta = \arccos(\mathbf{u}_i \cdot \mathbf{u}_j)$. Thus,

$$\mathbb{P}(X_{ij} = 1) = \frac{\arccos(\mathbf{u}_i \cdot \mathbf{u}_j)}{\pi}$$

The expected weight of the cut produced by the algorithm is:

$$\mathbb{E}\left[\sum_{(i,j)\in E} w(i,j) \cdot X_{ij}\right] = \sum_{(i,j)\in E} w(i,j) \cdot \mathbb{P}(X_{ij} = 1) = \sum_{(i,j)\in E} w(i,j) \cdot \frac{\arccos(\mathbf{u}_i \cdot \mathbf{u}_j)}{\pi}$$

To relate this to the optimal SDP value, recall the SDP objective:

$$\sum_{(i,j)\in E} w(i,j) \cdot \left(\frac{1 - \mathbf{u}_i \cdot \mathbf{u}_j}{2}\right)$$

We use the inequality $\frac{\arccos x}{\pi} \geq \frac{1-x}{2}$ for $x \in [-1, 1]$ (Lemma 3.4 [GW95]), which holds because the function $\frac{\arccos x}{\pi}$ is convex and lies above the line $\frac{1-x}{2}$. Thus,

$$\sum_{(i,j)\in E} w(i,j) \cdot \frac{\arccos(\mathbf{u}_i \cdot \mathbf{u}_j)}{\pi} \geq \sum_{(i,j)\in E} w(i,j) \cdot \left(\frac{1 - \mathbf{u}_i \cdot \mathbf{u}_j}{2}\right)$$

Let $\mathrm{OPT}_{\mathrm{SDP}}$ denote the optimal value of the SDP. Then,

$$\mathbb{E}\left[\sum_{(i,j)\in E} w(i,j) \cdot X_{ij}\right] \geq \alpha \cdot \mathrm{OPT}_{\mathrm{SDP}}$$

for $\alpha = \frac{2}{\pi} \min\left(\frac{\theta}{1-\cos(\theta)}\right)$, where $0 \leq \theta \leq \pi$. Thus, the approximation ratio α is:

$$\alpha \approx 0.878.$$

□

2.4.4 Time complexity

Formulating the SDP

Formulating the SDP involves constructing the block matrix C, the linear constraints b, and the constraint matrix. This preparation step involves the structure of the graph and takes $O(|V|^2)$ time.

Solving the SDP

Solving the SDP is the most computationally intensive part of the algorithm. While solving an SDP exactly is generally hard, for our problem, we use efficient polynomial-time interior-point methods. According to [GL83], and supported by our analysis, the complexity of solving the SDP using interior-point methods can be bounded by $O(|V|^3 \cdot \log(1/\epsilon))$, where ϵ is the desired accuracy of the solution.

Random Hyperplane Rounding

After solving the SDP, the rounding step involves generating a random unit vector and partitioning the vertices based on their scalar product with this vector. Both generating the random vector and computing the scalar products for all vertices take $O(|V|^2)$ time.

Overall Time Complexity

Combining all the steps, the overall time complexity of the Goemans-Williamson algorithm can be approximated to $O(|V|^3 \cdot \log(1/\epsilon))$, since solving the SPD dominates the running time for large values of $|V|$.

Chapter 3

Min-Cut

3.1 Hao-Orlin algorithm

3.1.1 Idea

The innovation of the Hao-Orlin algorithm lies in its efficient use of multiple max-flow computations to identify the global minimum cut in the graph. By calculating multiple s-t cuts, it effectively explores various partitions of the graph to ensure the identification of the true minimum cut. That is why, this algorithm leverages the classic max-flow min-cut theorem, which states that the maximum flow between two nodes in a network is equal to the capacity of the minimum cut separating those nodes. For more details on the equivalence of min-cut and max-flow, we refer to Section 1.4.

At a high level, the algorithm works by iteratively solving a series of max-flow problems. It begins by selecting an arbitrary node s as the source. The algorithm then repeatedly selects another node t from the remaining nodes and computes the maximum flow from s to t. The capacity of the resulting s-t cut is recorded. By systematically varying the sink node t and solving up to $2|V| - 2$ max-flow problems, the algorithm ensures that it has considered all potential cuts.

3.1.2 Implementation

The implementation of the Hao-Orlin algorithm is designed to iteratively determine the minimum cut in a directed graph. The key idea is to progressively build a set S of nodes, starting from a single source node, and to compute the maximum flow to various sink nodes, thereby identifying potential minimum cuts.

Once a new sink node t' is selected, the implementation uses any max flow algorithm (i.e. `MaxFlowT` sub-procedure) to compute the maximum flow from the source node (node 0) to t'. The result of this computation is the capacity of the cut that separates the set S from the rest of the graph. This cut value is then compared to the current `minCutValue`, and if it is smaller, then it is updated.

The following code snippet illustrates this implementation:

```
void run() {
    minCutValue = INT_MAX;
    minCutSet.assign(graph->numberOfNodes(), false);
    std::vector<bool> visited(graph->numberOfNodes(), false);
    visited[0] = true;
    std::vector<int> S = {0};
    while (S.size() < graph->numberOfNodes()) {
        int t_prime = -1;
        for (int i = 0; i < graph->numberOfNodes(); ++i) {
            if (!visited[i]) {
                t_prime = i;
                break;
            }
        }
        MaxFlowT minCutSolver(*graph, 0, t_prime);
        minCutSolver.run();
        double currentMinCutValue = minCutSolver.getFlowSize();
        minCutValue = std::min(minCutValue, currentMinCutValue);
        visited[t_prime] = true;
        S.push_back(t_prime);
    }
}
```

3.1.3 Correctness

The correctness of the Hao-Orlin algorithm is rooted in the equivalence between the max-flow and min-cut problems, which is more thoroughly explored in Section 1.4. This theorem guarantees that finding the maximum flow between any two nodes s and t in a flow network will also give the minimum s-t cut. To identify the global minimum cut of the network, multiple executions of the max-flow algorithm are necessary. We claim, that instead of examining each pair of i.e. $\binom{|V|}{2}$ pairs in total, we need only $2|V| - 2$ pairs.

From Lemma 1.4.2, we get that we do not need to check every (s, t) pair for s-t min-cut, but only consider pairs of vertices that define uniquely separating cuts. Taking that into account we can choose u to be a fixed node and t' which represents all other nodes. From that we cover all values of s-t min-cuts by only using $2 \cdot (|V| - 1)$ pairs.

Now we want to make sure we find the global minimum cut with our algorithm. Therefore, let (S^*, T^*) be the optimal global minimum cut. Without general loss we can claim that $u \in S^*$ and due to fact that $T \neq \emptyset$, there is t' such that $t' \in T^*$. By the max-flow min-cut theorem and Lemma 1.4.2, the maximum flow $f^*(u, t')$ between u and t' equals the capacity of minimum u-t' cut for all $t' \neq u$. Therefore, especially we will get pair of (u, t') such that:

$$\text{val}(f^*(u, t')) = c(S^*, T^*)$$

By repeatedly applying the max-flow algorithm to identify u-t' cuts for fixed u and respectively changing t', it guarantees that the global minimum cut of the network is found.

3.1.4 Time complexity

The Hao-Orlin algorithm requires solving up to $2|V| - 2$ max-flow problems. This upper bound comes from the process of iterating over different sink nodes t while keeping a fixed source node s. Each max-flow computation, therefore, contributes to the overall time complexity.

By combining these factors, the overall time complexity of the Hao-Orlin algorithm can be expressed as:

$$O((2|V| - 2) \cdot T_{\text{max-flow}})$$

where $T_{\text{max-flow}}$ is the time complexity of the chosen max-flow algorithm which can vary, however using the Edmonds-Karp algorithm we get $O(|V| \cdot |E|^2)$ [EK72]. More advanced max-flow algorithms, such as Orlin's algorithm which can handle it in $O(|V| \cdot |E|$ [Orl13], giving us the overall complexity of:

$$O((2|V| - 2) \cdot (|V| \cdot |E|)) = O(|V|^2 \cdot |E|)$$

3.2 Stoer-Wagner algorithm

3.2.1 Idea

The Stoer-Wagner algorithm is a pivotal algorithm in graph theory, specifically designed to find the minimum cut in an *undirected graph* (Definition 1.1.1). The Stoer-Wagner algorithm is built on the concept of successively finding and contracting the most tightly connected (in terms of the total weight of edges between them) pairs of vertices until the entire graph is reduced to a single vertex. The intuition behind this method involves repeatedly identifying the smallest cuts in progressively contracted graphs, which ensures that no minimum cut is overlooked.

Graph Contraction Graph contraction is a technique where two vertices u and v are merged into a single vertex, s_{uv}. All edges previously incident to u or v are now incident to s_{uv}, and any parallel edges between u and v are summed up. This contraction process reduces the complexity of the graph, making it simpler to identify cuts as the algorithm proceeds.

