Engineering Fast Maximum Flows to Speed Up Critical Weighted Independent Set Reductions

Markus Everling

September 25, 2025

4205481

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisors:
Dr. Ernestine Großmann
Dr. Kenneth Langedal

Acknowledgments

I would like to express my gratitude to Prof. Dr. Schulz, for providing me the opportunity to write this thesis under his supervision. I would especially like to thank Ernestine Großmann and Kenneth Langedal, who provided excellent guidance and advice, as well as quick feedback throughout the project, which was invaluable to its success. Finally, I would like to thank my friends and family, for proofreading large parts of this thesis, and for their unending support throughout my studies.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, September 25, 2025

M. Everling

Markus Everling

Abstract

The Maximum Weight Independent Set (MWIS) problem is a fundamental NP-hard problem with numerous real world applications. Given an undirected graph with vertex weights, the problem is to find a subset of the vertices of maximum weight, such that no two of the vertices are adjacent. The MWIS problem also has many closely related graph problems, such as the Minimum Weight Vertex Cover problem and Maximum Weight Clique problem.

To effectively solve large input instances, solvers for the MWIS problem employ data reductions, which attempt to reduce the size of the instance to be solved, while preserving optimality of the constructed solution. One powerful but computationally expensive data reduction is the Critical Weighted Independent Set reduction, which identifies a critical weighted set on the input graph by computing a maximum flow on a specially constructed flow network. From this critical weighted set it then computes a critical weighted independent set, of which all vertices can then be removed from the input graph.

In this work, we implement different maximum flow algorithms into the MWIS solver KaMIS, and compare them with regards to suitability for the CWIS reduction. In addition to algorithmic changes, we also introduce optimizations related to flow graph data structures, as well as the construction of flow networks.

Contents

CHAPTER 1

Introduction

The MAXIMUM WEIGHT INDEPENDENT SET (MWIS) problem is a fundamental NP-hard problem in graph theory. Given an undirected graph with vertex weights, the problem is to find a subset of the vertices of maximum weight, such that no two vertices in the set are adjacent.

State-of-the-art solvers employ a number of different data reductions to minimize the solution space that they need to search. One powerful but computationally expensive data reduction for the MWIS problem is the Critical Weighted Independent Set (CWIS) reduction, which computes a maximum flow on a specially constructed flow graph, from which it identifies a critical weighted independent set that is then reduced away in the input graph.

In this work, we discuss the CWIS reduction in detail, compare different maximum flow algorithms with regard to their suitability for use in the CWIS reduction, and employ various optimizations to further improve the reduction's performance.

1.1 Motivation

The MWIS problem has many closely related graph problems, with a wide range of practical applications. One such problem is the MINIMUM WEIGHT VERTEX COVER (MWVC) problem, in which the goal is to find a subset of vertices of minimum weight, such that every edge is incident to at least one vertex in the subset. Solving the MWIS problem is considered equivalent to solving the MWVC problem, as for any MWIS $I \subseteq V$ on a graph $G = (V, E)$, the set $V \setminus I$ is a MVWC. Another related problem is the MAXIMUM WEIGHT CLIQUE (MWC) problem, in which the goal is to find a subset of maximum weight, such that all vertices in the subset are pairwise adjacent. Solving the MWC problem on a graph G is equivalent to solving the MWIS problem on its complement graph $\overline{G}$. However, solving the MWIS problem using a MWC solver is often impractical, as large real world graphs are usually very sparse, so constructing the dense complement graph can quickly run into memory limitations.

In practice, these problems are used to optimize vehicle routing [17], for network security simulations [20], and in network performance measurements [42]. While this is by no means an exhaustive list of applications, it should be clear that there is a lot of interest in efficient solvers for the MWIS problem, which can then be used to solve the other related problems too.

Due to the computational complexity of the MWIS problem, it is often infeasible to run an exact solver directly on large graphs. Instead, state-of-the-art solvers employ many different data reductions, which try to reduce the size of a graph in such a way that given a MWIS of the reduced graph, one can efficiently reconstruct a MWIS of the original graph.

One powerful such reduction is the Critical Weighted Independent Set (CWIS) reduction, which finds a CWIS, and commits it to the MWIS solution. Internally, this is done by computing a maximum flow in a specially constructed flow network, and identifying vertices in the CWIS on its residual graph. Since this is computationally expensive compared to other simpler reductions, the entire reduction pipeline can be sped up significantly with the right flow algorithm and careful optimizations.

KAMIS [1] is a state-of-the-art MWIS solver, and the first MWIS solver to implement the CWIS reduction. Though many of our experiments are agnostic toward any specific solver, we use KAMIS for a baseline implementation of the CWIS reduction.

1.2 Our Contribution

In this work, we provide a full self-contained description of the CWIS reduction, explanation of the algorithm used, and a proof of correctness of said algorithm. Furthermore, we implement different maximum flow algorithms into the KAMIS solver and compare them with regards to their suitability for the CWIS reduction. More specifically:

- We implement Tidal Flow [21] and Dinitz's Algorithm [15] in KAMIS. The MWIS solver in KAMIS utilizes data reductions including the CWIS reduction, previously implemented with the Push-Relabel maximum flow algorithm. We integrate two different maximum flow algorithms into the existing KAMIS codebase, namely Tidal Flow and Dinitz's Algorithm (discussed in Sections 3.2 and 3.1). These implementations are designed as a drop-in replacement for the existing Push-Relabel implementation, to be minimally intrusive in the codebase and to allow for easy comparison between the algorithms.

- We optimize the backing flow graph data structure used by them, minimizing memory indirections and accelerating flow graph construction. While the new flow graph is only used in our Dinitz implementation, it could be used as a drop-in replacement in the old Push-Relabel code, or in any future code operating on flow graphs.

- We compare the different max-flow algorithms with regards to the CWIS reduction. We provide timing comparisons for many different input graphs of different sizes. These timings include the runtime of just the CWIS reduction as well as the runtime of the entire reduction pipeline. In addition to timings for our implementations, we also compare to the Push-Relabel implementation from the Boost library [7]. In both of the experiments, our Dinitz implementation with our new flow graph comes out on top, leading to large speedups over the existing baseline code.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 lays out definitions used throughout the other chapters, including definitions related to graphs, to the MWIS problem and to flow networks. Chapter 3 gives an overview of the three relevant maximum flow algorithms, namely Dinitz's Algorithm, Tidal Flow, and Push-Relabel. It also discusses the structure of MWIS solvers, and lists known solvers using the CWIS reduction along with their choice of flow algorithm used in the CWIS reduction implementation. Chapter 4 defines the CWIS reduction and gives a description of the algorithm used to implement the reduction. It also provides detailed proofs for the correctness of the reduction and of the algorithm. Chapter 5 gives further details about the implementation of Tidal Flow and Dinitz's Algorithm into KaMIS, and lists and explains the optimizations we introduced. Chapter 6 presents experimental evaluations, comparing the different implementations of the CWIS reduction, both in isolation and in the context of the full KaMIS reduction pipeline. Finally, Chapter 7 discusses our findings, and lists suggestions for future work related to the CWIS reduction.

CHAPTER 2

Fundamentals

In this chapter, we provide definitions which are used throughout the other chapters. We give basic definitions related to graphs in Section 2.1, as well as definitions relevant to the MWIS problem in Section 2.2. Section 2.3 defines data reductions in the context of the MWIS problem. Finally, Section 2.4 covers flow networks and concepts related to them, which are relevant to the Critical Weighted Independent Set reduction introduced in Chapter 4. Given the definitions provided in this chapter, no previous knowledge, other than basic mathematical notation, should be required to read the rest of the thesis.

2.1 General Definitions

An *undirected graph* $G = (V, E)$ consists of a finite vertex set V and an edge set E, such that each $e \in E$ is a 2-element subset of V. We denote the number of vertices of G as $|V|$ and its number of edges as $|E|$. By convention we also refer to $|V|$ as n and to $|E|$ as m, unless specified otherwise. For vertices $u, v \in V$, we say u is *adjacent* to v, if $\{u, v\} \in E$. For a vertex u and an edge e, we say u is *incident* to e (and e is incident to u) if $u \in e$. By this definition, a graph cannot contain any parallel edges (distinct edges incident to one pair of vertices) or self-loops (edges that are only incident to one vertex).

Given a vertex v, its *neighborhood* $N(v)$ is the set of vertices adjacent to v, i.e.

$$N(v) := \{u \in V : \{u, v\} \in E\}.$$

The neighborhood of a set of vertices $U \subseteq V$ is the union of its elements' neighborhoods:

$$N(U) := \bigcup_{v \in U} N(v)$$

Note that U may overlap with $N(U)$. For convenience, we also define the shorthand

$$N^*(U) := N(U) \setminus U.$$

We call a graph $G' = (V', E')$ a *subgraph* of G, denoted by $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. For a vertex set $U \subseteq V$, we also define its *induced subgraph* of G, denoted by $G[U]$:

$$G[U] \coloneqq (U, \{\{u, v\} \in E : u, v \in U\})$$

Put into words, $G[U]$ is the maximal subgraph of G with vertex set U, containing all edges that have both endpoints in U.

For a vertex $v \in G$, we define its *degree* in G as

$$\deg(v) \coloneqq |\{e \in E : v \in e\}| = |N(v)|$$

i.e. the number of edges incident to v.

A *path* $P = (v_1, ..., v_k)$ on G is a sequence of vertices, such that $\{v_i, v_{i+1}\} \in E$ for all $i \in \{1, \ldots, k-1\}$, and $|\{v_1, \ldots, v_k\}| = k$, i.e. no vertex may appear twice in one path. For vertices $u, v \in V$, we call a path starting at u and ending at v a u-v path. We say a graph is *connected*, if there exists a u-v path for any two vertices $u, v \in V$.

A *vertex weight function* $w : V \to \mathbb{R}_{>0}$ is a function that maps to each vertex in G a positive weight. For a subset $U \subseteq V$, we denote the weight of U with regards to w as

$$w(U) \coloneqq \sum_{u \in U} w(u).$$

A *directed graph* $G = (V, E)$ is defined similarly to an undirected graph, except that its edge set consists of ordered pairs of vertices, i.e. all edges have the shape $e = (u, v)$ with $u, v \in V$ and $u \neq v$. As in undirected graphs, parallel edges and self-loops (edges of the shape (v, v)) are not allowed, but pairs of opposing edges are, so for vertices $u, v \in V$, both (u, v) and (v, u) may lie in E. Subgraphs and induced subgraphs, paths, and connectedness are also defined analogously to undirected graphs.

In directed graphs, the concept of a degree is replaced by the concept of indegree and outdegree; we define the indegree of a vertex u as

$$\deg_{\text{in}}(u) \coloneqq |\{v \in V : (v, u) \in E\}|$$

and its outdegree as

$$\deg_{\text{out}}(u) \coloneqq |\{v \in V : (u, v) \in E\}|$$

i.e. the indegree and outdegree describe the number of incoming and outgoing edges in u respectively.

By convention, when talking about graphs without qualifying directness, we mean undirected graphs, unless the context dictates directed graphs.

2.2 Independent Sets

Let $G = (V, E)$ be an undirected graph. We call a set of vertices $I \subseteq V$ an *independent set* in G, if there exists no edge $\{u, v\} \in E$ such that $u, v \in I$. An independent set $I \subseteq V$ is called *maximum*, if no larger independent set exists in G, i.e. if

$$|I| = \max\{|J| : J \subseteq V \text{ is independent in } G\}.$$

Given a vertex weight function $w : V \to \mathbb{R}_{>0}$, we can generalize this notion of a maximum independent set: We call an independent set $I \subseteq V$ a *maximum weight independent set* (MWIS), if no independent set with larger weight exists, i.e. if

$$w(I) = \max\{w(J) : J \subseteq V \text{ is independent in } G\}.$$

Maximum independent sets are not to be confused with *maximal* independent sets; the latter is any independent set $I \subseteq V$ such that no proper supersets $J \supsetneq I$ are independent, i.e. I is maximally independent if adding any vertex to it would break its independence. While a maximum independent set is trivially also maximal, not every maximal independent set is also maximum.

2.3 Data Reductions

Computing a MWIS of a weighted graph (and consequently, a MIS on an unweighted graph), has been shown to be NP-hard. To find exact solutions on large graphs, a solver first has to try to efficiently minimize the size of the instance to be solved, before running an exhaustive search on the reduced instance.

Data Reductions are rules that specify different ways to minimize input graphs, while keeping the ability to reconstruct a MWIS solution to the original graph, given a solution for the reduced graph. A data reduction consists of two algorithms, called REDUCEGRAPH and RESTORESOLUTION.

REDUCEGRAPH takes in an input graph G, and tries to reduce it to a smaller graph G', by removing vertices or edges, or folding multiple vertices or edges together. If it is able to reduce the graph, it returns the reduced graph G' as output. Otherwise, it signals that G is irreducible according to the rules of this reduction.

RESTORESOLUTION takes in a MWIS I' of the reduced graph G', and reconstructs from it a MWIS I of the original graph G. Given an oracle MWIS that is able to choose a MWIS for any input graph, it is therefore guaranteed that the equality

$$w(\textsc{Mwis}(G)) = w(\textsc{RestoreSolution}(\textsc{Mwis}(\textsc{ReduceGraph}(G))))$$

holds for any graph G, regardless of which MWIS the oracle chooses for both the original and the reduced instance.

Iteratively running a selected list of such data reductions, until none can reduce the graph further, yields a (locally) maximally reduced instance to be solved by an exact solver. While many such reductions exist, in this thesis the focus lies on the Critical Weighted Independent Set (CWIS) reduction, which is discussed in Chapter 4.

2.4 Flow Networks

A flow network F is defined as a 4-tuple (G, c, s, t), consisting of a connected directed graph $G = (V, E)$, an edge capacity function $c : E \to \mathbb{R}_{\geq 0}$ and distinguished source and sink vertices $s, t \in V$. We additionally require that for any edge $(u, v) \in E$, its opposing edge (v, u) must also lie in E, and that exactly one of the two edges has nonzero capacity.

A *flow* on F is a function $f : E \to \mathbb{R}$ with three conditions: For any edge $(u, v) \in E$, its flow is bounded by its capacity, i.e.

$$f(u, v) \leq c(u, v),$$

its flow equals the negation of the flow on the opposing edge, i.e.

$$f(u, v) = -f(v, u),$$

and for any vertex $v \in V \setminus \{s, t\}$, the amount of incoming flow must equal the amount of outgoing flow, i.e.

$$\sum_{(u,v)\in E} f(u, v) - \sum_{(v,w)\in E} f(v, w) = 0.$$

The *value* $\text{val}(f)$ of the flow f is defined as the flow leaving the source, i.e.

$$\text{val}(f) = \sum_{(s,w)\in E} f(s, w).$$

A flow f is called *maximum*, if no flows exist on F with a higher flow value.

Intuitively, one can imagine a flow network as a network of pipes (modeled by the edges) and pipe intersections (modeled by the vertices). The amount of flow going through any given pipe is bounded by its capacity, and the flow value represents the total throughput of a given flow, with a maximum flow representing the most units of flow that can be pushed through the pipe network.

Numerous different algorithms to efficiently compute maximum flows have been devised since the problem's introduction by Ford and Fulkerson in 1956 [29]. We go into detail on relevant algorithms in Chapter 3, but many of them share common concepts, which are presented here.

2.4.1 Preflows

Some maximum flow algorithms do not always operate on a valid flow, but use a relaxed definition called a *preflow*. Introduced by Karzanov in [31], preflows are very similar to flows, but use a relaxed variation of the conservation constraint. For a preflow f, the following must hold for all vertices $v \in V \setminus \{s,t\}$:

$$\sum_{(u,v)\in E} f(u,v) - \sum_{(v,w)\in E} f(v,w) \geq 0$$

Put into words, in a preflow the amount of flow entering a vertex may exceed the amount of flow exiting it. To quantify this, we introduce the *excess function* $\chi_f : V \to \mathbb{R}_{\geq 0}$, defined as follows:

$$\chi_f(v) = \sum_{(u,v)\in E} f(u,v) - \sum_{(v,w)\in E} f(v,w)$$

2.4.2 Residual Graphs

Let $F = (G,c,s,t)$ be a flow network operating on the graph $G = (V,E)$, and let f be a flow or a preflow on F. For an edge $(u,v) \in E$, we define its *residual capacity* with regards to f to be

$$c_f(u,v) := c(u,v) - f(u,v).$$

The residual capacity of an edge represents the amount of additional flow that could be pushed through this edge, without breaking the capacity constraint.

We say an edge $(u,v) \in E$ is a *residual edge* if $c_f(u,v) > 0$, i.e. if some amount of flow can be pushed through it. We denote the set of all residual edges with regards to f as E_f. The *residual graph* $G_f = (V,E_f)$ is then defined as the graph with vertices V containing only the residual edges of the original network. Note that, because of the symmetry constraint that $f(u,v) = -f(v,u)$ for all $\{u,v\} \in E$, an edge with zero capacity can still be a residual edge, if $f(u,v) < 0$.

Many maximum flow algorithms operate on the residual graph instead of the flow network itself, as it reduces the data per edge from two values (flow and capacity) to just one (residual capacity).

2.4.3 Augmenting Paths

Let $F = (G,c,s,t)$ be a flow network operating on the graph $G = (V,E)$, and let f be a flow on F. An *augmenting path* $P = (v_1,...,v_k)$ is an s-t path on the residual graph G_f, i.e. $v_1 = s$ and $v_k = t$. The *value* of P is defined as

$$\text{val}(P) := \min_{i\in\{1,...,k-1\}} \{c_f(v_i,v_{i+1})\}$$

and describes the smallest residual capacity of the edges on P. Importantly, by *augmenting* f along P, i.e. subtracting val(P) from all residual capacities along P, we get a new flow f' with value val(f') = val(f) + val(P). Additionally, f is maximum if and only if there exists no augmenting path on G_f, as shown by Ford and Fulkerson in [22].

The notion of augmenting paths was used in the first description of a generic maximum flow algorithm, the Ford-Fulkerson algorithm [29], introduced in 1956. The algorithm starts with a zero flow f, and then repeatedly searches for augmenting paths in G_f, augmenting f along them once found. When no augmenting path remains, f is maximum. This original algorithm does not specify the mechanism used to search for augmenting paths, or the order in which augmenting paths have to be discovered. Many flow algorithms are simply refinements of the original Ford-Fulkerson algorithm, specifying the mechanisms to find augmenting paths.

2.4.4 Level Graphs

Let $F = (G, c, s, t)$ be a flow network operating on the graph $G = (V, E)$, and let f be a flow on F. Additionally, for a vertex $v \in V$, let $d(v)$ denote the length of the shortest s-v path in G_f, or ∞ if no such path exists. The *level graph* L_f is the subgraph of G_f with the vertex set V, containing only those edges $(u, v) \in E_f$ for which $d(u) + 1 = d(v)$ holds.

In other words, for any vertex $v \in V$, the only s-v paths in L_f are those of shortest length. In particular, out of all augmenting paths on G_f, L_f contains only those of minimum length. This is very useful for some maximum flow algorithms that aim to process augmenting paths in order from shortest to longest; by operating on the level graph instead of the full residual graph, they can ensure that any augmenting path they find is of minimum length.

2.4.5 Blocking Flows

Let $F = (G, c, s, t)$ be a flow network operating on the graph $G = (V, E)$, and let f be a flow on F. Let H be a connected subgraph of G_f containing both s and t. A *blocking flow* is a maximum flow f' on the flow network (H, c_f, s, t). Given such a blocking flow f', one can augment the entirety of f by subtracting from every edge's residual capacity its flow value in f', thereby creating a new flow of value val(f) + val(f').

Some maximum flow algorithms make use of this by repeatedly computing the blocking flow on certain subgraphs of G_f; with the right choice of subgraphs, this can be more efficient than individually searching for augmenting paths in the entire residual graph. In particular, a blocking flow on the level graph L_f is able to saturate all shortest augmenting paths on G_f at once.

2.4.6 Cuts

Let $F = (G, c, s, t)$ be a flow network operating on the graph $G = (V, E)$. A *cut* is a partition of V into two sets, S and T, such that $s \in S$ and $t \in T$. Additionally, $G[S]$ and

$G[T]$ must be connected, in the sense that for any $v \in S \setminus \{s\}$, there must exist a s-v path in $G[S]$, and for any $v \in T \setminus \{t\}$, there must exist a v-t path in $G[T]$. The *cut edges* are the set E' of all edges going from S to T, i.e.

$$E' = \{(u, v) \in E : u \in S, v \in T\}.$$

We define the *value* of the cut to be the sum of the capacities of all cut edges, denoted by $\text{val}(S, T)$. A *minimum cut* is then defined as a cut of minimum value. Ford and Fulkerson proved the following correspondence between minimum cut and maximum flow:

Let f be a maximum flow on F, and let S be the set of vertices reachable from s in G_f. Then $(S, V \setminus S)$ is a minimum cut in F.

CHAPTER 3

Related Work

This chapter aims to give an overview of the algorithms relevant to the thesis. As the main work on the thesis is a comparison between different maximum flow algorithms for use in the Critical Weighted Independent Set reduction, this chapter mainly concerns itself with these flow algorithms. As such, we describe in detail the two newly implemented algorithms, Dinitz's Algorithm and Tidal Flow, in Sections 3.1 and 3.2, and give an overview of the previously existing baseline algorithm, Push-Relabel, in Section 3.3. Finally, we discuss the KAMIS architecture and CWIS implementations already used by some existing MWIS solvers in Section 3.4.

3.1 Dinitz's Algorithm

Dinitz's Algorithm, introduced by Yefim Dinitz in 1970, is an early refinement to the classic Ford-Fulkerson algorithm [29], providing strong polynomial time bounds and improving time complexity over the previously existing Edmonds-Karp algorithm [18]. While it was originally introduced by Dinitz in [14], we focus here on a version presented again by Dinitz in [15] with algorithmic changes made by Even [19] and implementation refinements suggested by Cherkassky [11].

The core idea behind Dinitz's Algorithm is to not search for individual augmenting paths in the residual graph, but to instead reduce the scope of the search for augmenting paths to a level graph as described in Section 2.4.4. The algorithm initializes a zero flow f, repeatedly computes level graphs on G_f and then searches for augmenting paths in the level graph, until none remain. The main structure of the algorithm is shown in Algorithm 1.

3.1.1 Computing the Level Graphs

To compute the level graph on a residual graph $G_f = (V, E_f)$, it is easiest to compute $d(v)$ for all $v \in V$, using a Single-Source-Shortest-Path algorithm. Since the edges all have unit

Algorithm 1: The Dinitz algorithm's outer structure.

```
Function Dinitz($G_f$, $s$, $t$):
    loop
        $L_f \leftarrow$ level graph on $G_f$
        if $t \notin L_f$ then
            return
        while $L_f$ contains an augmenting path do
            $P \leftarrow$ augmenting path in $L_f$
            Augment $L_f$ along $P$ by $\mathrm{val}(P)$
```

distances, the easiest and most efficient algorithm is a simple breadth first search from s. The exact structure of this BFS depends on the choice of data structure for L_f; Cherkassky recommends in [11] to not build any auxiliary network for L_f, and to instead only compute $d(v)$ and check whether an edge $(u, v) \in E$ lies in L_f by checking whether $d(v) = d(u)+1$ and $c(u, v) > f(u, v)$. As Dinitz notes in [15], the BFS may stop early once it encounters t, as we only care about augmenting paths in L_f and thus any $v \in V \setminus \{t\}$ with $d(v) \geq d(t)$ may be omitted from L_f entirely.

3.1.2 Augmenting the Level Graphs

Once a given level graph L_f has been computed, the next task is to augment the flow on it until it blocks L_f. Dinitz's Algorithm accomplishes this using a modified depth first search. Even though the goal of Dinitz's Algorithm is to go through augmenting paths in order of length, it does not need to use a BFS to search for the individual augmenting paths (unlike Edmonds-Karp [18]), because the level graph structure guarantees that any augmenting path in L_f will be a shortest augmenting path in G_f.

Dinitz's Algorithm further modifies the DFS to take advantage of the structure of the graph to identify edges it can safely skip, pruning large parts of the search space. Specifically, since for any residual edge (u, v), the level graph L_f contains at most one of (u, v) and (v, u) as an edge, any edge that gets saturated by an augmenting path in L_f may never get unsaturated by another later discovered augmenting path. Thus, whenever the DFS sees an edge that is saturated, the algorithm knows that this edge will never have to be checked again in a later DFS iteration on the same level graph. A recursive pseudocode implementation of this DFS is shown in Algorithm 2. It carries with it a progress array p, storing the next neighbor to be visited for each vertex, as well as an upper bound ε on the amount of flow that can be pushed from the source through the currently visited path. The return value of the function is simply the amount of flow that may be pushed to u. If the DFS reaches the sink, it returns exactly the value of the augmenting path that it found, augmenting each edge along that path as the recursion unravels.

Algorithm 2: Dinitz's modified DFS. u is the vertex currently visited by the DFS, ε is the amount of flow that can be sent via the current path of the DFS and p is the progress array used to prune the level graph.

```
Function Augment ($u$, $\varepsilon$, $p$):
    if $u = t$ then
        return $\varepsilon$
    while $p(u) \leq \deg_{\text{out}}(u)$ do
        $v \leftarrow p(u)$-th neighbor of $u$
        $r \leftarrow c(u,v) - f(u,v)$
        if $d(v) = d(u) + 1$ and $r > 0$ then
            $\varepsilon' \leftarrow$ Augment ($v$, $\min\{\varepsilon, r\}$, $p$)
            if $\varepsilon' > 0$ then
                $f(u,v) \leftarrow f(u,v) + \varepsilon'$
                $f(v,u) \leftarrow f(v,u) - \varepsilon'$
                return $\varepsilon'$
        $p(u) \leftarrow p(u) + 1$
    return 0
```

The inner loop of Dinitz's Algorithm then simply sets $p(v) = 1$ for all $v \in V$, and repeatedly calls `Augment` (s, ∞, p), until 0 is returned, at which point L_f is blocked by f, and Dinitz's Algorithm needs to compute a new level graph.

3.1.3 Correctness and Performance

The crucial property underlying Dinitz's Algorithm's correctness is that the distance from s to t increases in G_f with each iteration of the outer loop; that is, $d(t)$ is always strictly larger in a given level graph than in the one preceding it. This was proven by Dinitz for his original algorithm [14], and by Even for his improved version [19]. Additionally, on a given level graph, each iteration of `Augment` either saturates at least one edge, or signals that the level graph is blocked, so each outer iteration of Dinitz's Algorithm takes finite time. Since $d(t)$ strictly increases with each outer iteration, and $d(t)$ is trivially bounded to be at most $n - 1$, the algorithm then consists of a finite number of outer iterations, each taking finite time. At the end, since t is disconnected from s in G_f, a maximum flow has been found, so the algorithm is correct.

Regarding runtime bounds, the pruning behavior of the DFS enables each augmenting path search to complete in amortized $O(n)$ time; since each augmenting path saturates at least one edge, this leads to $O(nm)$ runtime to find a blocking flow on a given level graph. The level graph computations each take $O(m)$ time using BFS, and there are $O(n)$ outer iterations, so the total runtime is bounded by $O(n \cdot (m + nm)) = O(n^2 m)$.

3.2 The Tidal Flow Algorithm

The Tidal Flow algorithm [21], introduced by Matthew Fontaine in 2018, is designed to be an easily teachable, easily implementable, and competitively performant maximum flow algorithm compared to other popular algorithms.

Similarly to Dinitz's Algorithm, it uses level graphs, and search for augmenting paths within each level graph to find a blocking flow in the level graph. The mechanism used to find the blocking flows is quite different from the one used in Dinitz' algorithm however; whereas Dinitz uses a modified depth first search to find a blocking flow in $O(nm)$ time, Tidal Flow instead uses a novel procedure which the author calls a *tide cycle*. Using a preflow based approach, each tide cycle augments not just one path, but every edge in the level graph, until the flow blocks the level graph.

3.2.1 The Tide Cycle

One iteration of the tide cycle works as follows: It receives as input the vertices V, the edge capacities c and current flow f which it modifies in place, as well as E_L, the edge list of the current level graph, sorted by ascending distance from s. The tide cycle is divided into three phases, which the author calls *High Tide*, *Low Tide*, and *Erosion*. During the high tide phase, an upper bound is calculated on how much flow can reach each vertex through E_L. During low tide, this upper bound is then reduced to a feasible preflow, using a backwards pass through E_L. Finally, during the erosion phase, each edge's residual capacity is updated according to the values calculated in the low tide phase with a final forward pass through E_L. Pseudocode for the tide cycle function is given in Algorithm 3. In the pseudocode, h refers to the high tide values, l to the low tide values, and p keeps track of how much flow each edge in E_L is "promised" to receive.

For a given level graph, the `TideCycle` function must be called until it returns 0; only then can the Tidal Flow algorithm compute the next level graph.

3.2.2 Correctness and Performance

To prove the correctness of the algorithm, Fontaine simply argues that every iteration of tide cycle on a non-blocking flow saturates at least one edge in the network. Since for a given level graph, saturated edges never become unsaturated, it then takes at most m tide cycles to produce a blocking flow on that level graph. As Dinitz showed for his algorithm, the next computed level graph will then only include augmenting paths of higher length, bounding the number of required level graph and blocking flow computations by n. Therefore, the algorithm always terminates, producing a valid maximum flow for any input.

For the time complexity, consider that a single tide cycle iteration trivially takes $O(m)$ time, and a single level graph computation takes $O(m)$ time using breadth first search. Since there are at most n level graph computations, and at most m tide cycle iterations per level graph, the total runtime complexity then becomes $O(n \cdot (m + m \cdot m)) = O(nm^2)$. While this is asymptotically slower than the $O(n^2m)$ of Dinitz's Algorithm, Fontaine conjectures that often way fewer than m tide cycles are required to find a blocking flow, hoping that the algorithm can outperform its asymptotic worst case bounds in many cases in practice. He does however mention that it is difficult to find tighter bounds for the average number of required tide cycles, as certain pathological patterns in the flow network can "trick" the algorithm into augmenting very little flow per tide cycle.

Fontaine reports performance competitive with Dinitz's Algorithm and far better than other algorithms examined in the paper for bipartite sparse flow networks with high edge capacities. As this is exactly the shape of network produced in the Critical Weighted Independent Set reduction (as described in Section 4.4.2), this made Tidal Flow an interesting candidate to include in the comparison.

3.3 The Push-Relabel Algorithm

The Push-Relabel algorithm [25], introduced by Goldberg and Tarjan in 1986, works quite differently than the two previously described algorithms. Building on ideas from Karzanov's algorithm [31], Push-Relabel departs from the idea of searching for augmenting paths in favor of a preflow based approach. Starting with an initial feasible preflow, the algorithm iteratively pushes excess flow around, either towards the sink or towards the source. It repeats this process until no vertices have excess flow, except for the source and sink vertices. At that point, the preflow becomes a feasible flow of maximum value.

To decide where to push excess flow on a residual graph $G_f = (V, E_f)$, Push-Relabel uses a *labeling function* $d : V \to \mathbb{N}_0 \cup \{\infty\}$, defined as a function with $d(s) = n, d(t) = 0$, and $d(u) \leq d(v)+1$ for all residual edges $(u, v) \in E_f$. For a vertex v, if $d(v) < n$ then $d(v)$ is a lower bound on the distance from v to t in G_f, and if $d(v) \geq n$ then $d(v) - n$ is a lower bound on the distance from v to s in G_f. Additionally, in the latter case, t is unreachable from v in G_f. The authors additionally introduce the notion of an *active vertex*, defined as a vertex $v \in V \setminus \{s, t\}$ with $d(v) < \infty$ and $\chi_f(v) > 0$.

Given an initial preflow f and a labeling d that is valid for G_f, the algorithm repeatedly performs one of two operations, `Push` and `Relabel`. The `Push` operation is applicable to a vertex u if u is active and has a residual edge to a vertex v such that $d(u) = d(v) + 1$; in that case, the operation pushes as much excess flow as it can from u to v. The amount of excess pushed is bounded by $\min\{\chi_f(u), c_f(u, v)\}$. Notice that the `Push` operation either deactivates u or saturates the edge (u, v), or both.

The other operation, `Relabel`, is applicable to any active vertex u, such that $d(u) \leq d(v)$ for residual edges $(u, v) \in E_f$. It changes the labeling itself, setting the label $d(u)$ to the value $\min\{d(v) + 1 : (u, v) \in E_f\}$. In the case that there are no outgoing residual edges from u, it sets $d(u)$ to ∞.