3.2.2 Implementation

The `run` function orchestrates the overall execution of the Stoer-Wagner algorithm. It begins by preparing the graph's nodes and edges for processing. The core of this function is a loop that continues contracting the graph until only one vertex remains. Within each iteration of this loop, the function calls `minCutPhase` to determine the minimum cut for

the current state of the graph. By repeatedly finding and applying these minimum cuts, the algorithm ensures that the global minimum cut is identified. The value of the minimum cut found in each phase is compared and updated to keep track of the smallest cut found during the entire process.

```
void StoerWagnerMinCut::run() {
    std::vector<node> localVertices(graph->numberOfNodes());
    std::vector<std::vector<double>> edges(graph->numberOfNodes(),
            std::vector<double>(graph->numberOfNodes(), 0));
    minCutValue = INT_MAX;
    graph->forNodes([&](node u) {
        localVertices[u] = u;
    });
    graph->forEdges([&](node u, node v, double w) {
        edges[u][v] = w;
        edges[v][u] = w;
    });
    while (localVertices.size() > 1) {
        minCutValue = std::min(minCutValue, minCutPhase(localVertices, edges));
    }
}
```

The `minCutPhase` function is responsible for executing one phase of the Stoer-Wagner algorithm, which involves finding a minimum cut within a subset of the graph. The function starts by determining the connectivity weights of the vertices, identifying the most tightly connected vertex at each step. It iteratively selects the vertex with the highest connectivity to the growing set of selected vertices. This selection process continues until only one vertex remains unselected. At this point, the weight of the last added vertex represents the minimum cut for this phase.

The function then updates the graph by merging the last added vertex with its predecessor in the sequence, adjusting the edge weights accordingly. This merging (or contraction) of vertices simulates the reduction of the graph, ensuring that all possible cuts are considered in subsequent iterations. By updating the weights and repeating the selection process, the algorithm guarantees that no potential minimum cut is overlooked.

```
double StoerWagnerMinCut::minCutPhase(std::vector<node>& vertices,
        std::vector<std::vector<double>>& edges) {
    int size = vertices.size();
    std::vector<double> weightsPhase(size, 0);
    std::vector<bool> added(size, false);
    int prev, last = 0;
    for (int i = 0; i < size; ++i) {
        prev = last, last = -1;
        for (int j = 0; j < size; ++j) {
```

```
            if (!added[j] && (last == -1 || weightsPhase[j] > weightsPhase[last])) {
                last = j;
            }
        }
        if (i == size - 1) {
            double minCut = weightsPhase[last];
            for (int j = 0; j < size; ++j) {
                if (j != last) {
                    edges[vertices[prev]][vertices[j]] +=
                                    edges[vertices[last]][vertices[j]];
                    edges[vertices[j]][vertices[prev]] +=
                                    edges[vertices[j]][vertices[last]];
                }
            }
            vertices.erase(vertices.begin() + last);
            return minCut;
        }
        added[last] = true;
        for (int j = 0; j < size; ++j) {
            if (!added[j]) {
                weightsPhase[j] += edges[vertices[last]][vertices[j]];
            }
        }
    }
    return INT_MAX;
}
```

3.2.3 Correctness

The algorithm is based on two main principles: the preservation of a global minimum cut through edge contractions and the accurate identification of the minimum cut during each phase. Below, we provide a proof of correctness, demonstrating that the algorithm reliably finds the minimum cut.

Theorem 3.2.1. The global minimum cut is preserved through edge contraction.

Proof. Let us apply Definition 1.1.11 to an undirected graph $G = (V, E, w)$. The capacity of a cut (S, T) is given by:

$$w(S,T) = \sum_{(u,v)\in E: u\in S, v\in T} w(u,v).$$

where the *minimum cut* is the cut (S, T) that minimizes $w(S, T)$ (Definition 1.1.14).

During the execution of the Stoer-Wagner algorithm, vertices are iteratively contracted. Let G' be the graph obtained after contracting an edge (u, v) in G. The key property is

that any cut in G' corresponds to a cut in G, and vice versa, with the same cut weight.

If u and v are in the same part of the cut (i.e., both in S or both in T), contracting u and v into a single vertex s_{uv} does not change the cut value. This is because the edge (u, v) is internal to S or T and does not cross the cut. Formally, if $u, v \in S$, then after contraction, the new vertex s_{uv} will still be in S, and the cut value remains the same:

$$w(S, T) = w((S \cup \{s_{uv}\}) \setminus \{u, v\}, T).$$

If u and v are in different parts of the cut (i.e., $u \in S$ and $v \in T$), the contraction must ensure that the sum of the weights of the edges crossing the cut remains unchanged. When u and v are contracted into a single vertex s_{uv}, the weight of the edge connecting s_{uv} to any other vertex x is the sum of the weights of the edges connecting u and v to x. Specifically, the new weight $w(s_{uv}, x)$ is defined as follows:

- If x was a neighbor of both u and v, then:

$$w(s_{uv}, x) = w(u, x) + w(v, x).$$

- If x was a neighbor of only one of u or v, then (without loss of generality we assume it was u):

$$w(s_{uv}, x) = w(u, x).$$

Thus, the contraction process preserves the minimum cut property, as the cut weight remains unchanged whether u and v are in the same or different parts of the cut. □

Theorem 3.2.2. The algorithm correctly finds the minimum cut during each phase.

Proof. Let A be the set of vertices selected in a phase, and let v be the last vertex added to A. The cut $(A \setminus \{v\}, \{v\})$ is identified as a candidate for the minimum cut. The correctness of this identification relies on the property that the last added vertex, being the most tightly connected to the rest, represents a critical point for evaluating the cut weight.

To formally prove the correctness, we use mathematical induction on the number of vertices n in the graph. We claim that for any graph with k vertices (where $k \leq n$), the Stoer-Wagner algorithm correctly identifies the minimum cut.

For $n = 2$, the algorithm trivially identifies the cut between the two vertices, which is the only possible cut and thus the minimum cut.

Assume that the algorithm correctly identifies the minimum cut for any graph with k vertices, where $k < n$. Now, Consider a graph G with n vertices. During a phase, the algorithm contracts the graph to G' with $n - 1$ vertices. By the induction hypothesis, the algorithm correctly finds the minimum cut in G'. Since the contraction preserves the minimum cut property, the identified cut in G' corresponds to the minimum cut in G.

By iteratively applying this process, the algorithm eventually reduces the graph to two vertices, ensuring that the minimum cut identified in each phase is correct. Therefore, by mathematical induction, the Stoer-Wagner algorithm correctly finds the minimum cut in any undirected, weighted graph. □

Overall, the Stoer-Wagner algorithm systematically reduces the graph while preserving the minimum cut properties, ensuring that all potential cuts are considered. By leveraging the properties of edge contraction and maximum adjacency, the algorithm guarantees that the smallest cut is found efficiently and accurately. This rigorous approach underpins the correctness of the Stoer-Wagner algorithm.

3.2.4 Time Complexity

The time complexity of the Stoer-Wagner algorithm is an important aspect to consider, as it impacts the algorithm's efficiency and practicality for large graphs. The algorithm's time complexity can be analyzed by examining the key operations performed during its execution.

Lemma 3.2.1. The single pass of the `minCutPhase` function takes $O(|V|^2 + |E|)$ time.

Proof. Selection of the most tightly connected vertex involves finding the vertex with the maximum weight among those not yet added to the growing set. This requires scanning all vertices, resulting in a time complexity of $O(|V|)$ for each selection. Since this selection is performed $|V|$ times in each phase, the total time complexity for this operation is $O(|V|^2)$.

After selecting a vertex, the edge weights are updated to reflect the new connectivity. This involves updating the weights for all edges incident to the selected vertex. The time complexity for updating the edge weights is $O(|E|)$. □

Lemma 3.2.2. The overall time complexity of the Stoer-Wagner algorithm is $O(|V|^3 + |V| \cdot |E|)$.

Proof. First observe that the `run` function performs $|V| - 1$ iterations, as in each iteration, one vertex is contracted. Then by combining this observation with Lemma 3.2.1 we get that overall time complexity is:

$$O(|V| \cdot (|V|^2 + |E|)) = O(|V|^3 + |V| \cdot |E|).$$

□

It is worth noting that the time complexity of the `minCutPhase` function can be improved to $O(|E| + |V| \log |V|)$ by using a heap (priority queue) to manage the selection of the vertex that has the largest sum of edge weights connecting it [SW97]. This optimization leverages the efficient logarithmic time complexity for insertion and extraction operations provided by the heap data structure.

3.3 Karger algorithm

3.3.1 Idea

Karger's algorithm is a randomized algorithm designed to find the minimum cut in an unweighted graph. The algorithm leverages randomness to repeatedly contract edges until

only two vertices remain, at which point the remaining edges between these two vertices represent a cut in the original graph. The key insight is that, by contracting edges in a certain manner, the probability of retaining a minimum cut in the contracted graph is sufficiently high, making the algorithm efficient in finding such cuts with high probability.

The process begins with an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. The algorithm proceeds by randomly selecting an edge $(u, v) \in E$ and contracting it, effectively merging the vertices u and v into a single vertex while preserving the multiplicity of edges. This contraction reduces the number of vertices by one and the total number of edges is adjusted accordingly. The process is repeated until only two vertices remain.

Formally, the contraction of an edge (u, v) involves the following steps:

- Merge vertices u and v into a single vertex w.
- Replace all edges (u, x) and (v, x) with edges (w, x) for all $x \in V \setminus \{u, v\}$.
- Remove self-loops, i.e., edges of the form (w, w).