A generic algorithm may just repeatedly search for active vertices for which either operation is applicable, and apply that operation to them; Goldberg and Tarjan show that any

Algorithm 3: The tide cycle algorithm. E_L is the edge list of the level graph, ordered by distance from s. The algorithm modifies f in place and returns the amount of augmented flow. 0 is returned *iff* f blocks the level graph.

```
Function TideCycle($V$, $E_L$, $c$, $s$, $t$, $f$):
    $p(u,v) \leftarrow 0 \quad \forall (u,v) \in E_L$
    $h(v) \leftarrow 0 \quad \forall v \in V$
    $h(s) \leftarrow \infty$
    foreach $(u,v) \in E_L$ do
        $p(u,v) \leftarrow \min\{c(u,v) - f(u,v), h(u)\}$
        $h(v) \leftarrow h(u) + p(u,v)$
    if $h(t) = 0$ then
        return 0
    $l(v) \leftarrow 0 \quad \forall v \in V$
    $l(t) \leftarrow h(t)$
    foreach $(u,v) \in E_L$ in reverse order do
        $p(u,v) \leftarrow \min\{p(u,v), h(u) - l(u), l(v)\}$
        $l(v) \leftarrow l(v) - p(u,v)$
        $l(u) \leftarrow l(u) + p(u,v)$
    $h(v) \leftarrow 0 \quad \forall v \in V$
    $h(s) \leftarrow l(s)$
    foreach $(u,v) \in E_L$ do
        $p(u,v) \leftarrow \min\{p(u,v), h(u)\}$
        $h(u) \leftarrow h(u) - p(u,v)$
        $h(v) \leftarrow h(v) + p(u,v)$
        $f(u,v) \leftarrow f(u,v) + p(u,v)$
        $f(v,u) \leftarrow f(v,u) - p(u,v)$
    return $h(t)$
```

execution of this algorithm will lead to a correct maximum flow. To prove this, they show that for any active vertex, exactly one of the operations will be applicable. Furthermore, they show that the algorithm retains the invariant that d is a valid labeling, and that if the algorithm ever terminates with all distance labels being finite, the preflow will at that point be a maximum flow. They then reason that for a given vertex, its distance label only ever increases, and is bounded above by $2n - 1$. Finally, they bound the total number of `Push` and `Relabel` operations to be finite, proving that the algorithm always terminates, and that no distance labels will be infinite upon termination, implying that the result will always be a maximum flow.

Goldberg and Tarjan also present an efficient ordering of operations for this algorithm, using a First-In-First-Out queue of active vertices to be processed. This efficient algorithm goes through each vertex in the queue, and repeatedly applies either `Push` or `Relabel` to the dequeued vertex, until either its excess becomes 0 or its distance label gets increased. Any vertices that become active during this process are added to the end of the queue, and any vertex whose distance label gets increased also gets readded to the queue to be reprocessed later.

Goldberg and Tarjan show that by using this particular ordering, the number of *nonsaturating* pushes (`Push` operations along an edge (u, v) that leave (u, v) unsaturated) gets bounded to $O(n^2)$. With this, they prove that the entire algorithm runs in $O(n^3)$ time, asymptotically faster than both Tidal Flow and Dinitz's Algorithm.

3.4 KAMIS and Other Solvers

KAMIS, presented by Lamm et al. in [32], is a state-of-the-art MAXIMUM WEIGHT INDEPENDENT SET solver, introducing several new reduction rules, and using a novel reduction algorithm based on the *branch-and-reduce* [4] paradigm. Branch-and-reduce builds on an existing approach for exact solvers for combinatorial problems, called *branch-and-bound* [33]. A solver using the branch-and-bound approach systematically searches the entire solution space of a problem, while keeping track of lower or upper bounds for the optimal solution. If any partial solution falls outside those bounds, the entire subtree of the search tree rooted at that solution may be discarded. As an example, a solver for the MWIS problem may keep track of the highest weight of any independent set found so far. If a partial solution excludes enough vertices that the weight of all non-excluded vertices is less than the best weight found so far, the solution, and all of its descendants in the search tree, may immediately be discarded.

Branch-and-reduce builds on this approach by introducing a reduction step before every branch, ensuring that only inputs that are irreducible by the chosen reduction rules are branched from. When branching, a small portion of the graph is either included in or excluded from the partial solution, effectively removing that portion from the graph. Many local data reductions are then checked only in the vicinity of the changes, instead of rechecking the entire graph. Since data reductions are quite fast (often linear in the graph size), they can often reduce the search space a lot more efficiently than a pure exhaustive search, which is exponential in the graph size.

KAMIS implements the reduction step of the branch-and-reduce algorithm using a so-called *reduction pipeline*, a list of different data reductions to try on a given problem instance. The reduction algorithm goes through the list of reductions, trying to reduce the given instance with each one. If one reduction succeeds, the algorithm restarts from the beginning of the pipeline. Only when all data reductions fail does the algorithm return the reduced instance.

A number of MWIS solvers, including KAMIS, already implement the Critical Weighted Independent Set reduction. While they all use the same basic algorithm, which is described in Section 4.5, that algorithm does not dictate the choice of flow algorithm used in each solver. We are aware of six different solvers implementing the CWIS reduction:

KaMIS by Lamm et al. [32] from 2019, Struction by Gellner et al. [23] from 2021, GNN-VC by Langedal et al. [34] from 2022, m^{2}wis by Großmann et al. [26] from 2023 and LearnAndReduce by Großmann et al. [27] from 2024 all use Push-Relabel implementations for their CWIS reduction code. The singular outlier is Solve by Xiao et al. [41], which opts to use the ISAP maximum flow algorithm by Ahuja et al. [3]. It is not surprising that the majority of the solvers use Push-Relabel; when the shape of the input graphs is unknown, Push-Relabel is generally the algorithm of choice. However, the evaluation done by Fontaine [21] suggests that sparse bipartite flow networks with high capacities, as used in the CWIS reduction (see Section 4.5), may be better suited to the Tidal Flow algorithm, or Dinitz's Algorithm.

CHAPTER 4

The Critical Weighted Independent Set Reduction

Let $G = (V, E)$ be a graph with vertex weight function $w : V \to \mathbb{R}_{>0}$. A *critical weighted set* in G is a vertex set $U_c \subseteq V$, such that

$$w(U_c) - w(N(U_c)) = \max\{w(U) - w(N(U)) : U \subseteq V\}.$$

We define critical weighted independent sets analogously, so an independent set I_c is a *critical weighted independent set* in G if the following holds:

$$w(I_c) - w(N(I_c)) = \max\{w(I) - w(N(I_c)) : I \text{ is independent in } G\}$$

Note that $w(U_c) \geq w(N(U_c))$ and $w(I_c) \geq w(N(I_c))$ always hold, as $w(\emptyset) - w(N(\emptyset)) = 0$.

The *Critical Weighted Independent Set* (CWIS) reduction, introduced by Butenko and Trukhanov in [9], is a data reduction for the MWIS problem, using critical weighted independent sets. Given a non-empty critical weighted independent set I_c, the reduction is defined by its parts REDUCEGRAPH and RESTORESOLUTION:

$$\begin{aligned} \textsc{ReduceGraph}(G) &:= G \setminus (I_c \cup N(I_c)) \\ \textsc{RestoreSolution}(I) &:= I \cup I_c \end{aligned}$$

Put into words, the reduction rule states that I_c is guaranteed to be a subset of some maximum weight independent set. Removing I_c and its neighborhood from G therefore reduces the problem to finding a maximum weight independent set on the rest of G.

On the surface, this looks like a very easy and powerful reduction, that dominates many other simpler reductions. However, finding a critical weighted independent set in G is nontrivial. Ageev [2] introduced a polynomial-time algorithm to find a critical weighted independent set, by reducing the problem to a maximum flow computation on a specially constructed flow network. The remainder of this chapter will be describing this algorithm and the steps used to derive it, and proving their correctness.

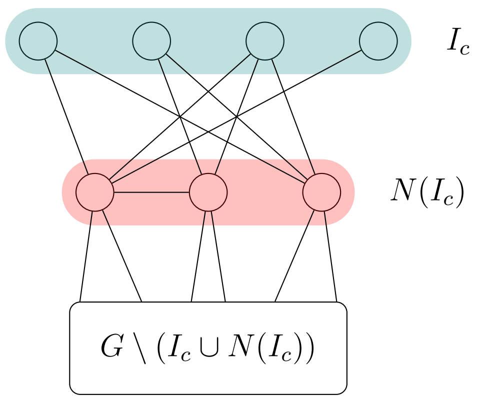

Figure 1: Visualization of the CWIS reduction. [12] describes this as a crown structure, where the spikes on the crown form a critical weighted independent set. Since I_c has at least the same weight as $N(I_c)$, all of I_c may be included in, and all of $N(I_c)$ excluded from the solution.

4.1 Proving the Reduction's Correctness

First, we must prove that the reduction as stated actually works, i.e. reducing a graph, solving on the reduced graph, and restoring that solution always produces a maximum weight independent set on the original graph. Butenko and Trukhanov [9] proved this using the following theorem:

Theorem 1.
Let $G = (V, E)$ be a graph with vertex weight function $w : V \to \mathbb{R}_{>0}$ and let I_c be a critical weighted independent set in G. Then there exists a maximum weight independent set I, such that $I_c \subseteq I$.

Proof. Let J be a maximum weight independent set in G, and consider $I := (J \cup I_c) \setminus N(I_c)$. Since I_c is independent, $I_c \cap N(I_c) = \emptyset$, so $I_c \subseteq I$. And since J is independent and any vertices in J that are adjacent to some vertex in I_c get removed, it follows that I is also independent. It remains to show that I is of maximum weight, i.e. that $w(I) \geq w(J)$.

Assume for the sake of contradiction that $w(I) < w(J)$. It follows that

$$w(J \setminus I) = w(J) - w(J \cap I) > w(I) - w(I \cap J) = w(I \setminus J). \tag{4.1}$$

By construction of I, we have $I \setminus J = I_c \setminus J$ and $J \setminus I = J \cap N(I_c)$, so (4.1) becomes

$$w(J \cap N(I_c)) > w(I_c \setminus J). \tag{4.2}$$

Since I_c is independent, we know that $I_c \cap N(I_c) = \emptyset$, and thus $N(I_c \cap J) \cap (N(I_c) \cap J) = \emptyset$. And since both $N(I_c \cap J)$ and $N(I_c) \cap J$ are subsets of $N(I_c)$, the following inequality is guaranteed to hold:

$$w(N(I_c)) \geq w(N(I_c) \cap J) + w(N(I_c \cap J)) \tag{4.3}$$

Combining (4.2) and (4.3) we get

$$\begin{aligned} w(I_c) - w(N(I_c)) &= w(I_c \setminus J) + w(I_c \cap J) - w(N(I_c)) \\ &< w(N(I_c) \cap J) + w(I_c \cap J) - w(N(I_c)) \\ &\leq w(N(I_c) \cap J) + w(I_c \cap J) - w(N(I_c) \cap J) - w(N(I_c \cap J)) \\ &= w(I_c \cap J) - w(N(I_c \cap J)) \end{aligned}$$

which contradicts with I_c being of critical weight. Thus, $w(I) \geq w(J)$, so I is a maximum weight independent set containing I_c as a subset. □

Using this theorem, it is easy to see the correctness of the reduction: Any critical weighted independent set I_c will be contained in some maximum weight independent set, so we can include it in the solution and search for the rest of that MWIS in the remaining graph after removing I_c and its neighborhood.

4.2 Obtaining a CWIS From a Critical Weighted Set

Suppose we had an algorithm to compute a critical set U_c on our input graph. Ageev [2] showed that it is then also possible to efficiently construct a critical weighted independent set I_c from U_c:

Theorem 2.
Let $G = (V, E)$ be a graph with vertex weight function $w : V \to \mathbb{R}_{>0}$ and let U_c be a critical weighted set in G. Then

$$I_c := U_c \setminus N(U_c)$$

is a critical weighted independent set in G.

Proof. Clearly, I_c is independent in G. Let $U'_c := U_c \setminus I_c$ and $X := N^*(U'_c) \cup N(I_c)$, then $N(U_c) = U'_c \cup X$ and $U'_c \cap X = \emptyset$.

First, to show the latter, let $v \in U'_c$, i.e. $v \in U_c$ and $v \notin I_c$. Since $v \in U_c$ it follows that $N(v) \cap I_c = \emptyset$ and therefore also $v \notin N(I_c)$. And clearly $v \notin N^*(U'_c)$, so $v \notin X$, and thus $U'_c \cap X = \emptyset$.

Next, we show that $N(U_c) = U'_c \cup X$. Substituting the definitions for U'_c and X and using the identity $U_c \setminus I_c = U_c \cap N(U_c)$ gives us

$$\begin{aligned} U'_c \cup X &= (U_c \setminus I_c) \cup (N^*(U_c \setminus I_c) \cup N(I_c)) \\ &= (U_c \setminus I_c) \cup N(U_c \setminus I_c) \cup N(I_c) \\ &= (U_c \cap N(U_c)) \cup N(U_c) \\ &= N(U_c). \end{aligned}$$

Algorithm 4: Isolating a CWIS from a critical weighted set

```
Function IsolateCWIS($U_c$):
    $H \leftarrow \emptyset$
    $I_c \leftarrow \emptyset$
    foreach $u \in U_c$ do
        $H \leftarrow H \cup \{u\}$
    foreach $u \in U_c$ do
        foreach $\{u, v\} \in E$ do
            if $v \in H$ then
                continue to next $u$.
        $I_c \leftarrow I_c \cup \{u\}$
    return $I_c$
```

Finally, we use these results to arrive at the following inequality:

$$\begin{aligned} w(U_c) - w(N(U_c)) &= (w(U_c') + w(I_c)) - (w(U_c') + w(X)) \\ &= w(I_c) - w(X) \\ &= w(I_c) - w(N^*(U_c') \cup N(I_c)) \\ &\le w(I_c) - w(N(I_c)) \end{aligned}$$

Since U_c is a critical weighted set, I_c must then be a critical weighted independent set. □

Given a critical weighted set U_c in list representation, one can efficiently compute $I_c = U_c \setminus N(U_c)$ in linear time, e.g. using a hash set, as shown in Algorithm 4. It remains to find an algorithm to identify a critical weighted set in G.

4.3 Finding Critical Weighted Sets Using ILP

Let $G = (V, E)$ be a graph with vertex weight function $w : V \to \mathbb{R}_{>0}$. Consider the following $\{0, 1\}$-integer linear program:

$$\max \quad \sum_{u \in V} w(u)x_u - \sum_{v \in V} w(v)y_v \tag{4.4}$$

$$\text{s.t.} \quad y_v \ge x_u \qquad \forall \{u, v\} \in E \tag{4.5}$$

$$x_u, y_u \in \{0, 1\} \qquad \forall u \in V$$

Ageev [2] showed the following:

Theorem 3.
Let $(x^*_v), (y^*_v)$ be $\{0,1\}$-vectors forming an optimal solution to the above ILP. Then

$$X^* := \{u \in V : x^*_u = 1\}$$

is a critical weighted set in G.

Proof. Let $X^* := \{u \in V : x^*_u = 1\}$ and $Y^* := \{v \in V : y^*_v = 1\}$. Note that $N(X^*) \subseteq Y^*$, otherwise there would exist some $\{u, v\} \in E$ such that $u \in X^*$ and $v \notin Y^*$. By construction of X^* and Y^* this would mean that $y^*_v = 0 < 1 = x^*_u$, violating constraint (4.5).

Now suppose there existed some $v \in Y^* \setminus N(X^*)$. Setting y^*_v to 0 would not violate any constraints, but would increase the objective value (4.4) by $w(v)$, which by definition of w is strictly positive. This would contradict the optimality of the solution $(x^*_v), (y^*_v)$, thus it must be that $Y^* \setminus N(X^*) = \emptyset$ and therefore $Y^* = N(X^*)$.

Consider now some arbitrary $X \subseteq V$ and let $(x_v), (y_v)$ be $\{0,1\}$-vectors such that for all $v \in V$, $x_v = 1 \Leftrightarrow v \in X$ and $y_v = 1 \Leftrightarrow v \in N(X)$. Clearly, $(x_v), (y_v)$ is a feasible solution to the ILP, so we get

$$\begin{aligned} w(X) - w(N(X)) &= \sum_{u \in V} w(u) x_u - \sum_{v \in V} w(v) y_v \\ &\leq \sum_{u \in V} w(u) x^*_u - \sum_{v \in V} w(v) y^*_v \\ &= w(X^*) - w(Y^*) = w(X^*) - w(N(X^*)) \end{aligned}$$

thus X^* must be a critical weighted set in G. □

To find a critical set in G, it therefore suffices to solve the ILP. While this may not seem like progress, as ILPs in general are NP-hard, this specific one is an instance of the so called *Selection Problem*, which is efficiently solvable, as described next.

4.4 The Selection Problem

The *Selection Problem* was introduced in 1970 by Rhys [37] and further analyzed by Balinski [5]. The problem statement goes as follows:

Suppose you are given a finite set F of *facilities*, and a finite set A of *activities*, with each facility $f \in F$ having some associated *cost* $c_f > 0$, and each activity $a \in A$ having some associated *profit* $p_a > 0$. Additionally, you are given a *dependency* function $d : A \to \mathcal{P}(F)$, mapping to each activity the facilities required to perform that activity. A *selection* is a set of activities $\Sigma \subseteq A$ with the associated *value*

$$\text{val}(\Sigma) := \sum_{a \in \Sigma} p_a - \sum_{f \in \bigcup d(\Sigma)} c_f.$$

Put into words, the value is simply the sum of the profits gained from all activities in Σ minus the costs associated with constructing the required facilities $\bigcup d(\Sigma)$. The Selection Problem is then to find a selection of maximum value.

4.4.1 Expressing the Selection Problem as an ILP

Rhys [37] showed that the problem can be solved using the following ILP formulation:

$$\begin{aligned} \max \quad & \sum_{a \in A} p_a x_a - \sum_{f \in F} c_f y_f \\ \text{s.t.} \quad & y_f \geq x_a && \forall a \in A, f \in d(a) \\ & x_a \in \{0,1\} && \forall a \in A \\ & y_f \in \{0,1\} && \forall f \in F \end{aligned}$$

Given an optimal solution $(x^*_a), (y^*_f)$ to the ILP, the set $\Sigma^* := \{a \in A : x^*_a = 1\}$ is a selection of maximum value. The proof for this is very similar to the proof given in Section 4.3, and will therefore be omitted here.

Consider a graph $G = (V, E)$ with vertex weight function $w : V \to \mathbb{R}_{>0}$ and notice that the ILP described in Section 4.3 becomes exactly the same as the ILP above when choosing $F = A = V$ as well as $c_v = p_v = w(v)$ and $d(v) = N(v)$ for all $v \in V$. Thus, solving the Selection Problem is sufficient to find a critical weighted set in G.

4.4.2 Solving the Selection Problem using Flow Networks

Balinski [5] showed a way to solve the Selection Problem efficiently using flow networks. For an instance (F, A, c, p, d) of the selection problem with $F \cap A = \emptyset$, consider the flow network $H = (G, c, s, t)$ where $G = (V, E)$ with

$$V = \{s, t\} \cup F \cup A$$

and

$$E = \{(s, a) : a \in A\} \cup \{(f, t) : f \in F\} \cup \{(a, f) : a \in A, f \in d(a)\}.$$

For an edge $(u, v) \in E$, its capacity shall be as follows:

$$c(u, v) = \begin{cases} p_v & \text{if } u = s, \\ c_u & \text{if } v = t, \\ \infty & \text{otherwise} \end{cases}$$

While our definition of flow networks technically does not allow infinite edge capacity, the ∞ is only used for brevity here, and may be substituted with a large finite value if needed. For any edge mentioned above, H should also include its opposite edge, with capacity 0. Figure 2 shows roughly what this flow network may look like for a given instance of the Selection Problem.

Using this network, Balinski [5] proved the following:

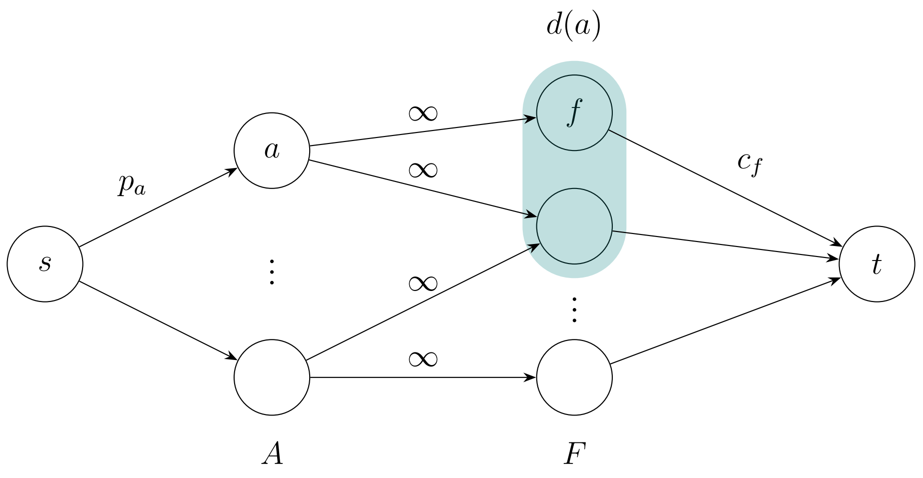

Figure 2: Example flow network for an instance of the Selection Problem. Each activity a has a corresponding vertex in the left partition, with capacity p_a coming from the source. Each facility f has a corresponding vertex in the right partition, with capacity c_f going to the sink. Lastly, each activity has edges of infinite capacity going to all of its dependencies.

Theorem 4.
Let (S, T) be a minimum cut on H. Then, $A \cap S$ is a selection of maximum value.

The proof presented here is adapted from Balinski's [5]. As an aid in the proof, we introduce the notion of *maximal selections*. We call a selection Σ maximal, if $d(a) \not\subseteq \bigcup d(\Sigma)$ for all $a \in A \setminus \Sigma$, i.e. if no unselected activity has all its requirements met by the selected activities already. We first prove the following lemmata:

Lemma 1.
Let $\Sigma \subseteq A$ be a selection of maximum value. Then Σ is maximal.

Proof. Suppose for contradiction that Σ is not maximal. Then there must exist an activity $a' \in A \setminus \Sigma$ such that $d(a') \subseteq \bigcup d(\Sigma)$. Let $\Sigma' := \Sigma \cup \{a'\}$, then the following holds for their values:

$$\begin{aligned}
\text{val}(\Sigma') &= \sum_{a \in \Sigma'} p_a - \sum_{f \in \bigcup d(\Sigma')} c_f \\
&= p_{a'} + \sum_{a \in \Sigma} p_a - \sum_{f \in \bigcup d(\Sigma)} c_f \\
&= p_{a'} + \text{val}(\Sigma) \\
&> \text{val}(\Sigma)
\end{aligned}$$

Because Σ was chosen to be of maximum value, this is a contradiction, so Σ must be maximal. □

Lemma 2.
There exists a one-to-one correspondence between cuts of finite value in H and maximal selections in A.

Proof. First, let (S, T) be any cut of finite value, and consider the selection $\Sigma := A \cap S$. Because the cut value is finite, there may be no cut edge (a, f) with $a \in S, f \in T$, so for any a in Σ, it must be that $d(a) \subseteq S$. On the other hand, any $f \in F \cap S$ must have an s-f path in $H[S]$, of which the second last vertex must be in Σ. Thus, any facility in S must be required by at least one activity in S. And finally, since any activity $a \in A \cap T$ must have an a-t path in $H[T]$, the second vertex of that path must be a facility in T, so any activity not in Σ has at least one required facility not in S.

All in all, we get that Σ must be a maximal selection, and that $S = \{s\} \cup \Sigma \cup \bigcup d(\Sigma)$. And since we were able to deduce the entirety of S solely from $A \cap S$, it must also be that the function $(S, T) \mapsto A \cap S = \Sigma$ mapping finite cuts to maximal selections is injective.

For the other direction, consider some maximal selection Σ. Let $S := \{s\} \cup \Sigma \cup \bigcup d(\Sigma)$ and $T := V \setminus S$. Clearly, (S, T) is a cut in H. And since for any $a \in \Sigma$, all its dependencies $d(A)$ also lie in S, the cut has finite value. Because $A \cap F = \emptyset$, the function $\Sigma \mapsto (S, T)$ is also trivially injective, and is clearly the inverse of the function mapping finite cuts to maximal selections. □

Lemma 3.
A minimum cut in H corresponds to a maximal selection of maximum value in A, according to the correspondence from Lemma 2.

Proof. Let (S, T) be a minimum cut. The cut value of (S, T) must be finite, as e.g. the cut $(\{s\}, V \setminus \{s\})$ trivially has finite value. Let thus $\Sigma := A \cap S$ be the maximal selection corresponding to (S, T). Additionally, let Σ' be any other maximal selection, with its corresponding cut (S', T'). Then the values of the cuts are related by

$$\begin{aligned} \mathrm{val}(S, T) &= \sum_{a \in A \setminus \Sigma} p_a + \sum_{f \in \bigcup d(\Sigma)} c_f \\ &\leq \sum_{a \in A \setminus \Sigma'} p_a + \sum_{f \in \bigcup d(\Sigma')} c_f = \mathrm{val}(S', T'). \end{aligned}$$

Let $p := \sum_{a \in A} p_a$. Then, $\sum_{a \in \Sigma} p_a = p - \sum_{a \in A \setminus \Sigma} p_a$ and $\sum_{a \in \Sigma'} p_a = p - \sum_{a \in A \setminus \Sigma'} p_a$. With this, the above becomes

$$\begin{aligned} p - \mathrm{val}(\Sigma) &= p - \sum_{a \in \Sigma} p_a + \sum_{f \in \bigcup d(\Sigma)} c_f \\ &\leq p - \sum_{a \in \Sigma'} p_a + \sum_{f \in \bigcup d(\Sigma')} c_f = p - \mathrm{val}(\Sigma') \end{aligned}$$

and therefore $\mathrm{val}(\Sigma') \leq \mathrm{val}(\Sigma)$. □

Algorithm 5: Finding critical weighted independent sets

```
Function FindCWIS(V, E, w):
    $V' \leftarrow$ copy of $V$
    $V_H \leftarrow \{s,t\} \cup V \cup V'$
    $E_H \leftarrow \{(s,v) : v \in V\} \cup \{(v',t) : v \in V\} \cup \{(u,v') : \{u,v\} \in E\}$
    $c(s,v) \leftarrow w(v) \quad \forall v \in V$
    $c(v',t) \leftarrow w(v) \quad \forall v \in V$
    $c(u,v') \leftarrow \infty \qquad \forall \{u,v\} \in E$
    $H \leftarrow ((V_H, E_H), c, s, t)$
    $f \leftarrow$ MaxFlow($H$)
    $S \leftarrow \{v \in V : \exists s\text{-}v$ path in $G_f\}$
    $U_c \leftarrow S \cap V$
    $I_c \leftarrow$ IsolateCWIS($U_c$)
    return $I_c$
```

With these lemmata, the theorem becomes quite simple to prove:

Proof for Theorem 4. Let (S,T) be a minimum cut on H. According to Lemma 3, (S,T) corresponds to a maximal selection Σ of maximum value. By Lemma 2, this selection is exactly $\Sigma := A \cap S$. And since any selection of maximum value must also be maximal (as per Lemma 1), Σ is in fact a selection of maximum value. □

4.5 The Final Algorithm

Putting all of the steps together, we arrive at an algorithm to efficiently identify a critical weighted independent set in a graph $G = (V,E)$ with vertex weight function $w : V \to \mathbb{R}_{>0}$. The algorithm constructs the special flow network described in Subsection 4.4.2, using disjoint copies V and V' of the vertex set as the activity and facility sets, and calculates a maximum flow on it. It then identifies a minimum cut, by scanning the residual graph from the source. From this minimum cut, it identifies the maximum selection, which is also a critical weighted set, as shown in Subsection 4.4.1. Finally, from the critical weighted set it constructs a critical weighted independent set, as described in Section 4.2. The pseudocode for the whole process is shown in Algorithm 5.

The algorithm is agnostic toward the exact flow algorithm used, and the graph scan can be implemented using either breadth first or depth first search. Since everything other than the maximum flow calculation runs in linear time ($O(n+m)$), the runtime of the whole algorithm will be dominated by that of the maximum flow algorithm used.

CHAPTER 5

Engineering Fast Flow Algorithms for the Critical Weighted Independent Set Reduction

In this chapter, we describe our implementations of the Dinitz and Tidal Flow algorithms. These algorithms have already been broadly described in Sections 3.1 and 3.2 respectively. Thus, the descriptions given here focus more on implementation details than on the algorithms themselves. We describe algorithm specific implementation choices in Sections 5.1 and 5.2, as well as algorithm agnostic data structure considerations in Section 5.3. Additionally, we describe optimizations to KAMIS's CWIS reductions that are independent of the flow algorithms in Sections 5.4 and 5.5.

5.1 The Dinitz Implementation

Our implementation of Dinitz's Algorithm closely follows the structure described in Section 3.1. Following Cherkassky's advice [11], we do not construct auxiliary data structures to represent the level graph, and instead use a simple distance array, excluding any edges (u, v) for which $d(v) \neq d(u) + 1$. Because of the pruning done by Dinitz's modified DFS, any edge of the flow graph that does not lie in the level graph will only be looked at once (per level graph) and immediately discarded. With this, computing the level graph becomes a very simple BFS, using a FIFO queue to compute $d(v)$ for all vertices. To make this queue as efficient as possible, we leverage the fact that the BFS looks at each vertex at most once, and thus at most n items are ever enqueued. Therefore, the queue can simply be an array of length n with indices pointing to the first and last element; $\mathrm{enqueue}$ and $\mathrm{dequeue}$ operations simply write to or read from the array at the corresponding index and increment it.

To find augmenting paths, our implementation uses a recursive DFS, much like the one shown in Algorithm 2. In principle, it is possible to replace the recursive DFS with an iterative variant that manually manages its stack, but the recursive version outperformed the iterative version in our experiments. Pseudocode faithful to our C++ implementation of Dinitz's Algorithm is given in Algorithm 7.

5.2 The Tidal Flow Implementation

The broad structure of our Tidal Flow implementation is very similar to that of Dinitz's Algorithm, with Dinitz's DFS being replaced by the tide cycle algorithm as shown in Algorithm 3. However, the level graph is represented quite differently for Tidal Flow than it is for Dinitz's Algorithm. Instead of an implied level graph using a distance array, we store a list of all the edges in the level graph, ordered by distance from the source. While it is possible in principle to use the distance array approach for Tidal Flow too, it appeared less efficient in early experiments. Since each iteration of the tide cycle algorithm needs to traverse the edges of the level graph in BFS order three times, the tide cycle algorithm itself would need to do three BFS searches when using the distance array representation. And unlike Dinitz's Algorithm, edges that lie outside the level graph cannot simply be pruned away after first encountering them. These overheads outweigh the cost of building a full edge list of each level graph.

To construct this edge list, the BFS implementation itself needs to be altered. For each visited vertex u, all (u, v) edges with $d(v) = d(u) + 1$ need to be appended to the edge list. This also means that the BFS cannot just exit once it encounters t, as vertices left in the queue may still have edges going to t. Instead, we switch to only enqueueing newly encountered vertices if the BFS has not yet visited the sink. Pseudocode faithful to our C++ implementation of Tidal Flow is given in Algorithm 8.

5.3 Flow Graph Data Structures

There are many different ways for flow graphs to be represented in memory. While the algorithms themselves are agnostic to the exact representation of the flow graphs, choosing the right underlying data structure can still lead to significant performance wins. Important for the performance of our flow algorithms are the ability to efficiently iterate all outgoing edges from a vertex, to query and modify the flow and capacity of an edge, and to query the opposing edge of a given edge in the flow graph. Flow graphs are defined to always have these opposing edges, and augmenting the flow along an edge requires augmenting the flow along the opposing edge too.

The existing maximum flow algorithm in KaMIS, a Push-Relabel implementation inherited from KaHIP [30], uses an *Adjacency List* to represent its flow graph. In this representation, each vertex holds a list, containing information about all of its outgoing edges. Part of this information for an edge (u, v) is the index of the edge (v, u) in v's edge list, which is required to query the opposing edge efficiently. The flow graph is then effectively a list of length n, containing many smaller lists, whose lengths add up to m.