3.3.2 Implementation

The `run()` function is responsible for executing the algorithm multiple times and recording the smallest cut value encountered. This repetition leverages the probabilistic nature of Karger's algorithm, which increases the likelihood of identifying the true minimum cut through multiple independent trials. The function iterates over a predetermined number of repetitions, each time invoking `runOnce` to perform a single iteration of the algorithm. The smallest cut value found across all iterations is stored as the final result.

```
1 void KargerMinCut::run() {
2     double bestMinCutValue = INT_MAX;
3     for (int i = 0; i < repeat; ++i) {
4         runOnce();
5         bestMinCutValue = std::min(bestMinCutValue, minCutValue);
6     }
7     minCutValue = bestMinCutValue;
8 }
```

The `runOnce` function embodies the core of Karger's algorithm, executing the edge contraction process until only two vertices remain. Initially, the function sets up the required data structures, including the parent and rank vectors used for the union-find operations [GF64], which help efficiently manage the merging of vertex subsets.

The algorithm begins by calculating the total weight of all edges and selecting edges randomly based on their weights. This weighted random selection ensures that edges with higher weights have a proportional chance of being selected, which is crucial for maintaining the algorithm's probabilistic guarantees.

The main loop continues contracting edges until only two vertices are left. During each iteration, a random edge is selected, and its endpoints' subsets are identified using the `find` function. If the selected edge connects two different subsets, the subsets are merged using the `unionSub` function, and the number of vertices is decremented. The weight of the selected edge is also subtracted from the total weight to maintain the correct probabilities for subsequent selections.

```
void KargerMinCut::runOnce() {
    int vertices = graph->numberOfNodes();
    std::vector<int> parent(vertices), rank(vertices, 0);
    minCutValue = 0;
    std::random_device rd;
    std::mt19937 gen(rd());
    double totalWeight = 0;
    std::vector<WeightedEdge> edges;
    graph->forEdges([&](node u, node v, edgeweight weight) {
        edges.emplace_back(u, v, weight);
        totalWeight += weight;
    });
    std::uniform_real_distribution<> dis(0.0, totalWeight);
    for (int i = 0; i < graph->numberOfNodes(); ++i) {
        parent[i] = i;
    }
    while (vertices > 2) {
        double pick = dis(gen);
        double cumulativeWeight = 0;
        int selectedEdge = -1;
        for (int i = 0; i < edges.size(); ++i) {
            cumulativeWeight += edges[i].weight;
            if (cumulativeWeight >= pick) {
                selectedEdge = i;
                break;
            }
        }
        int subset1 = find(parent, edges[selectedEdge].u);
        int subset2 = find(parent, edges[selectedEdge].v);
        if (subset1 != subset2) {
            unionSub(parent, rank, subset1, subset2);
            vertices--;
            totalWeight -= edges[selectedEdge].weight;
        }
    }
    graph->forEdges([&](node u, node v, edgeweight weight) {
```

```
            int subset1 = find(parent, u), subset2 = find(parent, v);
            if (subset1 != subset2) {
                minCutValue += weight;
            }
        });
}
```

3.3.3 Correctness and time complexity

The correctness of Karger's algorithm is rooted in its probabilistic nature, specifically the probability that the minimum cut survives through the random edge contractions. Here, we provide a formal proof of this probability and determine the number of iterations required to achieve at most $\frac{1}{|V|}$ chance of error. Additionally, we discuss the time complexity of the algorithm for both a single run and multiple iterations.

Karger's algorithm repeatedly contracts edges until only two vertices remain. Each contraction step merges two vertices into one, reducing the number of vertices while preserving the structure of the graph with a high probability of retaining the minimum cut.

Let $G = (V, E)$ be an undirected graph (Definition 1.1.1) with $|V|$ vertices and let C be a minimum cut in G. Initially, the probability of not selecting an edge from the minimum cut C in the first contraction is:

$$1 - \frac{|C|}{m}$$

For each subsequent contraction, the probability that the minimum cut survives is proportional. The overall probability that the minimum cut survives through $|V| - 2$ contractions is:

$$\prod_{i=0}^{n-3} \left(1 - \frac{2}{|V| - i}\right) = \frac{2}{|V| \cdot (|V| - 1)}$$

Thus, the probability of finding the minimum cut in one run of Karger's algorithm is $\frac{2}{|V| \cdot (|V|-1)}$. To increase the probability of finding the minimum cut, the algorithm is repeated multiple times. To achieve a high probability of finding the minimum cut, consider repeating the algorithm t times. The probability of failing to find the minimum cut in one run is $1 - \frac{2}{|V| \cdot (|V|-1)}$. The probability of failing in all t runs is:

$$\left(1 - \frac{2}{|V| \cdot (|V| - 1)}\right)^t$$

If we set $t \geq \frac{|V| \cdot (|V|-1) \cdot \ln(|V|)}{2}$ then we can obtain that

$$\left(1 - \frac{2}{|V| \cdot (|V| - 1)}\right)^t \leq \frac{1}{|V|}$$

Therefore, to achieve a $\frac{1}{|V|}$ probability of error, the algorithm should be repeated $O(|V|^2 \cdot \log |V|)$ times. This ensures a high probability of finding the minimum cut in the graph.

The time complexity of one run of Karger's algorithm can be analyzed based on the

steps involved in each contraction. The algorithm involves:

- Selecting a random edge: $O(|E|)$ time.
- Contracting the selected edge: $O(\log |V|)$ time, using the union-find data structure with path compression and union by rank.
- Updating the graph and edge weights: $O(|E|)$ time.

Since there are $|V| - 2$ contractions, the total time complexity for one run is $O(|V|^2)$

Given that the algorithm needs to be repeated $O(|V|^2 \cdot \log |V|)$ times to achieve a high probability of finding the minimum cut, the overall time complexity becomes $O(|V|^2 \cdot |V|^2 \cdot \log |V|) = O(|V|^4 \cdot \log |V|)$

It is noteworthy that our implementation currently operates in $O(|V|^2)$ time for each run. However, this can be optimized using Kruskal's algorithm for constructing minimum spanning trees. By leveraging Kruskal's algorithm, the contraction steps can be performed more efficiently, reducing the time complexity for each run to $O(|E| \log |E|)$ [Kru56]. In dense graphs, where $|E| = O(|V|^2)$, this can further be improved to $O(|E|)$ time using more advanced data structures [Kar93]. Consequently, the overall time complexity for multiple iterations becomes $O(|V|^2 \cdot \log |V|) \cdot O(|E| \cdot \log |E|) = O(|V|^2 \cdot |E| \cdot \log |V| \log |E|)$ or even, this could be further reduced to $O(|V|^2 \cdot |E| \cdot \log |V|)$.

In summary, with the proper settings, Karger's algorithm is both correct with high probability and computationally feasible. The probabilistic guarantee is ensured by multiple independent iterations, while the time complexity remains manageable for practical applications. By repeating the algorithm $O(|V|^2 \cdot \log |V|)$ times, we achieve a high probability of identifying the minimum cut, making Karger's algorithm a robust and efficient solution for the minimum cut problem in undirected graphs.

3.4 Karger-Stein algorithm

0

3.4.1 Idea

Karger-Stein algorithm is an extension of Karger's original randomized min-cut algorithm. The primary goal of both algorithms is to find the minimum cut in an undirected graph, but Karger-Stein version improves upon the basic method by incorporating a more sophisticated recursive approach that enhances the probability of finding the true minimum cut.

The innovative aspect of Karger-Stein algorithm lies in its recursive structure. Instead of contracting edges until only two vertices are left, the algorithm splits the graph into smaller subproblems. Specifically, it randomly selects an edge and contracts it, then applies the algorithm recursively to the resulting graph. This splitting is done until the graph is sufficiently small (a threshold defined by the algorithm), at which point a simpler base

case method, such as Karger's original algorithm, is applied to find the minimum cut of the smaller graphs.

Here is a step-by-step breakdown of the algorithm's idea:

1. **Recursive Splitting:** Begin with the original graph G. Randomly choose an edge e and contract it, resulting in a smaller graph G'.

2. **Recursive Call:** Recursively apply Karger-Stein algorithm to G'. This recursion continues, effectively breaking down the problem into smaller subproblems.

3. **Threshold Base Case:** Once the graph G' reaches a predetermined size threshold, switch to a simpler min-cut algorithm (e.g., Karger's original algorithm) to find the minimum cut.

4. **Combining Results:** Compare the minimum cuts obtained from the recursive calls and select the smallest one as the final result.

The recursive nature of the algorithm significantly boosts the probability of accurately finding the minimum cut compared to the original algorithm. By leveraging multiple recursive calls and combining their results, Karger-Stein approach mitigates the randomness of single edge contractions and achieves a more reliable outcome.

In essence, Karger-Stein algorithm elegantly combines randomness with recursion to enhance the efficiency and reliability of finding the minimum cut in undirected graphs. This blend of probabilistic and recursive techniques forms the core intuition behind the algorithm and underpins its effectiveness in solving the min-cut problem.

3.4.2 Implementation

First, the `run` function oversees the overall execution of the algorithm. It repeatedly calls the `runOnce` function to perform the algorithm multiple times, as each run provides a probabilistic chance of identifying the minimum cut. By running the algorithm several times, we ensure a higher probability of finding the optimal minimum cut.

```
void KargerSteinMinCut::run() {
    double bestMinCutValue = INT_MAX;
    for (int i = 0; i < repeat; ++i) {
        runOnce();
        bestMinCutValue = std::min(bestMinCutValue, minCutValue);
    }
    minCutValue = bestMinCutValue;
}
```

Next, the `recursiveMinCut` function embodies the recursive nature of Karger-Stein algorithm. It takes the current number of vertices as an argument and applies the recursive contraction method until the graph is sufficiently small. This function starts by checking if the number of vertices is less than or equal to a small threshold (equal to 6 in this case).

If so, a simpler `standardMinCut` algorithm is applied. Otherwise, the function calculates a threshold T based on the current number of vertices. The threshold T is determined as the current number of vertices divided by the square root of 2, ensuring a gradual reduction in graph size while preserving the structure necessary for accurate min-cut determination.

This function initiates by setting up the union-find data structures to manage the contracted vertices. It then iteratively contracts edges chosen randomly, reducing the graph until the number of vertices is reduced to T. Each edge contraction merges two vertices into one, thereby decrementing the vertex count and updating the total weight of the graph. The algorithm continues contracting edges until the graph is small enough, at which point the recursive calls are made to further reduce the graph size to the threshold. By recursively applying the min-cut algorithm to smaller subgraphs and comparing the results, the algorithm ensures a higher probability of accurately determining the minimum cut.

```
int KargerSteinMinCut::recursiveMinCut(int currentV) {
    if (currentV <= 6) {
        return standardMinCut();
    }
    int T = static_cast<int>(std::ceil(currentV / std::sqrt(2)));
    std::vector<int> parent(currentV), rank(currentV, 0);
    for (int i = 0; i < currentV; ++i) {
        parent[i] = i;
    }
    std::random_device rd;
    std::mt19937 gen(rd());
    double totalWeight = 0;
    std::vector<WeightedEdge> weightedEdges;
    graph->forEdges([&](node u, node v, edgeweight weight) {
        weightedEdges.emplace_back(u, v, weight);
        totalWeight += weight;
    });
    int vertices = currentV;
    std::uniform_real_distribution<> dis(0.0, totalWeight);
    while (vertices > T) {
        double pick = dis(gen);
        double cumulativeWeight = 0;
        int selectedEdge = -1;
        for (int i = 0; i < weightedEdges.size(); ++i) {
            cumulativeWeight += weightedEdges[i].weight;
            if (cumulativeWeight >= pick) {
                selectedEdge = i;
                break;
            }
```

```
        }
        int subset1 = find(parent, weightedEdges[selectedEdge].u);
        int subset2 = find(parent, weightedEdges[selectedEdge].v);
        if (subset1 != subset2) {
            unionSub(parent, rank, subset1, subset2);
            vertices--;
            totalWeight -= weightedEdges[selectedEdge].weight;
        }
    }
    return std::min(recursiveMinCut(T), recursiveMinCut(T));
}
```

3.4.3 Correctness

Karger-Stein algorithm for finding the minimum cut of a graph is a probabilistic algorithm that builds on Karger's basic min-cut algorithm by recursively contracting edges in a graph. The correctness of the algorithm relies on the observation that the minimum cut survives in the contracted graph with high probability.

Lemma 3.4.1 (Lemma 4.3 from [KS96])**.** The probability that Karger-Stein algorithm finds the minimum cut is at least $\Omega\left(\frac{1}{\log|V|}\right)$.

By running the algorithm $\log^2|V|$ times, the probability of failing each time is:

$$\left(1-\frac{1}{\log|V|}\right)^{\log^2|V|}$$

We use for small x the approximation of $(1-x)^y \leq e^{-xy}$:

$$\left(1-\frac{1}{\log|V|}\right)^{\log^2|V|} \leq e^{-\log^2|V|/\log|V|} = e^{-\log|V|} = \frac{1}{|V|}$$

Thus, the probability that the algorithm fails every time is at most $\frac{1}{|V|}$. Therefore, the probability of at least one success after $\log^2|V|$ independent runs is at least $1-\frac{1}{|V|}$.

3.4.4 Time complexity

The time complexity of the Karger-Stein min-cut algorithm can be proved directly from the recursive nature of the algorithm. Each step involves contracting approximately half of the nodes in the graph. Let $T(|V|)$ be the time complexity for a graph with $|V|$ nodes. The recurrence relation for the Karger-Stein algorithm is:

$$T(n) = 2T\left(\frac{|V|}{\sqrt{2}}\right) + O(|V|^2)$$

This follows because the algorithm makes two recursive calls on a graph with $\frac{|V|}{\sqrt{2}}$ nodes, and each contraction step takes $O(|V|^2)$ time. Solving this recurrence relation using the

Master Theorem for divide-and-conquer recurrences gives us the overall time complexity of $T(|V|) = O(|V|^2 \cdot \log |V|)$.

In conclusion, the Karger-Stein algorithm correctly finds the minimum cut of a graph with high probability. By running the algorithm $O(\log^2 |V|)$ times, the probability of finding the minimum cut becomes $1 - O\left(\frac{1}{|V|}\right)$. The time complexity of each run of the algorithm is $O(|V|^2 \cdot \log |V|)$, making it efficient for practical purposes.

Chapter 4

Implementation and benchmark

4.1 Implementation

The algorithms discussed in this paper were implemented in C++. It utilizes the *NetworKit* and *Boost* frameworks for efficient handling of graph data structures and algorithmic operations. The source code for the implementation is publicly available in the **koala-networkit** github repository and can be accessed at `https://github.com/krzysztof-turowski/koala-networkit/pull/8`.

4.1.1 Implementation details

The core components of the implementation include several key classes and methods for handling the max-cut and min-cut problems. Below, we outline the primary files and their responsibilities:

Max-cut files

- `include/max_cut/MaxCut.hpp` – Main header file which defines MaxCut interface. Inherited by all max-cut algorithms.
- `src/max_cut/MaxCut.cpp` – implementation of base MaxCut class.
- `src/max_cut/GreedyMaxCut.cpp` – implementation of `greedy` algorithm (See Section 2.1).
- `src/max_cut/BranchAndBoundMaxCut.cpp` – implementation of `branch and bound` algorithm (See Section 2.2).
- `src/max_cut/RankTwoRelaxationMaxCut.cpp` – implementation of `Burer` algorithm (See Section 2.3).
- `src/max_cut/GoemansWilliamsonMaxCut.cpp` – implementation of `Goemans-Williamson` algorithm (See Section 2.4).

- `test/testMaxCut.cpp` – Unit tests for ensuring correctness of the implemented algorithms using the GoogleTest framework.
- `benchmark/benchmarkMaxCut.cpp` – Benchmarking programs to evaluate the performance of the algorithms on various graph instances.

Min-cut files

- `include/min_cut/MinCut.hpp` – Main header file which defines MinCut interface. Inherited by all min-cut algorithms.
- `include/min_cut/HaoOrlinMinCut.hpp` – implementation of `Hao-Orlin` algorithm (See Section 3.1).
- `src/min_cut/MinCut.cpp` – implementation of base MinCut class.
- `src/min_cut/StoerWagnerMinCut.cpp` – implementation of `Stoer-Wagner` algorithm (See Section 3.2).
- `src/min_cut/KargerMinCut.cpp` – implementation of `Karger` algorithm (See Section 3.3).
- `src/min_cut/KargerSteinMinCut.cpp` – implementation of `Karger-Stein` algorithm (See Section 3.4).
- `test/testMinCut.cpp` – Unit tests for ensuring correctness of the implemented algorithms using the GoogleTest framework.
- `benchmark/benchmarkMinCut.cpp` – Benchmarking programs to evaluate the performance of the algorithms on various graph instances.

4.1.2 Correctness and testing

The correctness of each implemented algorithm was rigorously tested against known benchmarks and exhaustive algorithms for small and medium sized graph instances.

- **Unit Tests:** Implemented using GoogleTest, these tests cover all edge cases and ensure that the algorithms correctly find max-cut and min-cut solutions for various types of graphs.
- **Benchmark Graphs:** The benchmarking suite includes both randomly generated graphs and standard graph instances from established repositories, providing a comprehensive evaluation of algorithm performance.

4.1.3 Benchmarking and performance evaluation

The algorithms were benchmarked on a variety of graph instances to assess their performance. Graphs with varying sizes and densities were used to determine the scalability and efficiency of each algorithm.

Some of randomized graphs were generated using the Python 3 library `NetworkX`, which provides tools for creating and manipulating complex networks. Each data point in the performance evaluation represents the average result of 5-10 runs to ensure statistical significance.

To evaluate the performance of the algorithms, a diverse set of instances was utilized. These instances were sourced from two comprehensive libraries: the Bonn University Quadratic Programming Library (`http://bqp.cs.uni-bonn.de/library/html/instances.html`) and the Min-Cut/Max-Flow Problem Instances Library from the Technical University of Denmark (`https://data.dtu.dk/articles/dataset/`).