Iterating the outgoing edges of a vertex consists of simply iterating the edge list corresponding to that vertex. To identify a single edge, both its source vertex index, as well as its index within that vertex's edge list are required, and accessing an edge's information usually requires two memory operations, as you must first look up the corresponding edge list, before accessing the edge itself.

An alternative representation of the flow graph is the so-called *Compressed Sparse Row* (CSR) representation. Instead of storing edges in separate lists for separate vertices, all edges are stored in one big list of length m. As part of the information for an edge (u, v), the index of the edge (v, u) is also stored. For each vertex, a start and end index in the edge list are stored, to identify which edges belong to the vertex. Representing each vertex's outgoing edge range as a half-open interval $[\mathit{start}, \mathit{end}) = \{\mathit{start}, \ldots, \mathit{end} - 1\}$, a given vertex's end index coincides with the next vertex's start index. Thus, all indices can be stored in a list of length $n + 1$. Figure 3 shows a visualization of the adjacency list and CSR representations.

Iterating the outgoing edges of a vertex consists of iterating the edge index range of that vertex. To identify a single edge, only the index of the edge in the edge list is required. And unlike the adjacency list representation, CSR makes all edge information lookups a single memory access. It also improves spatial locality of the edge information in memory, which might lead to preferable cache access patterns. Additionally, the overall memory consumption of a CSR encoded graph is lower than that of the same graph encoded with adjacency lists, as the memory overhead of a list per vertex is higher than that of a single index per vertex.

Constructing a flow graph in CSR representation for a given graph is more involved than using adjacency lists; two passes over the input graph are needed. The first pass computes a prefix sum of the outdegrees, whose output is the vertex index array for the CSR. The edge list is then allocated, and the second pass fills in all edge information. While the CSR construction does require an extra pass over the input graph compared to adjacency list construction, it requires only two allocations in total, whereas adjacency list construction needs at least one allocation per non-isolated vertex. Pseudocode for CSR construction is shown in Algorithm 6.

Unlike the previously existing Push-Relabel implementation, our Dinitz implementation uses a flow graph backed by a CSR based data structure.

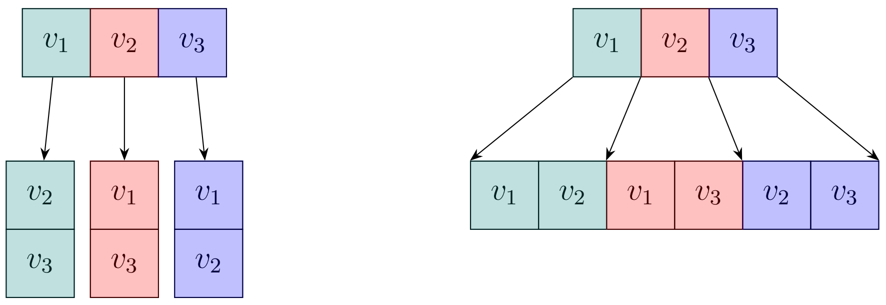

Figure 3: Visualization of adjacency list (left) and CSR (right) representations of the complete graph on three vertices. Neighborhoods of a vertex have the color corresponding to that vertex.

5.4 Identifying Full Critical Weighted Sets

As part of the CWIS reduction algorithm described in Section 4.5, the residual graph of the maximum flow must be scanned from s to discover all vertices in the critical weighted set. For a full graph scan, either a BFS or a DFS from s is required. For reasons unknown to us, the CWIS reduction implementation in KAMIS did not do such a full scan, but instead only used the set $\{v \in V : c_f(s, v) > 0\}$, considering it a critical weighted set. While this set is clearly a subset of the critical set that would be found by a full BFS, there is no guarantee that this subset itself is of critical weight. Considering the Critical Weighted Set problem as an instance of the Selection problem, one could easily imagine that the selection found by only considering direct neighbors of s is not a maximal selection, and is therefore not covered by the proofs in Chapter 4. Troublingly, using only a proper subset U_c' of the critical set U_c might break the correctness of the reduction itself. Since we compute the CWIS $I_c := U_c \backslash N(U_c)$, the independent set based on the subset U_c', that is $I_c' := U_c' \backslash N(U_c')$, might contain vertices of U_c' that were excluded from I_c, i.e. $I_c' \not\subseteq I_c$ might be possible. In this case, I_c' would no longer be guaranteed to be a subset of a MWIS on the input graph, making the reduction as implemented in KAMIS possibly incorrect. Note however that we do not know of any inputs for which the reduction as implemented actually leads to wrong results.

We alter the KAMIS implementation of the CWIS reduction to instead always do a full BFS on the residual graph of the maximum flow. Not only does this guarantee the correctness of the reduction, but it also maximizes the number of vertices which we reduce during the reduction itself. Very few of our input instances (see Section 6.2) show slight changes in the reduction offset of the reduction pipeline, but these are essentially noise and are to be expected when changing the CWIS reduction itself. They are not indicative of either version reducing more or less than the other in general.

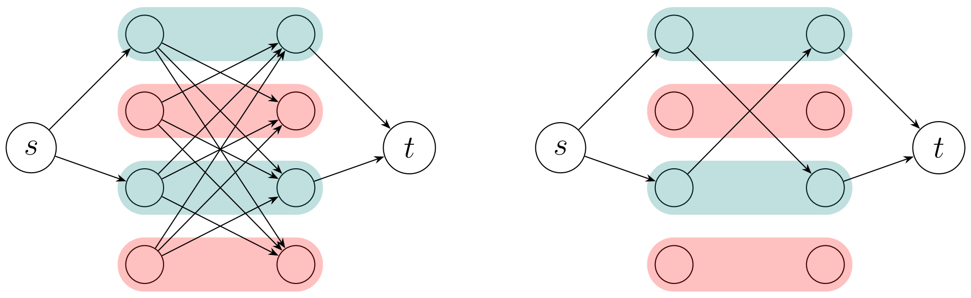

Figure 4: Visualization of the unpruned flow graph (left) and pruned flow graph (right) constructed for the complete graph on four vertices. The unset vertices and set vertices are highlighted in teal and red respectively.

5.5 Pruning the Flow Graph

To prevent having to rebuild huge almost identical graphs many times, data reductions in KaMIS do not build augmented copies of the input graph. Instead, KaMIS stores only the original input graph, and additionally tracks for each vertex if it has been included in or excluded from the current partial solution. We call a vertex *unset*, if it has not yet been included in or excluded from the current partial solution, or *set* if it has. Reductions in KaMIS then only operate on unset vertices, reducing the input graph by setting some unset vertices to be either included or excluded.

To build the flow graph for the CWIS reduction from this graph representation, KaMIS simply builds the flow graph as if all vertices were included in it, but omits any s-v and v'-t edges where v is a set vertex. By not allowing set vertices to receive any flow from the source or send any flow to the sink, maximum flows on this network have a clear correspondence to maximum flows on a network that would fully exclude all set vertices. The algorithm to find a CWIS works exactly the same, but construction of this network is simpler than construction of the network that includes only the necessary vertices.

While the correctness of the algorithm is preserved, performance of the maximum flow algorithms may still suffer, as many superfluous u-v' edges, where u or v is a set vertex, are still included in the network. And while there can never be flow through these edges, they are still included in the level graphs and may therefore significantly slow down the flow algorithms.

To prevent this, we make the very simple change of omitting any u-v' edges where at least one of u and v is a set vertex. Figure 4 visualizes the difference in the constructed flow networks. Note that set vertices are still included as isolated vertices in the flow graph. While it is possible to exclude these vertices by maintaining a mapping between vertex indices of the original graph and the compressed flow network, our measurements do not show performance improvements compared to our simpler flow network construction.

Algorithm 6: CSR construction of a flow graph for the CWIS reduction. Assumes that $V = \{0, \dots, n-1\}$. For the resulting flow network, $v' = v+n$ for each $v \in V$, $s = 2n$ and $t = 2n+1$. V_L and E_L are the index and edge lists of the CSR graph. An edge (u, v) is represented as a $(v, c(u,v), f(u,v),$ *index of* $(v,u))$ tuple.

```
Function AddEdges(E_L, P, u, v, c):
    $e \leftarrow P(u)$
    $e_r \leftarrow P(v)$
    $P(u) \leftarrow P(u) + 1$
    $P(v) \leftarrow P(v) + 1$
    $E_L(e) \leftarrow (v, c, 0, e_r)$
    $E_L(e_r) \leftarrow (u, 0, 0, e)$

Function BuildFlowGraph($G = (V, E), w$):
    $V_L \leftarrow$ List of length $2n + 3$.
    // Indices are offset by one for the prefix sum
    $V_L(s + 1) \leftarrow n$
    $V_L(t + 1) \leftarrow n$
    foreach $v \in V$ do
        // $v$ and $v'$ need an extra edge to source/sink
        $V_L(v + 1) \leftarrow \deg_{\text{out}}(v) + 1$
        $V_L(v + n + 1) \leftarrow \deg_{\text{out}}(v) + 1$
    foreach $i \in \{1, ..., 2n + 2\}$ do
        $V_L(i) \leftarrow V_L(i) + V_L(i - 1)$
    $E \leftarrow$ List of length $V_L(t + 1)$
    $P \leftarrow V_L$  // Progress array
    foreach $u \in V$ do
        AddEdges($E_L$, $P$, $s$, $u$, $w(u)$)
        AddEdges($E_L$, $P$, $u + n$, $t$, $w(u)$)
        foreach $v \in N(u)$ do
            AddEdges($E_L$, $P$, $u$, $v + n$, $\infty$)
    return $(V_L, E_L)$
```

CHAPTER 6

Experimental Evaluation

In this chapter, we present our experiments and findings. We describe the methodology with which the experiments were conducted in Section 6.1, and the datasets we used in Section 6.2. Section 6.3 describes our experiments and interprets their results.

6.1 Methodology

All experiments were conducted on a machine with an Intel Xeon Silver 4216 16-core processor and 93GB of main memory, running Ubuntu 20.04.1 LTS with Linux kernel 5.4.0-152-generic. All algorithms were written in C++ and integrated into KaMIS, which was compiled with g++ 9.4.0, using the -O3 optimization flag. Our code, along with raw timing data, is available at `https://github.com/MarkusEverling/KaMIS-cwis-dinitz/`.

One tool we use to compare different algorithms to each other are Performance Profiles [16]. These plots visualize for a set of algorithms how well each algorithm performs compared to the fastest one, for some fraction of input instances. The x-axis, labelled τ, represents a certain slowdown factor, and the y-axis represents the fraction of input instances for which a given algorithm was at most τ times slower than the fastest algorithm on that instance. The parameter τ ranges from 1 to the largest observed slowdown. Note also that while we draw continuous lines through the data points, performance profiles are inherently discrete, and their granularity is limited by the number of instances included in the profile.

6.2 Datasets

For our experiments, we use a large set of instances, which have been used in existing publications to evaluate MWIS solvers, made up of the graphs used by Gellner et al. [24]

and by Gu et al. [28]. This set includes 3d meshes derived from simulations using the finite element method (fe) [40], instances from dual graphs of triangle meshes (mesh) [38], real-world graphs from OpenStreetMaps (osm) [6, 10, 36], large social networks from the Stanford Large Network Dataset Repository (snap) [35], as well as graphs from the SuiteSparse Matrix Collection (ssmc) [13, 39]. For some of the instances, each vertex weight was increased by one to avoid many zero weight vertices. For unweighted graphs, each vertex was assigned a random weight uniformly distributed in the range [1, 200]. Overall, the set contains 213 graphs. A complete listing of these graphs, along with their vertex and edge count, is provided in Table 3.

Additionally, we produce pre-reduced input graphs from the instances mentioned above. For this, we use the `weighted_reduce` executable from the KaMIS framework, passing the `--kernel` flag to output the reduced graph, and disabling the CWIS reduction and any reductions that come after it in the reduction pipeline. The resulting graphs are reduced exactly as far as they would be before the first application of the CWIS reduction, and are therefore well suited for performance measurements of the CWIS reduction in isolation. After removing graphs that fully vanish from the reduction, we are left with 112 reduced instances. These instances, along with their vertex and edge count, are listed in Table 4.

6.3 Results

We split our experimental evaluation into two broad categories: Comparing one invocation of the CWIS reduction in isolation, and comparing the entire reduction pipeline with and without using our CWIS reduction.

6.3.1 The CWIS Reduction in Isolation

To evaluate different flow graph algorithms with regards to their suitability for the CWIS reduction, we compare performance of the reduction in isolation, on realistic input instances. For this, we use the pre-reduced instances described in Section 6.2. As these instances are exactly what the inputs would be in a full reduction, performance differences measured on them should be indicative of performance differences on the whole reduction pipeline.

Included in the comparison are the existing Push-Relabel implementation as well as our implementations of Tidal Flow and Dinitz's algorithm. Additionally, we include a Push-Relabel implementation from the Boost.Graph library [8]. As part of the very popular Boost family of C++ libraries [7], this is meant to serve as a representative for what performance may be expected when using an existing "off-the-shelf" flow algorithm implementation. Boost.Graph helpfully provides a type trait based bridge that allows the user to provide their own flow graph type. For the experiments, we use the CSR based flow graph data structure described in Section 5.3. For each algorithm and each input instance, we run the CWIS reduction 10 times and calculate the mean runtime for that instance. The CWIS reduction consists of building the flow graph, finding a maximum flow, and identifying a CWIS from it.

Figure 5 shows performance profiles for the timings of the CWIS reduction. Since many of the input instances are quite small, and therefore quite fast to solve, we show not only a performance profile of all 112 input instances, but also one profile containing only the 33 instances for which the at least one algorithm took at least 100 milliseconds. Removing small instances helps remove noise that is more noticeable on extremely short timings. Additionally, the difference in performance on large inputs is far more important to optimize, as we want to be able to reduce larger instances more efficiently.

It is immediately visible from the profiles that our implementation of Dinitz's Algorithm is extremely competitive compared to all other tested algorithms. In almost 90% of all instances, and over 70% of the large instances, Dinitz's Algorithm is the fastest out of all tested algorithms. At its worst, it is roughly 40% slower than the fastest algorithm.

Trailing closely behind is Boost's Push-Relabel implementation. While it is quite fast in most instances, there is one outlier where it is over 28 times slower than the fastest algorithm. Notably, Boost's Push-Relabel is consistently outperforming the existing KaMIS Push-Relabel implementation. It is however not an apples-to-apples comparison on an algorithmic level, as we use a CSR flow graph for Boost, and KaMIS uses an adjacency list flow graph for its Push-Relabel implementation. As Boost has quite strict requirements on the API that needs to be implemented for a user supplied flow graph structure,

Tidal Flow and the existing Push-Relabel are both consistently performing significantly worse than the other two algorithms. In almost all large instances, Tidal Flow is at least 3 times slower than the fastest algorithm. While Push-Relabel somewhat keeps up with the faster algorithms on roughly half the instances, it also experiences large slowdowns on many instances. In the worst cases, Push-Relabel is roughly 17 times slower, and Tidal Flow is over 30 times slower than the fastest algorithm.

Table 1 shows the runtimes of the CWIS reduction for the same 33 large instances included in the performance profile. They are ordered by ascending runtime of the baseline Push-Relabel implementation. For each of the tested algorithms, we list the mean runtime in seconds of the CWIS reduction per instance. For each instance, the fastest algorithm is highlighted. At the bottom of the table, we list for each algorithm the geometric mean of the runtimes across all 33 instances. We choose the geometric mean to ensure that not only the largest times have a meaningful impact on the average. Additionally, we list for each algorithm the arithmetic mean of the speedup over the baseline Push-Relabel across all instances. It is immediately visible that our implementation of Dinitz's Algorithm is fastest for the majority of instances. The two Push-Relabel implementations are fastest for very few instances, while Tidal Flow is fastest for no instances.

Additionally, comparing the runtimes directly, Dinitz's Algorithm has the largest speedups for the largest instances. In particular, its largest speedup compared to the baseline is the very largest instance, where Push-Relabel finishes in 124 seconds and Dinitz's Algorithm finishes in just 7 seconds. There is only one instance each, for which Boost's Push-Relabel and KaMIS Push-Relabel are significantly faster than Dinitz's algorithm.

And while Tidal Flow does become a lot faster than the baseline Push-Relabel for many large instances, it is slower than Dinitz's Algorithm for all instances, and suffers from huge slowdowns for some instances, in one case taking 326 seconds where Dinitz takes only 33 seconds.[1]

The average speedup over the baseline reflects this: Dinitz's algorithm is the fastest, being more than four times faster than the baseline on average.