4.2 Benchmark

4.2.1 Max-cut

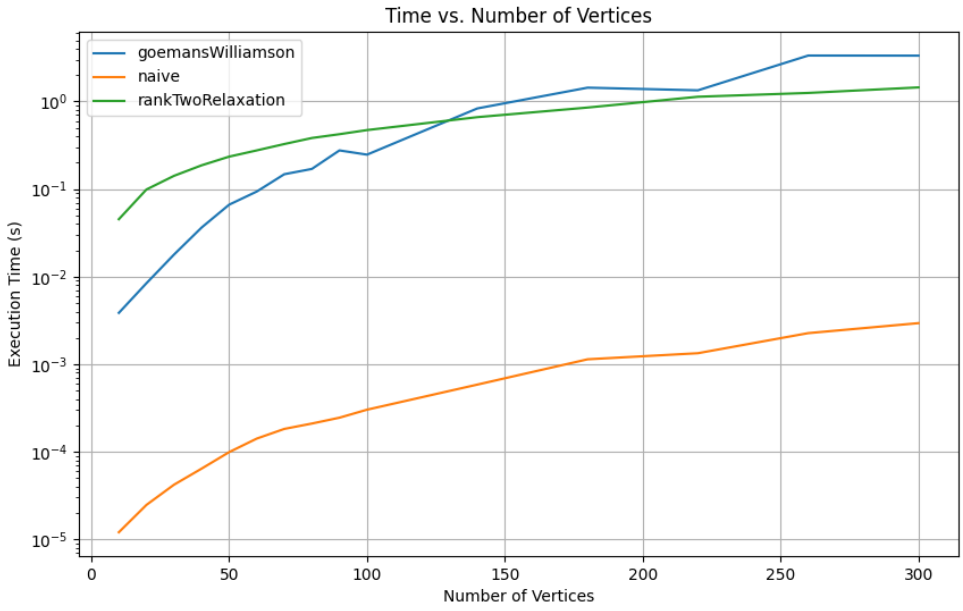

Figure 4.1: Randomized graphs with edges chosen uniformly.

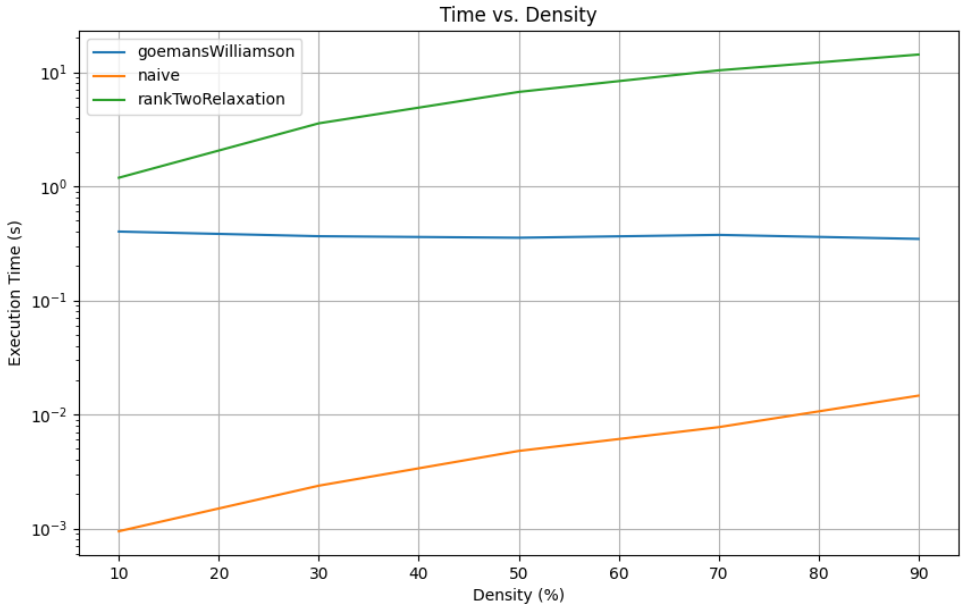

Figure 4.2: Randomized graphs with edges chosen uniformly ($|V| = 100$).

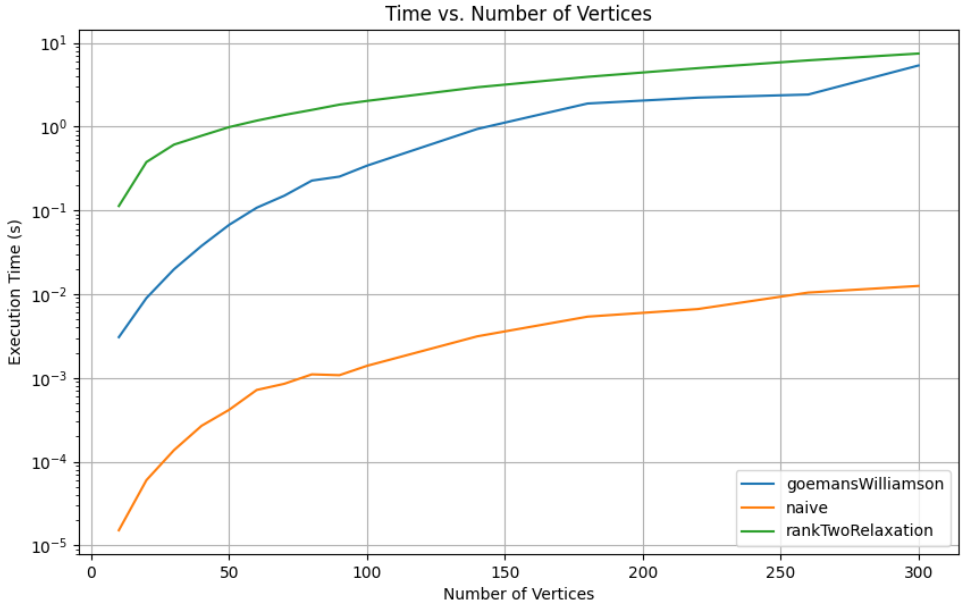

Figure 4.3: Randomized graphs with edges chosen uniformly ($|E| = 8 \cdot |V|$).

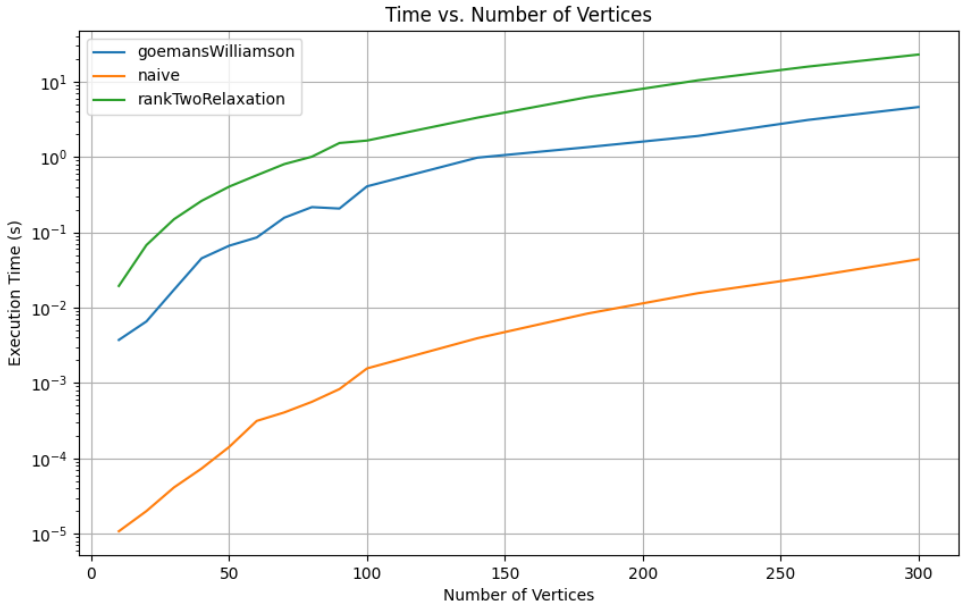

Figure 4.4: Randomized graphs with edges chosen uniformly ($|E| = \frac{|V|^2}{16}$).

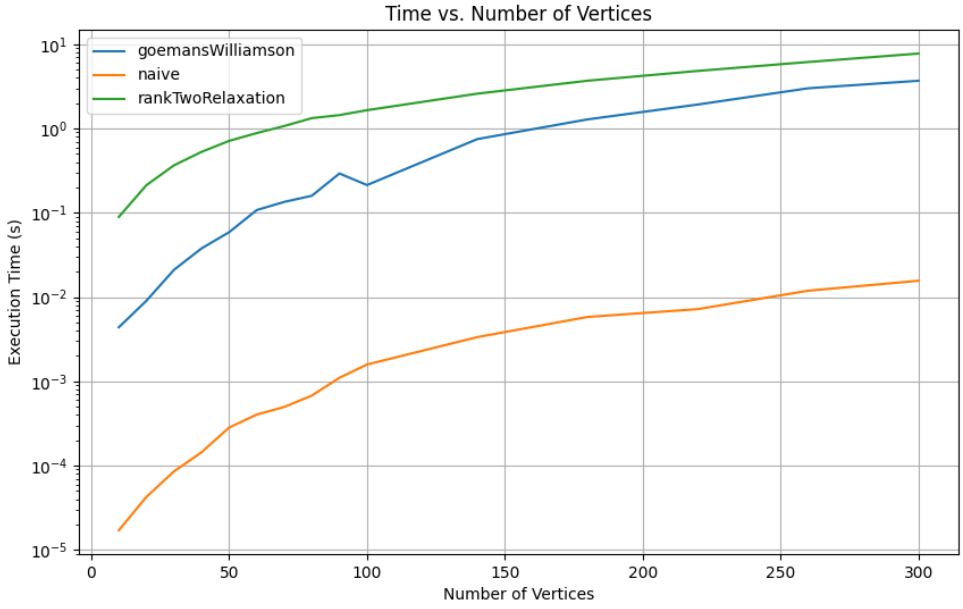

Figure 4.5: Randomized graphs with edges chosen uniformly ($|E| = |V| \cdot \log |V|$).

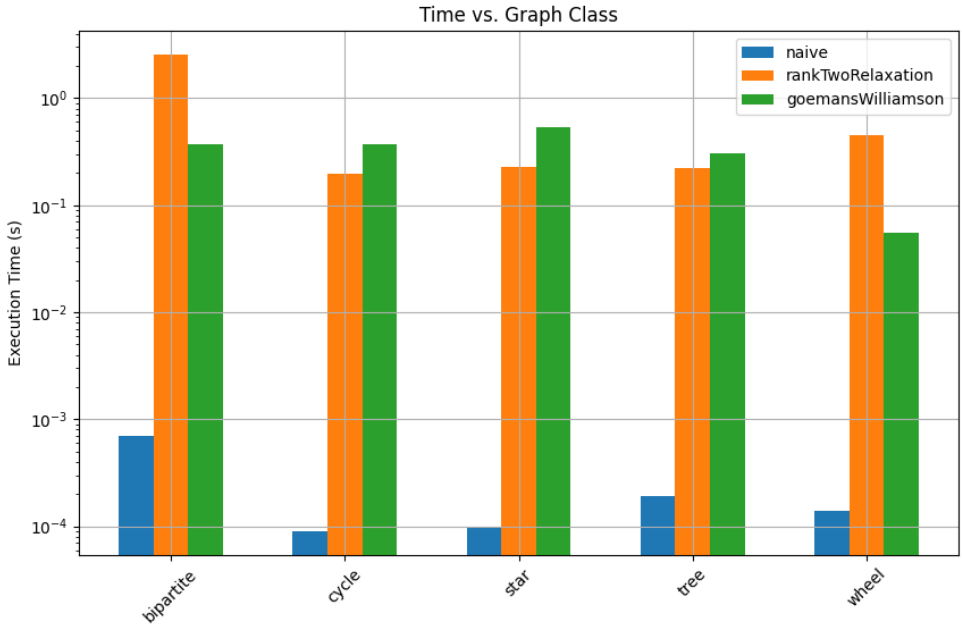

Figure 4.6: Randomized graphs with edges chosen uniformly ($|V| = 100$).

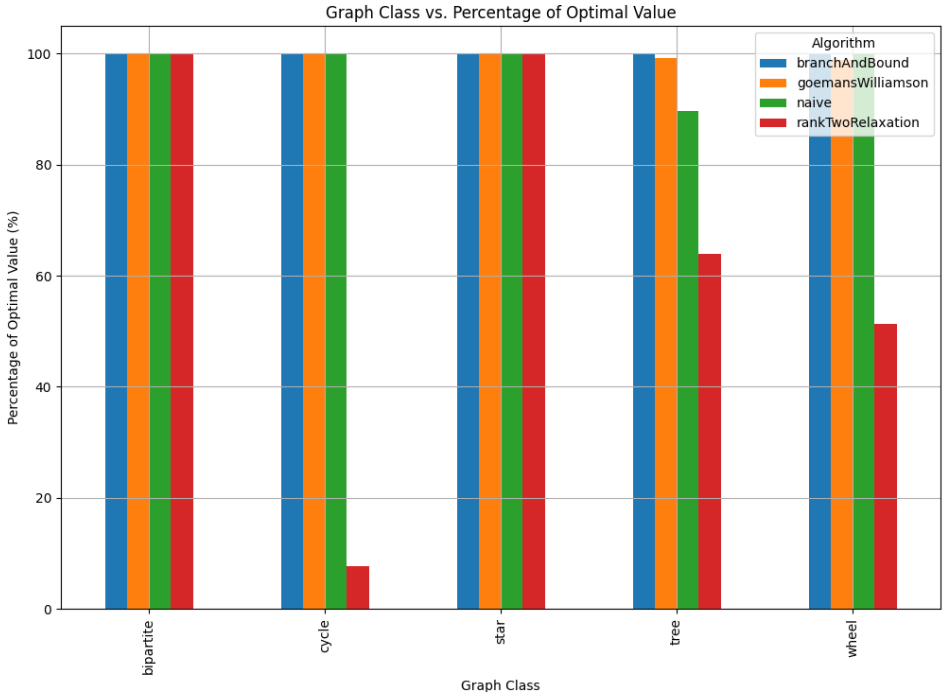

Figure 4.7: Randomized graphs with edges chosen uniformly ($|V| = 100$).

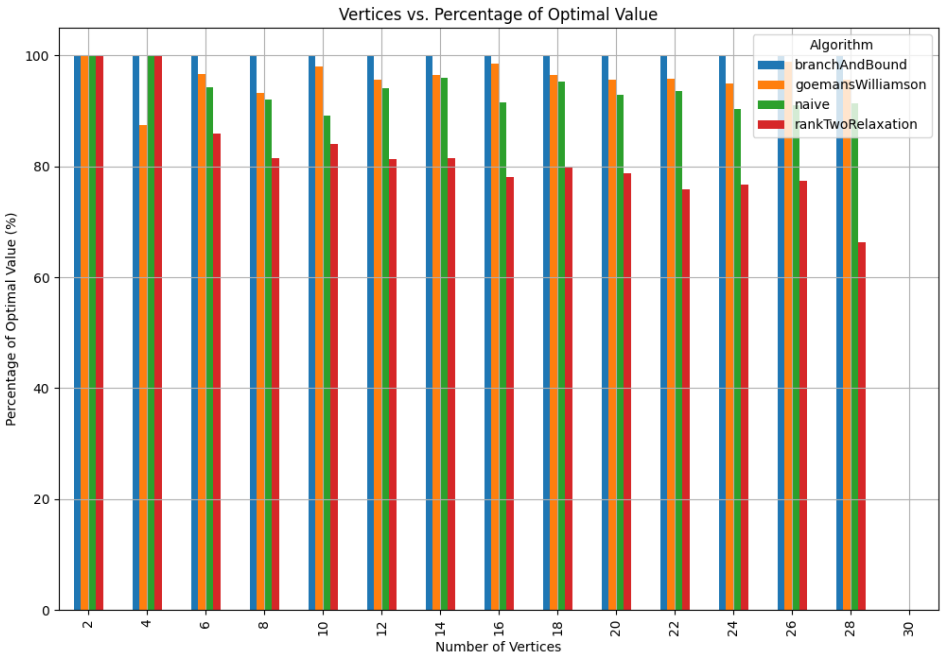

Figure 4.8: Randomized graphs with edges chosen uniformly.

Table 4.1: Max-cut instances benchmark results (time in seconds)

Instance	**Greedy**		**Burer**		**Goemans-Williamson**	
	Result	**Time**	**Result**	**Time**	**Result**	**Time**
be250	2.19×10^5	1.66×10^{-2}	1.71×10^5	1.18×10^1	2.14×10^5	4.67×10^{-1}
bqp250	4.15×10^5	2.05×10^{-2}	3.91×10^4	1.12×10^1	4.06×10^5	6.02×10^{-1}
deconv3x3	1.42×10^7	2.66×10^{-1}	1.16×10^5	1.42×10^2	1.89×10^7	1.99×10^2
hassan	5.41×10^6	1.89×10^{-4}	9.30×10^5	7.56×10^{-1}	5.02×10^6	3.46×10^{-2}
house	2.32×10^5	3.27×10^{-5}	3.43×10^4	2.51×10^{-1}	1.96×10^5	1.22×10^{-2}
HR_G1	8.93×10^5	6.55×10^{-1}	2.16×10^5	1.16×10^2	8.88×10^5	1.67×10^1
matching	2.61×10^6	1.91×10^{-4}	7.33×10^5	1.18×10^{-2}	2.78×10^6	6.44×10^{-3}
motor	2.91×10^5	1.88×10^{-4}	1.62×10^4	9.92×10^{-1}	2.67×10^5	6.99×10^{-2}
sg3dl	4.72×10^4	8.07×10^{-2}	3.18×10^3	9.72	4.79×10^4	6.13×10^1
superRes	1.82×10^9	1.02×10^1	1.08×10^8	1.88×10^2	1.91×10^9	3.11×10^2

Summary

From the benchmarks performed, we observe that among the approximate algorithms, Goemans-Williamson and the naive algorithms gives the best results. However, as the number of vertices increases, the performance gap between these two algorithms rises. The

runtime of the Goemans-Williamson and Burer algorithms are comparable to each other and become extremely slow for graphs with more than 10000 vertices. Consequently, for very large graphs, it is more efficient to use Karger's algorithm. On the other hand, for smaller graphs, the Goemans-Williamson algorithm is a prior choice. The time complexities and outcomes are consistent with our expectations based on the analysis.