Observation: Dinitz's Algorithm is by far the best algorithm with regards to the CWIS reduction, especially for large instances. We therefore proceed only with Dinitz's Algorithm for further measurements.

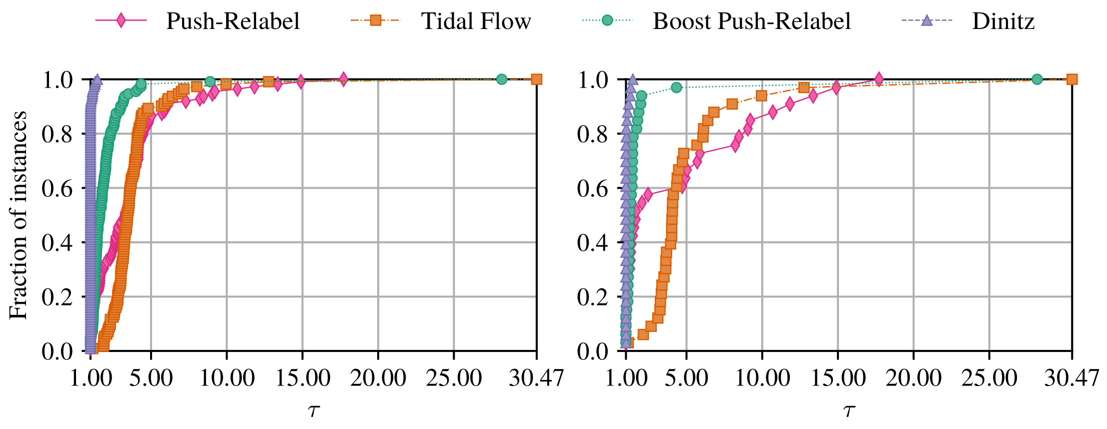

Figure 5: Performance profiles for the runtime of the CWIS reduction on pre-reduced inputs. The left profile contains all input graphs, the right profile contains only those inputs for which at least one algorithm took at least 100ms.

[1]Fontaine's results [21] suggest a much smaller gap in performance between Tidal Flow and Dinitz's Algorithm. We asked them if their implementations were available, but received no response.

Graph	**Push-Relabel**	**Tidal Flow**	**Dinitz**	**Boost**
snap_as-skitter-uniform	0.057	0.169	0.056	**0.043**
snap_web-BerkStan	0.067	0.201	0.056	**0.050**
snap_web-BerkStan-uniform	0.068	0.185	0.049	**0.046**
mesh_turtle-uniform	0.069	0.224	**0.062**	0.083
snap_roadNet-PA-uniform	**0.093**	0.373	0.094	0.108
snap_roadNet-PA	**0.093**	0.378	0.109	0.117
fe_sphere-uniform	0.098	0.030	**0.007**	0.207
snap_roadNet-TX-uniform	**0.108**	0.474	0.109	0.123
snap_soc-LiveJournal1-uniform	0.116	0.301	**0.069**	0.084
osm_virginia-AM3	0.144	0.104	**0.029**	0.051
osm_massachusetts-AM3	0.146	0.104	**0.030**	0.042
osm_kansas-AM3	0.154	0.090	**0.019**	0.026
snap_roadNet-CA-uniform	0.162	0.726	**0.154**	0.172
osm_district-of-columbia-AM2	0.204	0.194	**0.043**	0.074
fe_pwt-uniform	0.219	0.102	**0.089**	0.385
osm_vermont-AM3	0.249	0.173	**0.042**	0.052
mesh_buddha-uniform	0.316	1.390	**0.244**	0.274
ssmc_fl2010	0.332	1.220	**0.200**	0.289
mesh_dragonsub-uniform	0.345	2.167	**0.318**	0.366
mesh_ecat-uniform	**0.427**	2.615	0.439	0.447
ssmc_il2010	0.630	2.919	**0.456**	0.638
ssmc_ca2010	0.675	3.463	**0.432**	0.621
osm_washington-AM3	0.893	0.512	**0.156**	0.317
fe_rotor-uniform	1.197	5.780	**0.581**	1.146
osm_oregon-AM3	1.465	0.531	**0.159**	0.209
osm_greenland-AM3	1.857	0.712	**0.219**	0.273
fe_ocean-uniform	**2.206**	67.203	3.213	3.153
osm_idaho-AM3	3.017	0.679	**0.255**	0.306
osm_district-of-columbia-AM3	11.938	4.856	**1.319**	2.441
osm_rhode-island-AM3	14.066	3.187	**0.944**	1.387
snap_soc-pokec-relationships-uniform	30.968	326.081	33.088	**25.554**
osm_hawaii-AM3	49.149	14.269	**4.593**	5.647
osm_kentucky-AM3	124.247	15.061	**7.014**	8.404
Mean	0.558	0.873	**0.201**	0.287
Mean Speedup	1.000	1.416	**4.670**	3.177

Table 1: Runtimes in seconds for one invocation of the CWIS reduction on pre-reduced input graphs. The fastest time for each input is written in bold. Inputs for which all algorithms took at most 100 ms are excluded. The bottom rows show the geometric mean of each algorithm's runtimes, as well as the mean speedup of each algorithm over the baseline Push-Relabel.

6.3.2 The Full Reduction Pipeline

In addition to the measurements on the CWIS reduction in isolation, we also want to measure the impact that our faster reduction, as well as our other optimizations, have on the runtime of the entire reduction pipeline. As previously stated, we discard all algorithms other than the baseline Push-Relabel and our implementation of Dinitz's Algorithm.

To measure the runtime of the entire reduction pipeline, we use the `weighted_reduce` executable included in KaMIS. It takes in a graph as input, applies the whole reduction pipeline to it, and outputs various statistics about the reductions. The relevant statistics for us are reduction offset and runtime. The reduction offset is the difference between the weight of a MWIS on the reduced graph and the weight of a MWIS on the input graph. In other words, it is the weight of all vertices added in the reconstruction of the MWIS. Because different maximum flows can induce different minimum cuts, and because we changed the identification of the critical weighted set, as described in Section 5.4, our CWIS reduction might reduce different vertices than the baseline, which could then lead to changed behavior of other subsequent reductions and a different final reduction offset. In our measurements, very few instances had slight differences in reduction offsets for the different implementations, and there was no clear trend of one implementation leading to higher or lower reduction offsets than the other. As such, we consider the differing reduction offsets not meaningful to the algorithm's correctness or reduction quality, and instead look only at the runtime.

Figure 6 shows performance profiles for the runtimes of the `weighted_reduce` executable with the baseline CWIS reduction and with our Dinitz backed CWIS reduction. As before, we include one profile containing all 213 input graphs and one profile containing only instances for which the algorithms took a significant amount of time. In this case, we choose 2 seconds as the cutoff, leading to 32 instances on the right profile.

In both profiles, the version using Dinitz's Algorithm clearly outperforms the baseline. In roughly 70% of all instances, and almost 85% of large instances, the version using Dinitz's Algorithm is faster. Note that the graphs are a lot further apart on the plot containing only large instances. Including all instances leads to a lot more noise, as many small instances are solved very quickly. Additionally, many of the instances are fully reduced before they ever get to the CWIS reduction itself, in which case we do not expect any runtime difference between the versions. We still include those instances in the left profile for the sake of completeness, but the right profile, containing only the large instances, is a lot more relevant to real world applicability, as we mainly care about improving runtimes for instances which are difficult to solve in practice.

Focusing on the large instances, we see that the baseline version is more than 25% slower than our version for roughly half of the instances. Of the 32 instances, 7 are over 50% slower with the baseline version than with ours. In the worst case, the baseline version is over 2.3 times slower than ours.

In contrast, there are only 5 instances where the baseline outperforms our version using Dinitz's Algorithm. Of those, 4 are roughly 25% slower with our version than the with baseline, while the slowest one is almost 2 times slower.

Table 2 shows the runtime of the reduction pipeline reported by the `weighted_reduce` executable for the same 32 large instances that were included in the right performance profile in Figure 6, i.e. for all instances for which at least one of the versions took at least 2 seconds. The rows are sorted by ascending runtime of the baseline. As before, we give the runtime of each version in seconds for all instances, with faster times being highlighted. We also list the speedup of our version compared to the baseline for each instance. The bottom row of the table contains the geometric means of the runtimes across all instances, as well as the arithmetic mean of the speedup.

Comparing the runtimes, our version leads to very large speedups for many of the slowest input instances. The slowest instance, the graph osm_kentucky-AM3, takes over 850 seconds to fully reduce with the baseline version, but only 360 seconds with our version. Note that this speedup, with a factor of 2.36, is the largest speedup across all listed instances.

Additionally, some instances on which the CWIS reduction in isolation was slower using Dinitz's Algorithm, such as snap_soc-pokec-relationships-uniform, still show faster runtimes across the entire reduction pipeline with our version. We suspect that this is due to our changes to the critical weighted set identification, described in Section 5.4.

Another thing that the table shows is that 4 out of the 5 instances for which the baseline is faster belong to the finite element (fe) group of graphs. It may be worthwhile to investigate the structure of these graphs more closely, to understand the performance discrepancy between them and the rest of the input instances better. Our version still performs much better across the whole dataset, with a mean speedup of 30% on the large instances.

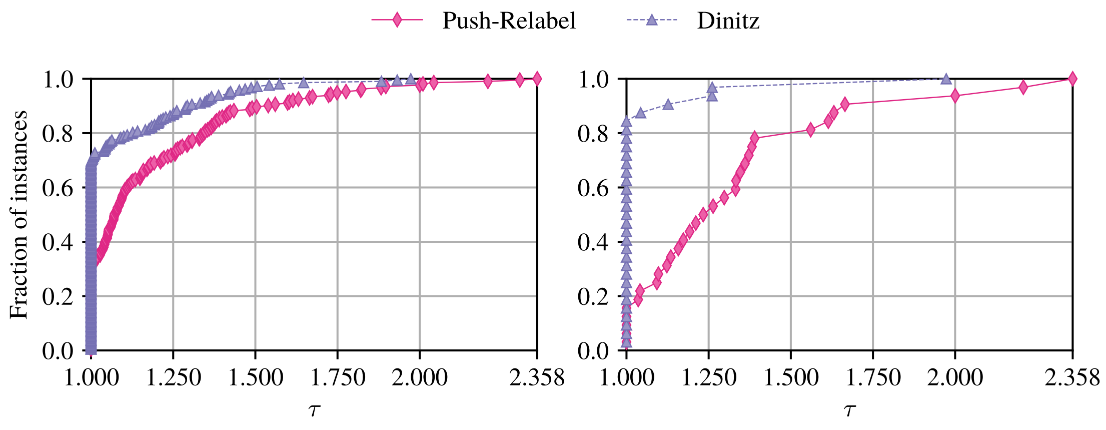

Figure 6: Performance profiles for the runtime of the `weighted_reduce` executable. The left profile contains all inputs, the right profile contains only those inputs for which at least one algorithm took at least 2s.

Graph	**Push-Relabel** (s)	**Dinitz** (s)	**Speedup**
fe_sphere-uniform	**1.325**	2.615	0.507×
snap_roadNet-PA	2.042	**1.861**	1.097×
snap_roadNet-TX-uniform	2.138	**1.904**	1.123×
mesh_ecat-uniform	2.158	**1.975**	1.093×
osm_west-virginia-AM3	2.415	**1.812**	1.333×
osm_alabama-AM3	2.602	**2.006**	1.297×
snap_web-BerkStan	2.715	**2.621**	1.036×
fe_pwt-uniform	**2.987**	3.765	0.793×
osm_puerto-rico-AM3	3.158	**2.285**	1.382×
fe_ocean-uniform	**3.278**	4.133	0.793×
snap_roadNet-CA-uniform	3.685	**3.142**	1.173×
mesh_buddha-uniform	3.782	**3.332**	1.135×
osm_district-of-columbia-AM2	3.976	**3.284**	1.211×
snap_web-BerkStan-uniform	**4.130**	4.309	0.959×
osm_florida-AM3	4.439	**3.333**	1.332×
ssmc_fl2010	6.545	**5.307**	1.233×
osm_greenland-AM3	7.503	**4.648**	1.614×
osm_mexico-AM3	7.582	**5.525**	1.372×
osm_oregon-AM3	8.802	**5.287**	1.665×
osm_virginia-AM3	13.349	**9.916**	1.346×
osm_washington-AM3	13.546	**9.957**	1.360×
ssmc_il2010	14.328	**12.019**	1.192×
osm_idaho-AM3	16.074	**8.036**	2.000×
fe_rotor-uniform	**16.296**	18.364	0.887×
ssmc_ca2010	19.709	**15.598**	1.264×
snap_as-skitter-uniform	32.737	**23.554**	1.390×
snap_soc-LiveJournal1-uniform	49.400	**42.667**	1.158×
osm_rhode-island-AM3	63.372	**28.706**	2.208×
osm_district-of-columbia-AM3	79.910	**51.208**	1.560×
osm_hawaii-AM3	229.668	**140.804**	1.631×
snap_soc-pokec-relationships-uniform	589.134	**565.899**	1.041×
osm_kentucky-AM3	852.455	**361.556**	2.358×
Mean	10.324	**8.299**	1.298×

Table 2: Runtime for one invocation of the whole `weighted_reduce` pipeline. The fastest time for each input is written in bold. The rightmost column shows the speedup of the reduction pipeline using Dinitz's Algorithm versus Push-Relabel. Inputs for which both algorithms took less than 2 s are excluded. The bottom row shows the geometric mean of each algorithm's runtimes, as well as the mean speedup.

CHAPTER 7

Discussion

In this final chapter, we provide a summary of our work, and of our experimental results, in Section 7.1, and give a list of suggestions for future work in Section 7.2.

7.1 Conclusion

In this work, we investigate the performance differences between different maximum flow algorithms, in the context of the Critical Weighted Independent Set reduction for MAXIMUM WEIGHT INDEPENDENT SET solvers. To do this, we integrate implementations of the Tidal Flow algorithm [21] and Dinitz's Algorithm [15] into the codebase of the MWIS solver KAMIS [1], as the findings by the author of Tidal Flow suggest that these algorithms are well suited for the shape of flow graph produced in the CWIS reduction. Additionally, we introduce several other optimizations related to the used data structures, pruning the constructed flow graph, and identification of the critical weighted set.

We compare the runtimes of the CWIS reduction in isolation on pre-reduced input graphs, using KAMIS's Push-Relabel, our Tidal Flow and Dinitz implementations, as well as the Push-Relabel implementation from the Boost.Graph library [8].

In the majority of test instances, our implementation of Dinitz's Algorithm is the fastest, with the two Push-Relabel implementations being faster for very few instances. Tidal Flow is the fastest for no instances. We therefore recommend to implementors of the CWIS reduction to use Dinitz's Algorithm, as shown in Algorithm 7 with an efficient flow graph data structure as described in Section 5.3.

To measure the impact of the faster CWIS reduction on the whole reduction pipeline, we compare the runtime of KAMIS's `weighted_reduce` program using the original CWIS reduction and our version. We measure significant speedups across many input instances, with a positive correlation between instance size and speedup. The highest speedup occurs on the largest instance, where our version is more than 2.3 times faster than the baseline, reducing the runtime by almost 500 seconds. On average, reducing large instances becomes 30% faster with our version of the CWIS reduction.

7.2 Future Work

There are still many aspects of the CWIS reduction which warrant attention. One very promising aspect is that of parallelization. As single-threaded performance improvements are slowing down with each generation of CPUs, manufacturers are instead scaling horizontally, packing dozens if not hundreds of cores into one CPU. Taking advantage of this parallelism could lead to large performance improvements for the CWIS reduction, compared to our current single threaded implementation. The construction of the flow graph, as well as the graph scan to identify a critical weighted set, are parallelizable using parallel prefix sum and BFS implementations; it would remain to determine a parallel flow graph algorithm that is well suited for the kind of flow network produced in the CWIS reduction.

It might also be possible to choose dynamically between different flow graph algorithms, based on heuristics of the constructed flow graph. While Dinitz's Algorithm is fastest for most instances in our dataset, it is still getting beaten by Push-Relabel on some instances. Identifying cheaply computable heuristics that predict which maximum flow algorithm is likely to be the fastest might further improve overall performance.