4.2.2 Min-cut

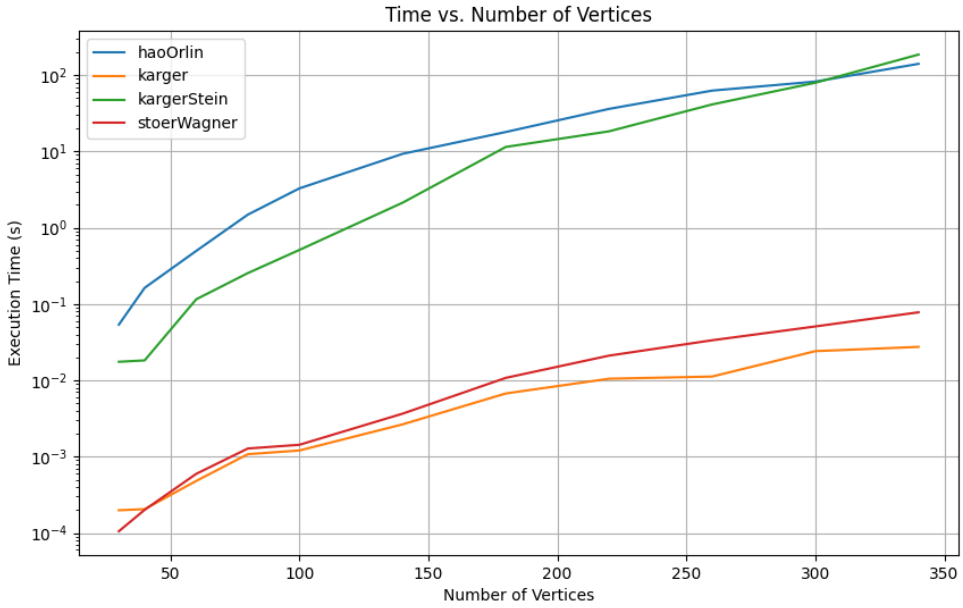

Figure 4.9: Randomized graphs with edges chosen uniformly.

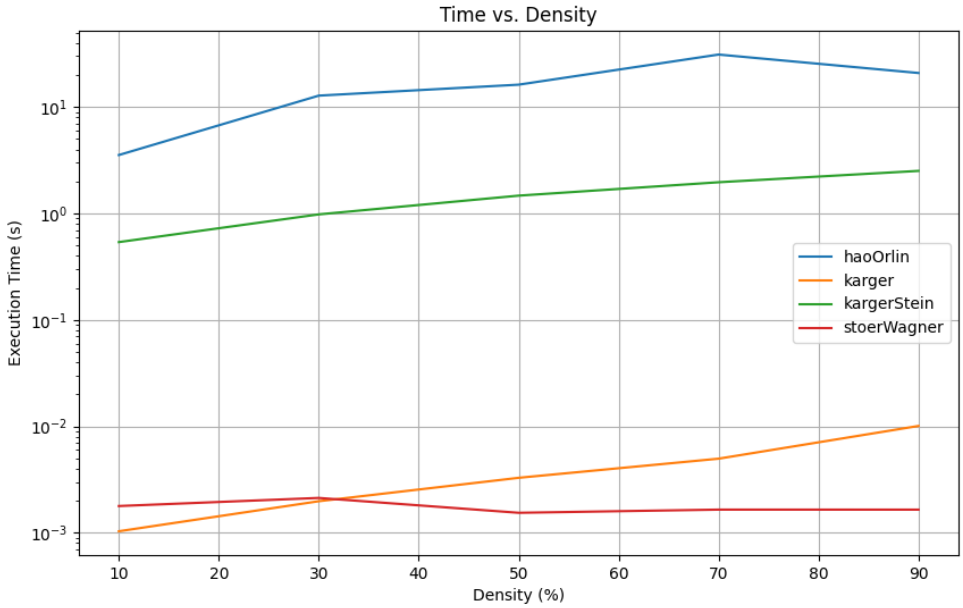

Figure 4.10: Randomized graphs with edges chosen uniformly ($|V| = 100$).

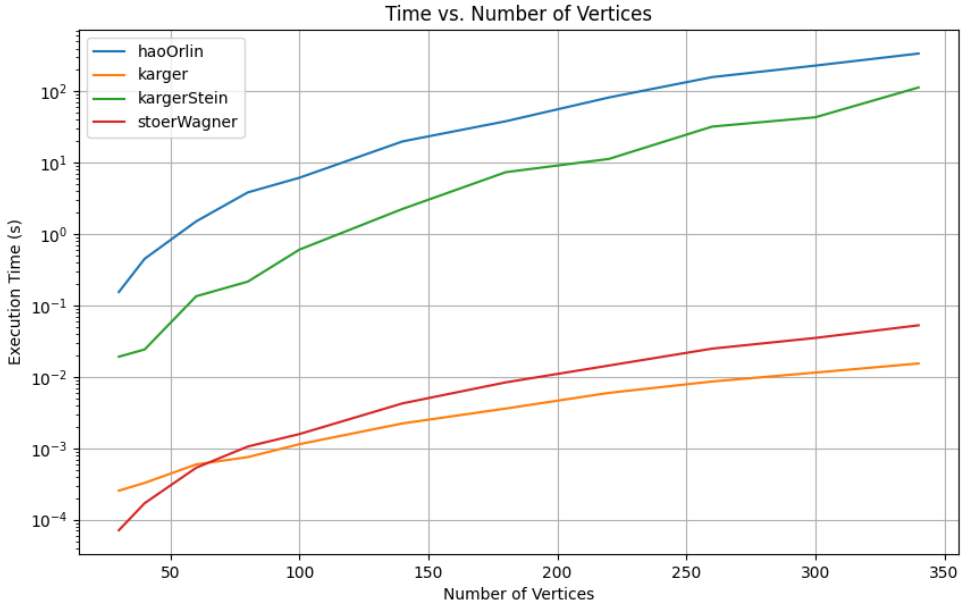

Figure 4.11: Randomized graphs with edges chosen uniformly ($|E| = 8 \cdot |V|$).

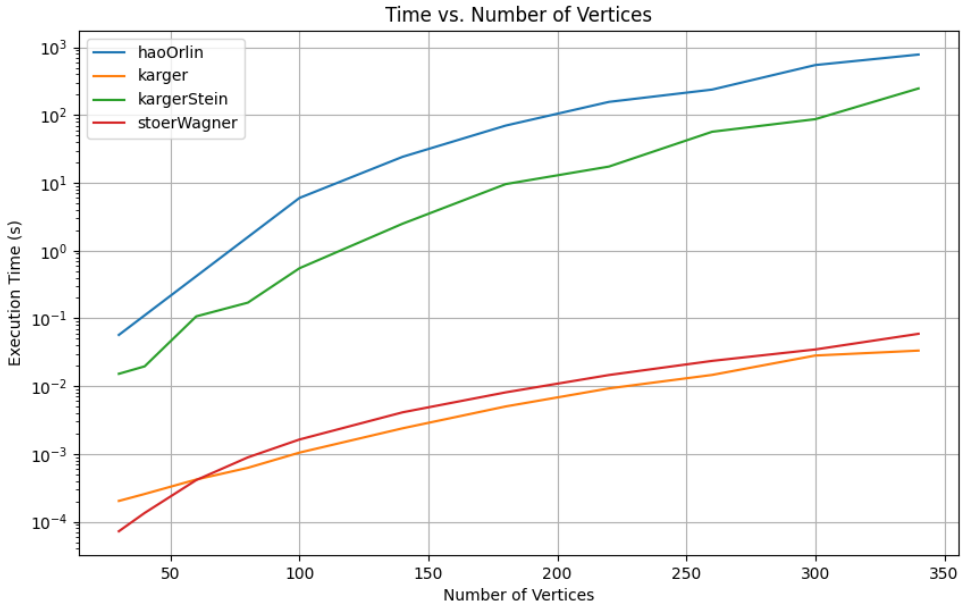

Figure 4.12: Randomized graphs with edges chosen uniformly ($|E| = \frac{|V|^2}{16}$).

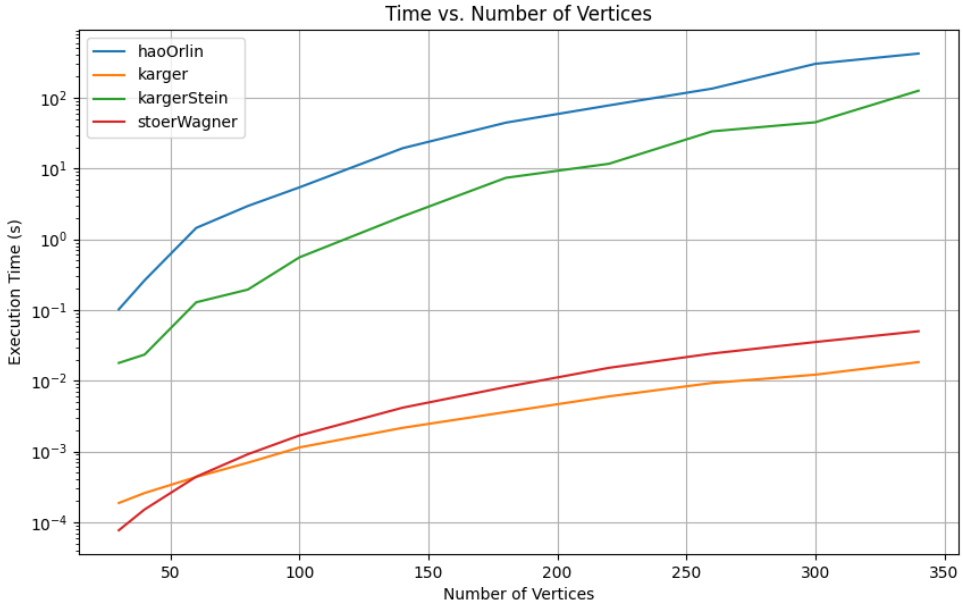

Figure 4.13: Randomized graphs with edges chosen uniformly ($|E| = |V| \cdot \log |V|$).

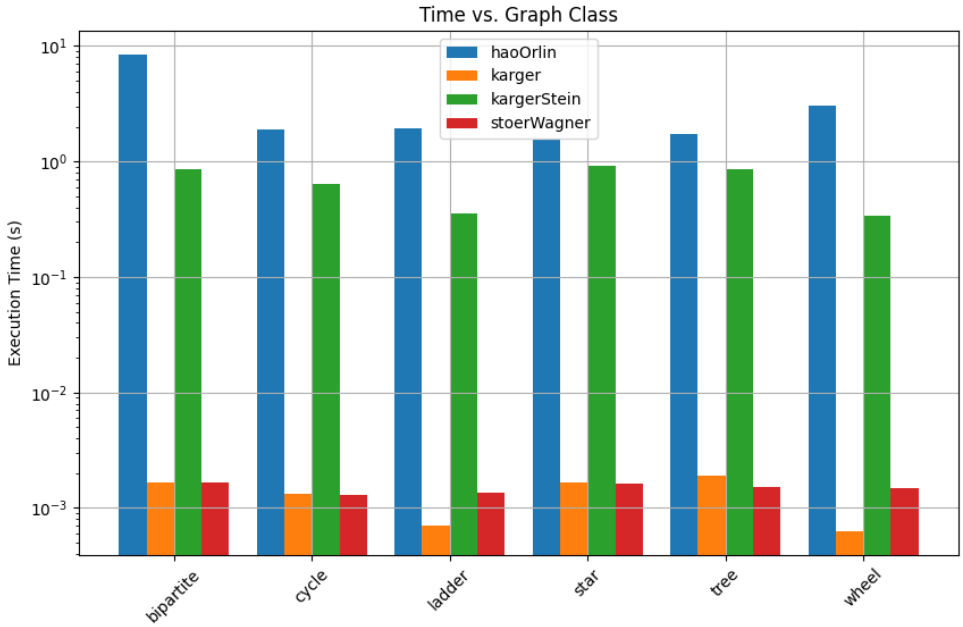

Figure 4.14: Randomized graphs with edges chosen uniformly ($|V| = 100$).

Table 4.2: Min-cut instances benchmark results (time in seconds)

Instance	**Hao-Orlin**		**Karger**	
	Result	**Time**	**Result**	**Time**
be250	1.42×10^{3}	2.19×10^{2}	1.42×10^{3}	1.29×10^{-2}
bqp250	2.41×10^{3}	2.83×10^{2}	2.41×10^{3}	1.00×10^{-2}
matching	6.02×10^{4}	2.42×10^{-1}	6.02×10^{4}	2.72×10^{-4}
motor	3.00×10^{1}	2.33	3.00×10^{1}	1.23×10^{-2}
deconv_graph3x3	5.38×10^{3}	7.24×10^{3}	5.86×10^{3}	1.13×10^{1}
HR_G1	2.13×10^{3}	8.13×10^{3}	2.69×10^{3}	2.07×10^{-1}
sg3dl	8.40×10^{1}	7.61×10^{3}	8.40×10^{1}	5.79×10^{-2}

Instance	**Karger-Stein**		**Stoer-Wagner**	
	Result	**Time**	**Result**	**Time**
be250	1.42×10^{3}	4.55×10^{1}	1.42×10^{3}	1.83×10^{-2}
bqp250	2.41×10^{3}	3.96×10^{1}	2.41×10^{3}	1.88×10^{-2}
matching	6.02×10^{4}	2.28×10^{-2}	6.02×10^{4}	1.30×10^{-4}
motor	3.00×10^{1}	3.13	3.00×10^{1}	7.37×10^{-4}
deconv_graph3x3	5.57×10^{3}	5.23×10^{3}	5.38×10^{3}	1.05×10^{1}
HR_G1	2.13×10^{3}	7.16×10^{3}	2.13×10^{3}	6.22×10^{-1}
sg3dl	8.40×10^{1}	4.15×10^{3}	8.40×10^{1}	1.23

Summary

From the benchmarks conducted on the min-cut algorithms, we observe that the Stoer-Wagner and Karger algorithms are the fastest. All algorithms produced approximately correct results, showcasing that the numerous iterations performed by Karger and Karger-Stein algorithms indicated correct results. The time complexities and outcomes are consistent with our expectations based on the analysis.

Bibliography

[Bie10] Daniel Bienstock. "Optimal Power Flow, Quadratic Programming, and Locational Marginal Prices". In: *Operations Research* 58.3 (2010), pp. 745–756.

[BMZ02] Samuel Burer, Renato Monteiro, and Yin Zhang. "Rank-Two Relaxation Heuristics for MAX-CUT and Other Binary Quadratic Programs". In: *SIAM Journal on Optimization* 12.2 (2002), pp. 503–521.

[Bol98] Béla Bollobás. *Modern Graph Theory*. New York: Springer, 1998.

[Cor+22] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2022.

[CR93] Sunil Chopra and Ramachandra Rao. "The Partition Problem". In: *Mathematical Programming* 59.1 (1993), pp. 87–115.

[DF13] Rodney Downey and Michael Fellows. *Fundamentals of Parameterized Complexity*. London: Springer, 2013.

[EK72] Jack Edmonds and Richard Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *Journal of the ACM* 19.2 (1972), pp. 248–264.

[FF56] Lester Ford and Delbert Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[GF64] Bernard Galler and Michael Fischer. "An Improved Algorithm for Transitive Closure on a Directed Graph". In: *Communications of the ACM* 7.5 (1964), pp. 301–303.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman and Company, 1979.

[GL83] Gene Golub and Charles Van Loan. *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press, 1983.

[GM12] Bernd Gärtner and Jiří Matoušek. *Approximation Algorithms and Semidefinite Programming*. Berlin-Heidelberg: Springer, 2012.

[GW95] Michel Goemans and David Williamson. "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". In: *Journal of the ACM* 42.6 (1995), pp. 1115–1145.

[Hås01] Johan Håstad. "Some Optimal Inapproximability Results". In: *Journal of the ACM* 48.4 (2001), pp. 798–859.

[HO94] James Hao and James Orlin. “A Faster Algorithm for Finding the Minimum Cut in a Graph”. In: *SIAM Journal on Computing* 23.4 (1994), pp. 661–673.

[Kar72] Richard Karp. “Reducibility Among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by Raymond Miller and James Thatcher. Springer, 1972, pp. 85–103.

[Kar93] David Karger. “Global Min-Cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm”. In: *SODA ’93: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 1993, pp. 21–30.

[Kho+07] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O’Donnell. “Optimal Inapproximability Results for MAX-CUT and Other 2-Variable CSPs?” In: *SIAM Journal on Computing* 37.1 (2007), pp. 319–357.

[Kru56] Joseph Kruskal. “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem”. In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50.

[KS96] David Karger and Clifford Stein. “A New Approach to the Minimum Cut Problem”. In: *Journal of the ACM* 43.4 (1996), pp. 601–640.

[LP12] Frauke Liers and Gregor Pardella. “Partitioning planar graphs: a fast combinatorial approach for max-cut”. In: *Computational Optimization and Applications* 51.1 (2012), pp. 323–344.

[ŁS11] Jakub Łącki and Piotr Sankowski. “Min-Cuts and Shortest Cycles in Planar Graphs in $O(n \log \log n)$ Time”. In: *Algorithms - ESA 2011: 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011, Proceedings*. Ed. by Camil Demetrescu and Magnús Halldórsson. Vol. 6942. Lecture Notes in Computer Science. Springer, 2011, pp. 155–166.

[Orl13] James Orlin. “Max Flows in $O(nm)$ Time, or Better”. In: *STOC ’13: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. ACM, 2013, pp. 765–774.

[PY91] Christos Papadimitriou and Mihalis Yannakakis. “Optimization, Approximation, and Complexity Classes”. In: *Journal of Computer and System Sciences* 43 (1991), pp. 425–440.

[SG76] Sartaj Sahni and Teofilo Gonzalez. “P-Complete Approximation Problems”. In: *Journal of the ACM* 23.3 (1976), pp. 555–565.

[SW97] Mechthild Stoer and Frank Wagner. “A Simple Min-Cut Algorithm”. In: *Journal of the ACM* 44.4 (1997), pp. 585–591.