While the minimum cut construction described in Section 4.4.2 always finds a critical weighted set, the found set is not guaranteed to have maximum cardinality compared to all other critical weighted sets. An interesting question is whether it is possible to extract more information from the computed maximum flow. Choosing higher cardinality critical weighted sets might lead to higher cardinality critical weighted independet sets, which would then reduce more vertices, resulting in a smaller reduced graph.

Finally, with a much faster CWIS reduction, it might be beneficial to investigate the order of reductions in the reduction pipeline itself. As the CWIS reduction dominates some other local reductions, one invocation of the CWIS reduction might be able to replace many invocations of simpler reductions.

Zusammenfassung

Das MAXIMUM WEIGHT INDEPENDENT SET (MWIS) Problem ist ein grundlegendes NP-schweres Problem, mit vielen verwandten Problemen wie dem MINIMUM WEIGHT VERTEX COVER Problem und mit zahlreichen praktischen Anwendungen. Für einen gegebenen ungerichteten Graphen mit Knotengewichten, lautet das Problem, eine Teilmenge der Knoten mit höchstmöglichem Gewicht zu identifizieren, so dass keine zwei der Knoten benachbart sind.

Um große Eingabeinstanzen effektiv lösen zu können, verwenden MWIS Solver sogenannte Data Reductions, welche versuchen, die Größe der zu lösenden Instanz zu reduzieren, ohne dabei die Optimalität der konstruierten Lösung zu verlieren. Eine mächtige aber rechnerisch aufwändige Data Reduction ist die Critical Weighted Independent Set Reduction, welche ein Critical Weighted Set auf dem Eingabegraphen identifiziert, indem sie einen Maximum Flow auf einem speziell konstruierten Flow Netzwerk berechnet. Ausgehend von diesem Critical Weighted Set berechnet sie dann ein Critical Weighted Independent Set, dessen Knoten dann alle vom Eingabegraphen entfernt werden können.

In dieser Arbeit implementieren wir in dem MWIS Solver KAMIS verschiedene Maximum Flow Algorithmen, sowie Optimierungen der für den Flow Graphen verwendeten Datenstruktur und der Konstruktion der Flow Graphen und vergleichen die Implementierungen mit Bezug auf die Tauglichkeit für Verwendung in der CWIS Reduction.

Bibliography

[1] KaMIS source code. `https://github.com/KarlsruheMIS/KaMIS/`.

[2] Alexander A. Ageev. On finding critical independent and vertex sets. *SIAM J. Discret. Math.*, 7(2):293–295, 1994.

[3] Ravindra K. Ahuja and James B. Orlin. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics (NRL)*, 38(3):413–430, 1991.

[4] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609:211–225, 2016.

[5] M. L. Balinski. Notes—on a selection problem. *Management Science*, 17(3):230–231, 1970.

[6] Lukas Barth, Benjamin Niedermann, Martin Nöllenburg, and Darren Strash. Temporal map labeling: a new unified framework with experiments. In Siva Ravada, Mohammed Eunus Ali, Shawn D. Newsam, Matthias Renz, and Goce Trajcevski, editors, *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 23:1–23:10. ACM, 2016.

[7] Boost C++ Libraries. `https://boost.org/libraries/1.89/`.

[8] Boost Graph Library. `https://boost.org/libraries/1.89/graph/`.

[9] Sergiy Butenko and Svyatoslav Trukhanov. Using critical sets to solve the maximum independent set problem. *Oper. Res. Lett.*, 35(4):519–524, 2007.

[10] Shaowei Cai, Wenying Hou, Jinkun Lin, and Yuanjie Li. Improving local search for minimum weight vertex cover by dynamic strategies. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1412–1418. ijcai.org, 2018.

[11] Boris V. Cherkassky. A fast algorithm for constructing a maximum flow through a network. In *Selected Topics in Discrete Mathematics: Proceedings of the Moscow Discrete Mathematics Seminar*, volume 1990, pages 23–30, 1972.

[12] Miroslav Chlebík and Janka Chlebíková. Crown reductions for the minimum weighted vertex cover problem. *Discret. Appl. Math.*, 156(3):292–312, 2008.

[13] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[14] Yefim Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 01 1970.

[15] Yefim Dinitz. Dinitz' algorithm: The original version and Even's version. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Theoretical Computer Science, Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, pages 218–240. Springer, 2006.

[16] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002.

[17] Yuanyuan Dong, Andrew V. Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio G. C. Resende, and Quico Spaen. New instances for maximum weight independent set from a vehicle routing application, 2021.

[18] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[19] Shimon Even. *Graph Algorithms*. Cambridge University Press, 2 edition, 2011.

[20] Eric Filiol, Edouard Franc, Alessandro Gubbioli, Benoit Moquet, and Guillaume Roblot. Combinatorial optimisation of worm propagation on an unknown network. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 1:2931 – 2937, 08 2007.

[21] Matthew Fontaine. Tidal flow: A fast and teachable maximum flow algorithm. *Olympiads in Informatics*, 12:25–41, 05 2018.

[22] Lester Randolph Ford and D. R. Fulkerson. *Flows in Networks*, pages 1–35. Princeton University Press, Princeton, 1962.

[23] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 128–142. SIAM, 2021.

[24] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 128–142. SIAM, 2021.

[25] Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum flow problem. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 136–146. ACM, 1986.

[26] Ernestine Großmann, Sebastian Lamm, Christian Schulz, and Darren Strash. Finding near-optimal weight independent sets at scale. In Sara Silva and Luís Paquete, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2023, Lisbon, Portugal, July 15-19, 2023*, pages 293–302. ACM, 2023.

[27] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. Accelerating reductions using graph neural networks and a new concurrent local search for the maximum weight independent set problem. *CoRR*, abs/2412.14198, 2024.

[28] Jiewei Gu, Weiguo Zheng, Yuzheng Cai, and Peng Peng. Towards computing a near-maximum weighted independent set on massive graphs. In Feida Zhu, Beng Chin Ooi, and Chunyan Miao, editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 467–477. ACM, 2021.

[29] Lester Jr and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 01 1956.

[30] KAHIP. `https://kahip.github.io/`.

[31] Alexander Karzanov. Determining the maximal flow in a network by the method of preflows. *Doklady Mathematics*, 15:434–437, 02 1974.

[32] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 144–158. SIAM, 2019.

[33] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi,

and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010.

[34] Kenneth Langedal, Johannes Langguth, Fredrik Manne, and Daniel Thilo Schroeder. Efficient minimum weight vertex cover heuristics using graph neural networks. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[35] Jure Leskovec and Andrej Krevl. Stanford Large Network Dataset Collection. `https://snap.stanford.edu/data/`.

[36] OpenStreetMap. `https://openstreetmap.org/`.

[37] John Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17:200–207, 11 1970.

[38] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):144, 2008.

[39] SuiteSparse Matrix Collection. `https://sparse.tamu.edu/`.

[40] Chris Walshaw. Graph partitioning archive. `https://chriswalshaw.co.uk/partition/`.

[41] Mingyu Xiao, Sen Huang, Yi Zhou, and Bolin Ding. Efficient reductions and a fast algorithm of maximum weighted independent set. In *Proceedings of the Web Conference 2021*, WWW '21, page 3930–3940, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Changyou Xing, Guomin Zhang, and Ming Chen. Research on universal network performance testing model. In *2007 International Symposium on Communications and Information Technologies*, pages 780–784, 2007.

Algorithms

Algorithm 7: Full implementation of Dinitz's Algorithm. Takes in a flow graph F and a feasible flow f, and modifies f in place to be maximum.

```
Function ComputeRanks($F = ((V, E), c, s, t)$, $f$):
    $Q \leftarrow$ Empty FIFO Queue
    foreach $v \in V$ do $d(v) \leftarrow 0$
    $Q$.enqueue($s$)
    $d(s) \leftarrow 1$
    while $Q$ is not empty do
        $u \leftarrow Q$.dequeue()
        if $u = t$ then break
        foreach $v \in N(u)$ do
            if $f(u, v) < c(u, v)$ and $d(v) = 0$ then
                $d(v) \leftarrow d(u) + 1$
                $Q$.enqueue($v$)
    return $d$
Function Augment($F = ((V, E), c, s, t)$, $f$, $p$, $d$, $u$, $\varepsilon$):
    if $u = t$ then return $\varepsilon$
    while $p(u) \leq \deg_{\text{out}}(u)$ do
        $v \leftarrow p(u)$-th neighbor of $u$
        $r \leftarrow c(u, v) - f(u, v)$
        if $r > 0$ and $d(v) = d(u) + 1$ then
            $\varepsilon' \leftarrow$ Augment($F$, $f$, $p$, $d$, $v$, $\min\{\varepsilon, r\}$)
            if $\varepsilon' > 0$ then
                $f(u, v) \leftarrow f(u, v) + \varepsilon'$
                $f(v, u) \leftarrow f(v, u) - \varepsilon'$
                return $\varepsilon'$
        $p(u) \leftarrow p(u) + 1$
    return $0$
Function Dinitz($F = ((V, E), c, s, t)$, $f$):
    loop
        $d \leftarrow$ ComputeRanks($F$, $f$)
        if $d(t) = 0$ then return
        foreach $v \in V$ do $p(v) \leftarrow 1$
        loop
            if Augment($F$, $f$, $p$, $d$, $s$, $\infty$) $= 0$ then break
```

Algorithm 8: Full implementation of the Tidal Flow algorithm. Arguments are the same as in Dinitz's Algorithm above.

```
Function ComputeLevelGraphEdges($F = ((V, E), c, s, t)$, $f$):
    foreach $v \in V$ do $d(v) \leftarrow 0$
    $E_L \leftarrow$ Empty List,   $Q \leftarrow$ Empty FIFO Queue
    $d(s) \leftarrow 1$,   $Q$.enqueue($s$)
    while $Q$ is not empty do
        $u \leftarrow Q$.dequeue()
        foreach $v \in N(u)$ do
            if $f(u, v) = c(u, v)$ then continue
            if $d(v) = 0$ then
                $d(v) \leftarrow d(u) + 1$
                if $d(t) = 0$ then $Q$.enqueue($v$)
            if $d(v) = d(u) + 1$ then $E_L$.push($(u, v)$)
    return $E_L$
Function TideCycle($V$, $E_L$, $c$, $s$, $t$, $f$):
    foreach $v \in V$ do $h(v) \leftarrow 0, l(v) \leftarrow 0$
    $h(s) \leftarrow \infty$
    foreach $(u, v) \in E_L$ do
        $p(u, v) \leftarrow \min\{c(u, v) - f(u, v), h(u)\}$
        $h(v) \leftarrow h(u) + p(u, v)$
    if $h(t) = 0$ then return 0
    $l(t) \leftarrow h(t)$
    foreach $(u, v) \in E_L$ in reverse order do
        $p(u, v) \leftarrow \min\{p(u, v), h(u) - l(u), l(v)\}$
        $l(v) \leftarrow l(v) - p(u, v)$,   $l(u) \leftarrow l(u) + p(u, v)$
    foreach $v \in V$ do $h(v) \leftarrow 0$
    $h(s) \leftarrow l(s)$
    foreach $(u, v) \in E_L$ do
        $p(u, v) \leftarrow \min\{p(u, v), h(u)\}$
        $h(u) \leftarrow h(u) - p(u, v)$,   $h(v) \leftarrow h(v) + p(u, v)$
        $f(u, v) \leftarrow f(u, v) + p(u, v)$,   $f(v, u) \leftarrow f(v, u) - p(u, v)$
    return $h(t)$
Function TidalFlow($F = ((V, E), c, s, t)$, $f$):
    loop
        $E_L \leftarrow$ ComputeLevelGraphEdges($F$, $f$)
        if $t$ is not reachable from $s$ in $E_L$ then return
        loop
            if TideCycle($V$, $E_L$, $c$, $s$, $t$, $f$) $= 0$ then break
```

Datasets

1 Unreduced Graphs

Graph	n	m	**File Size**
fe_sphere-uniform	16,386	49,152	566.35 KiB
fe_pwt-uniform	36,519	144,794	1.69 MiB
fe_body-uniform	45,087	163,734	1.95 MiB
fe_ocean-uniform	143,437	409,593	5.34 MiB
fe_rotor-uniform	99,617	662,431	7.76 MiB
mesh_cow-uniform	5,036	7,366	85.83 KiB
mesh_venus-uniform	5,672	8,508	99.02 KiB
mesh_fandisk-uniform	8,634	12,818	151.08 KiB
mesh_blob-uniform	16,068	24,102	304.24 KiB
mesh_gargoyle-uniform	20,000	30,000	386.65 KiB
mesh_face-uniform	22,871	34,054	444.25 KiB
mesh_feline-uniform	41,262	61,893	832.37 KiB
mesh_gameguy-uniform	42,623	63,850	859.99 KiB
mesh_bunny-uniform	68,790	103,017	1.37 MiB
mesh_dragon-uniform	150,000	225,000	3.18 MiB
mesh_turtle-uniform	267,534	401,178	5.92 MiB
mesh_dragonsub-uniform	600,000	900,000	13.68 MiB
mesh_ecat-uniform	684,496	1,026,744	15.65 MiB
mesh_buddha-uniform	1,087,716	1,631,574	25.31 MiB
osm_delaware-AM1	2	1	21 B
osm_indiana-AM1	2	1	21 B
osm_indiana-AM2	2	1	21 B
osm_new-mexico-AM3	3	3	31 B
osm_new-mexico-AM2	3	3	31 B
osm_new-mexico-AM1	3	3	31 B
osm_delaware-AM2	3	3	32 B

osm_new-jersey-AM2	4	6	47 B
osm_new-jersey-AM1	4	6	47 B
osm_new-jersey-AM3	4	6	47 B
osm_indiana-AM3	4	6	49 B
osm_delaware-AM3	5	9	64 B
osm_missouri-AM1	10	6	80 B
osm_wyoming-AM1	7	11	87 B
osm_wyoming-AM2	8	16	110 B
osm_missouri-AM2	13	12	122 B
osm_missouri-AM3	17	24	205 B
osm_arkansas-AM1	26	19	230 B
osm_wyoming-AM3	12	42	253 B
osm_alaska-AM1	31	31	333 B
osm_maine-AM1	38	29	351 B
osm_tennessee-AM1	49	39	460 B
osm_nebraska-AM1	40	46	464 B
osm_wisconsin-AM1	54	51	567 B
osm_puerto-rico-AM1	60	63	648 B
osm_mississippi-AM1	74	60	667 B
osm_south-carolina-AM1	75	69	767 B
osm_new-york-AM1	42	118	873 B
osm_ohio-AM1	78	96	935 B
osm_nevada-AM1	89	93	951 B
osm_connecticut-AM1	87	96	958 B
osm_california-AM1	77	130	1.09 KiB
osm_alaska-AM2	54	156	1.14 KiB
osm_minnesota-AM1	86	136	1.15 KiB
osm_west-virginia-AM1	65	150	1.17 KiB
osm_michigan-AM1	133	112	1.29 KiB
osm_north-carolina-AM1	93	150	1.30 KiB
osm_iowa-AM1	90	164	1.36 KiB
osm_arkansas-AM2	55	233	1.49 KiB
osm_montana-AM1	109	194	1.58 KiB
osm_wisconsin-AM2	89	219	1.65 KiB
osm_maine-AM2	81	243	1.72 KiB
osm_illinois-AM1	113	202	1.73 KiB
osm_maryland-AM1	104	216	1.74 KiB
osm_louisiana-AM1	157	181	1.85 KiB
osm_idaho-AM1	136	208	1.87 KiB
osm_colorado-AM1	128	232	2.01 KiB
osm_greenland-AM1	77	341	2.21 KiB
osm_canada-AM1	189	240	2.39 KiB

osm_pennsylvania-AM1	193	276	2.71 KiB
osm_tennessee-AM2	100	418	2.88 KiB
osm_mississippi-AM2	151	366	2.96 KiB
osm_new-hampshire-AM1	195	302	3.00 KiB
osm_utah-AM1	230	309	3.11 KiB
osm_alaska-AM3	86	475	3.12 KiB
osm_mexico-AM1	175	358	3.14 KiB
osm_vermont-AM1	128	418	3.28 KiB
osm_kansas-AM1	190	400	3.73 KiB
osm_wisconsin-AM3	136	588	4.24 KiB
osm_georgia-AM1	294	434	4.54 KiB
osm_nebraska-AM2	93	734	4.70 KiB
osm_south-carolina-AM2	165	713	5.15 KiB
osm_alabama-AM1	320	581	5.63 KiB
osm_maine-AM3	143	850	6.11 KiB
osm_michigan-AM2	241	750	6.35 KiB
osm_connecticut-AM2	211	975	7.06 KiB
osm_iowa-AM2	155	954	7.17 KiB
osm_arkansas-AM3	103	1,376	8.17 KiB
osm_mississippi-AM3	242	1,116	8.63 KiB
osm_oregon-AM1	381	996	9.02 KiB
osm_massachusetts-AM1	413	1,089	9.35 KiB
osm_puerto-rico-AM2	165	1,285	10.00 KiB
osm_florida-AM1	475	1,277	11.47 KiB
osm_hawaii-AM1	411	1,423	12.23 KiB
osm_nevada-AM2	242	1,531	12.41 KiB
osm_nebraska-AM3	145	2,168	13.67 KiB
osm_virginia-AM1	570	1,480	13.82 KiB
osm_ohio-AM2	211	1,815	14.12 KiB
osm_colorado-AM2	283	2,026	15.61 KiB
osm_illinois-AM2	261	2,138	16.98 KiB
osm_rhode-island-AM1	455	1,973	17.15 KiB
osm_minnesota-AM2	253	2,580	18.83 KiB
osm_kentucky-AM1	381	2,402	19.38 KiB
osm_michigan-AM3	376	2,459	19.97 KiB
osm_washington-AM1	713	2,316	21.04 KiB
osm_california-AM2	231	3,074	21.52 KiB
osm_tennessee-AM3	212	3,215	23.72 KiB
osm_canada-AM2	449	2,947	23.85 KiB
osm_louisiana-AM2	436	3,111	25.26 KiB
osm_new-hampshire-AM2	514	3,369	27.24 KiB
osm_connecticut-AM3	367	3,769	28.93 KiB

osm_pennsylvania-AM2	521	3,812	30.67 KiB
osm_south-carolina-AM3	317	4,508	33.58 KiB
osm_montana-AM2	307	5,154	33.76 KiB
osm_maryland-AM2	316	4,715	34.21 KiB
osm_utah-AM2	589	4,692	38.33 KiB
osm_new-york-AM2	224	6,399	45.69 KiB
osm_west-virginia-AM2	317	8,328	61.69 KiB
osm_georgia-AM2	746	7,753	63.24 KiB
osm_colorado-AM3	538	8,365	63.47 KiB
osm_mexico-AM2	516	9,411	73.60 KiB
osm_north-carolina-AM2	398	10,116	75.68 KiB
osm_ohio-AM3	482	11,376	85.48 KiB
osm_nevada-AM3	569	15,016	118.66 KiB
osm_kansas-AM2	602	16,474	130.57 KiB
osm_florida-AM2	1,254	16,936	139.31 KiB
osm_new-hampshire-AM3	1,107	18,021	141.27 KiB
osm_alabama-AM2	1,164	19,386	159.77 KiB
osm_canada-AM3	943	20,241	160.14 KiB
osm_puerto-rico-AM3	494	26,926	206.17 KiB
osm_california-AM3	587	27,536	208.01 KiB
osm_pennsylvania-AM3	1,148	26,464	221.95 KiB
osm_district-of-columbia-AM1	2,500	24,651	224.21 KiB
osm_minnesota-AM3	683	34,188	265.66 KiB
osm_idaho-AM2	552	35,221	274.77 KiB
osm_massachusetts-AM2	1,339	35,449	282.13 KiB
osm_vermont-AM2	766	37,607	289.18 KiB
osm_louisiana-AM3	1,162	37,077	300.54 KiB
osm_utah-AM3	1,339	42,872	345.37 KiB
osm_mexico-AM3	1,096	47,131	372.97 KiB
osm_greenland-AM2	686	50,218	379.65 KiB
osm_oregon-AM2	1,325	57,517	484.08 KiB
osm_montana-AM3	837	69,293	519.02 KiB
osm_virginia-AM2	2,279	60,040	548.80 KiB
osm_georgia-AM3	1,680	74,126	640.87 KiB
osm_new-york-AM3	837	88,728	687.54 KiB
osm_maryland-AM3	1,018	95,415	741.17 KiB
osm_west-virginia-AM3	1,185	125,620	1,006.88 KiB
osm_florida-AM3	2,985	154,043	1.38 MiB
osm_washington-AM2	3,025	152,449	1.41 MiB
osm_north-carolina-AM3	1,557	236,739	1.93 MiB
osm_hawaii-AM2	2,875	265,158	2.47 MiB
osm_rhode-island-AM2	2,866	295,488	2.55 MiB

osm_alabama-AM3	3,504	309,664	2.87 MiB
osm_massachusetts-AM3	3,703	551,491	5.07 MiB
osm_kentucky-AM2	2,453	643,428	5.51 MiB
osm_virginia-AM3	6,185	665,903	6.13 MiB
osm_kansas-AM3	2,732	806,912	7.42 MiB
osm_vermont-AM3	3,436	1,136,164	10.13 MiB
osm_district-of-columbia-AM2	13,597	1,609,795	15.66 MiB
osm_washington-AM3	10,022	2,346,213	22.39 MiB
osm_oregon-AM3	5,588	2,912,701	27.47 MiB
osm_greenland-AM3	4,986	3,652,361	33.33 MiB
osm_idaho-AM3	4,064	3,924,080	35.57 MiB
osm_rhode-island-AM3	15,124	12,622,219	122.66 MiB
osm_district-of-columbia-AM3	46,221	27,729,137	311.80 MiB
osm_hawaii-AM3	28,006	49,444,921	542.74 MiB
osm_kentucky-AM3	19,095	59,533,630	611.31 MiB
snap_ca-GrQc-uniform	5,242	14,484	153.12 KiB
snap_ca-GrQc	5,241	14,484	153.28 KiB
snap_p2p-Gnutella08-uniform	6,301	20,777	211.20 KiB
snap_p2p-Gnutella09-uniform	8,114	26,013	268.77 KiB
snap_ca-HepTh-uniform	9,877	25,973	281.14 KiB
snap_p2p-Gnutella06-uniform	8,717	31,525	323.97 KiB
snap_p2p-Gnutella05-uniform	8,846	31,839	328.36 KiB
snap_p2p-Gnutella04-uniform	10,876	39,994	418.60 KiB
snap_p2p-Gnutella25-uniform	22,687	54,705	654.98 KiB
snap_p2p-Gnutella24-uniform	26,518	65,369	787.32 KiB
snap_wiki-Vote-uniform	7,115	100,762	960.81 KiB
snap_ca-CondMat-uniform	23,133	93,439	1.06 MiB
snap_ca-CondMat	23,133	93,439	1.06 MiB
snap_p2p-Gnutella30-uniform	36,682	88,328	1.06 MiB
snap_ca-HepPh-uniform	12,008	118,489	1.18 MiB
snap_p2p-Gnutella31-uniform	62,586	147,892	1.83 MiB
snap_email-Enron	36,692	183,831	1.88 MiB
snap_email-Enron-uniform	36,692	183,831	1.88 MiB
snap_ca-AstroPh-uniform	18,772	198,050	2.10 MiB
snap_soc-Epinions1-uniform	75,879	405,740	4.17 MiB
snap_email-EuAll-uniform	265,214	364,481	4.70 MiB
snap_soc-Slashdot0811-uniform	77,360	469,180	5.09 MiB
snap_soc-Slashdot0902-uniform	82,168	504,230	5.49 MiB
snap_loc-gowalla_edges	196,591	950,327	11.20 MiB
snap_com-amazon	334,863	925,869	12.85 MiB
snap_web-NotreDame	325,729	1,090,108	14.86 MiB

snap_web-NotreDame-uniform	325,729	1,090,108	14.86 MiB
snap_roadNet-PA-uniform	1,088,092	1,541,898	24.10 MiB
snap_roadNet-PA	1,088,092	1,541,898	24.10 MiB
snap_web-Stanford-uniform	281,903	1,992,636	26.06 MiB
snap_roadNet-TX-uniform	1,379,917	1,921,660	30.94 MiB
snap_com-youtube	1,134,890	2,987,624	40.64 MiB
snap_roadNet-CA-uniform	1,965,206	2,766,607	45.70 MiB
snap_web-Google-uniform	875,713	4,322,051	59.58 MiB
snap_wiki-Talk-uniform	2,394,385	4,659,565	62.76 MiB
snap_web-BerkStan-uniform	685,230	6,649,470	87.56 MiB
snap_web-BerkStan	685,230	6,649,470	87.56 MiB
snap_as-skitter-uniform	1,696,415	11,095,298	147.80 MiB
snap_soc-pokec-relationships-uniform	1,632,803	22,301,964	300.13 MiB
snap_soc-LiveJournal1-uniform	4,847,571	42,851,237	614.62 MiB
ssmc_ri2010	25,181	62,875	774.08 KiB
ssmc_nh2010	48,837	117,275	1.45 MiB
ssmc_ga2010	291,086	709,028	9.92 MiB
ssmc_il2010	451,554	1,082,232	15.47 MiB
ssmc_fl2010	484,481	1,173,147	16.35 MiB
ssmc_ca2010	710,145	1,744,683	25.25 MiB

Table 3: Full input graphs, used to measure the performance of `weighted_reduce`.

2 Reduced Graphs

Graph	n	m	**File Size**
fe_body-uniform	10,208	30,058	440.76 KiB
fe_sphere-uniform	14,793	41,533	638.91 KiB
fe_pwt-uniform	32,966	124,071	1.92 MiB
fe_ocean-uniform	141,277	404,905	6.81 MiB
fe_rotor-uniform	98,001	638,164	9.92 MiB
mesh_cow-uniform	579	1,131	14.67 KiB
mesh_venus-uniform	1,007	1,902	25.16 KiB
mesh_fandisk-uniform	1,046	2,014	26.71 KiB
mesh_gargoyle-uniform	1,241	2,425	32.92 KiB
mesh_face-uniform	1,409	2,743	37.77 KiB
mesh_blob-uniform	1,677	3,218	45.21 KiB
mesh_gameguy-uniform	3,093	5,917	86.50 KiB
mesh_feline-uniform	4,438	8,652	128.17 KiB
mesh_bunny-uniform	9,529	18,379	277.59 KiB
mesh_dragon-uniform	10,309	19,671	298.95 KiB
mesh_turtle-uniform	25,301	49,120	805.84 KiB
mesh_buddha-uniform	80,026	153,857	2.56 MiB
mesh_ecat-uniform	87,769	170,303	2.83 MiB
mesh_dragonsub-uniform	92,517	178,945	2.98 MiB
osm_louisiana-AM2	6	10	111 B
osm_wisconsin-AM3	6	10	114 B
osm_california-AM1	7	12	139 B
osm_kansas-AM1	8	18	183 B
osm_mexico-AM1	9	20	213 B
osm_new-hampshire-AM2	9	21	216 B
osm_utah-AM2	8	22	220 B
osm_greenland-AM1	11	25	260 B
osm_oregon-AM2	11	32	318 B
osm_new-york-AM1	12	35	363 B
osm_maine-AM3	12	40	402 B
osm_washington-AM1	16	44	472 B
osm_canada-AM2	19	54	574 B
osm_north-carolina-AM2	19	69	703 B
osm_connecticut-AM3	19	101	997 B
osm_florida-AM2	28	106	1.10 KiB
osm_hawaii-AM1	30	126	1.31 KiB

osm_massachusetts-AM1	35	130	1.35 KiB
osm_michigan-AM3	36	135	1.41 KiB
osm_west-virginia-AM2	28	197	1.96 KiB
osm_kentucky-AM1	45	239	2.41 KiB
osm_mexico-AM2	34	270	2.64 KiB
osm_rhode-island-AM1	75	400	4.17 KiB
osm_south-carolina-AM3	79	630	6.31 KiB
osm_tennessee-AM3	67	736	7.37 KiB
osm_california-AM2	74	959	9.54 KiB
osm_alabama-AM2	118	1,052	11.06 KiB
osm_new-york-AM2	75	1,166	11.41 KiB
osm_nevada-AM3	102	1,246	12.58 KiB
osm_colorado-AM3	145	1,521	16.53 KiB
osm_massachusetts-AM2	184	3,389	36.66 KiB
osm_minnesota-AM3	165	4,063	42.05 KiB
osm_virginia-AM2	259	3,824	43.84 KiB
osm_maryland-AM3	222	4,136	45.51 KiB
osm_new-hampshire-AM3	275	4,454	49.47 KiB
osm_ohio-AM3	227	5,167	58.62 KiB
osm_vermont-AM2	159	6,600	70.40 KiB
osm_greenland-AM2	193	6,516	71.83 KiB
osm_hawaii-AM2	340	6,545	75.55 KiB
osm_washington-AM2	320	6,791	76.09 KiB
osm_louisiana-AM3	353	7,044	80.85 KiB
osm_pennsylvania-AM3	327	7,154	81.71 KiB
osm_canada-AM3	339	7,239	83.84 KiB
osm_district-of-columbia-AM1	746	7,960	93.90 KiB
osm_puerto-rico-AM3	223	9,403	104.77 KiB
osm_kentucky-AM2	300	11,389	127.13 KiB
osm_utah-AM3	529	18,108	206.42 KiB
osm_california-AM3	353	18,969	211.67 KiB
osm_mexico-AM3	521	24,479	277.08 KiB
osm_montana-AM3	377	37,060	416.99 KiB
osm_georgia-AM3	727	37,332	437.08 KiB
osm_new-york-AM3	544	51,957	590.16 KiB
osm_florida-AM3	1,047	62,452	719.99 KiB
osm_rhode-island-AM2	1,052	73,484	854.02 KiB
osm_west-virginia-AM3	977	108,648	1.24 MiB
osm_alabama-AM3	1,535	116,497	1.45 MiB
osm_north-carolina-AM3	1,058	149,557	1.70 MiB
osm_kansas-AM3	1,546	367,779	4.45 MiB
osm_massachusetts-AM3	1,999	371,448	4.61 MiB

osm_virginia-AM3	3,622	374,065	4.80 MiB
osm_district-of-columbia-AM2	5,734	533,846	6.83 MiB
osm_vermont-AM3	2,285	587,557	7.27 MiB
osm_oregon-AM3	3,694	2,033,546	26.77 MiB
osm_washington-AM3	7,930	2,064,840	27.17 MiB
osm_greenland-AM3	3,882	2,325,120	29.81 MiB
osm_idaho-AM3	3,137	2,725,557	34.43 MiB
osm_rhode-island-AM3	12,610	11,390,377	152.49 MiB
osm_district-of-columbia-AM3	31,917	15,907,976	236.60 MiB
osm_hawaii-AM3	23,906	39,708,565	580.22 MiB
osm_kentucky-AM3	16,977	54,313,331	757.93 MiB
snap_com-youtube	15	55	540 B
snap_wiki-Vote-uniform	337	1,764	20.38 KiB
snap_com-amazon	756	1,806	23.25 KiB
snap_loc-gowalla_edges	778	5,372	64.30 KiB
snap_web-Google-uniform	1,401	5,174	67.45 KiB
snap_web-NotreDame	2,778	12,409	172.49 KiB
snap_web-NotreDame-uniform	3,110	16,600	229.30 KiB
snap_web-Stanford-uniform	4,703	66,422	890.60 KiB
snap_roadNet-PA	31,981	64,253	1.04 MiB
snap_roadNet-PA-uniform	32,997	66,230	1.07 MiB
snap_as-skitter-uniform	14,841	76,823	1.11 MiB
snap_roadNet-TX-uniform	36,466	73,049	1.19 MiB
snap_web-BerkStan	15,744	94,511	1.35 MiB
snap_web-BerkStan-uniform	15,710	96,555	1.38 MiB
snap_roadNet-CA-uniform	50,356	101,522	1.66 MiB
snap_soc-LiveJournal1-uniform	15,476	151,571	2.13 MiB
snap_soc-pokec-relationships-uniform	878,495	11,707,095	198.95 MiB
ssmc_nh2010	5,827	14,262	206.07 KiB
ssmc_ri2010	6,517	15,846	230.11 KiB
ssmc_ga2010	16,075	36,920	570.78 KiB
ssmc_fl2010	72,276	169,052	2.73 MiB
ssmc_il2010	110,520	246,028	4.05 MiB
ssmc_ca2010	115,338	265,174	4.41 MiB

Table 4: Pre-reduced input graphs, used to measure the performance of one CWIS reduction.