SERIE B — INFORMATIK

A Simple Hypergraph Min Cut Algorithm[◇]

Regina Klimmek*
Frank Wagner**

B 96-02
March 1996

Abstract

We present an algorithm for finding the minimum cut of an edge-weighted hypergraph. It is simple in every respect. It has a short and compact description, is easy to implement and has a surprisingly simple proof of correctness. The runtime is $\mathcal{O}(|V|^2 \log |V| + |V| \cdot ||E||)$ where $||E||$ is the sum of the cardinalities of the hyperedges.

[◇]The second author is supported by a Heisenberg–Stipendium of the Deutsche Forschungsgemeinschaft.
*Fachbereich Mathematik, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin–Charlottenburg, Germany.
**Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin–Dahlem, Germany, e-mail: wagner@inf.fu-berlin.de.

1 Introduction

Graph connectivity is one of the classical subjects in graph theory, and has many practical applications, e. g. in chip and circuit design, reliability of communication networks, transportation planning and cluster analysis. Finding the minimum cut of an undirected edge-weighted graph is a fundamental algorithmical problem. Precisely, it consists in finding a nontrivial partition of a graph's vertex set V into two parts such that the sum of the weights of the edges connecting the two parts is minimum. Every such nontrivial partition is called a *cut*.

In most of the applications, especially in chip design [L90, chapter 5], [MWW95, Part 1] it is of much higher practical interest to cut *hypergraphs* into pieces, a more general structure where the graphs edges, connecting two vertices each, are replaced by hyperedges connecting an arbitrary subset of the vertices.

Formally the *hypergraph minimum cut problem* consists in minimizing the following weight function on the subsets A of V, given a set of hyperedges $E \subseteq 2^V$ that have positive weights $w(e)$:

$$w(A) = \sum \{w(e) \mid e \in E,\ e \cap A \neq \emptyset,\ e \cap V \setminus A \neq \emptyset\}$$

The usual approach to solve this problem for graphs is to use its close relationship to the maximum flow problem. Nagamochi and Ibaraki [NI92b] published the first rather complicated minimum cut algorithm that is not based on a flow algorithm having a running time of $\mathcal{O}(|V||E| + |V|^2 \log |V|)$. In the unweighted case they use a fast search technique to decompose a graph's edge set E into subsets $E_1, \ldots, E_\lambda$ such that the union of the first k E_i's is a k-edge-connected spanning subgraph of the given graph and has size at most $k|V|$. They simulate this approach in the weighted case. By avoiding the unnecessary simulated decomposition of the edge set, we present in [SW94] a remarkably simple minimum cut algorithm with the so far best deterministic running time established in [NI92b].

One possibility to solve the cut problem for hypergraphs would consist in modeling hypergraphs by graphs with the same cut properties as suggested in [L90, chapter 6.1.5]. But in [IWW93] it is shown that this approach does not work.

So far no efficient algorithm for the hypergraph mincut problem is known. Queyranne [Q95] generalizes the algorithm from [SW94] to the minimization of symmetric submodular functions. In this paper we first show that the hypergraph cut function w is symmetric and submodular, thus Queyranne's generalization yields an $\mathcal{O}(|V|^3 \cdot ||E||)$ algorithm where $||E||$ is the sum of the cardinalities of the hyperedges.

Then we show that a direct application of the technique from [SW94] leads to an $\mathcal{O}(|V|^2 \log |V| + |V| \cdot ||E||)$ algorithm.

2 The Submodularity

In this first section we show, that the cut function w in a hypergraph H is a symmetric submodular function. It is obvious that it is *symmetric*, i. e. $w(A) = w(V \setminus A)$ for every subset A of V.

Given two subsets A and B of V, w is *submodular* iff

$$w(A \cap B) + w(A \cup B) \leq w(A) + w(B)$$

We show this by distinguishing the hyperedges of H depending on their intersection with the four sets $A \setminus B$, $B \setminus A$, $A \cap B$ and $V \setminus (A \cup B)$:

$A\setminus B$	$B\setminus A$	$A\cap B$	$V\setminus(A \cup B)$	$w(A\cap B)$	$w(A\cup B)$	$w(A)$	$w(B)$
$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$
$\vert$	$\vert$	$\vert$		$\vert$		$\vert$	$\vert$
$\vert$	$\vert$		$\vert$		$\vert$	$\vert$	$\vert$
$\vert$		$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$
	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$
$\vert$	$\vert$					$\vert$	$\vert$
$\vert$		$\vert$		$\vert$			$\vert$
$\vert$			$\vert$		$\vert$	$\vert$	
	$\vert$	$\vert$		$\vert$		$\vert$	
	$\vert$		$\vert$		$\vert$		$\vert$
		$\vert$	$\vert$	$\vert$	$\vert$	$\vert$	$\vert$

The left half of the table lists the eleven different types of hyperedges, that have vertices in at least two of the four sets. A $|$ denotes a non-empty intersection. In the right half of the table a $|$ marks hyperedges that contribute to the cut above. One can see that those hyperedges that do not contribute twice to $w(A) + w(B)$, contribute just once to $w(A \cup B) + w(A \cap B)$, too. This shows the submodularity of w.

As is shown by Queyranne [Q95] such a function can be minimized using $\mathcal{O}(|V|^3)$ function evaluations. As for a given subset A of V we have to run through all the hyperedges to find $w(A)$, this yields an $\mathcal{O}(|V|^3 \cdot ||E||)$ algorithm for our problem.

3 The Faster Algorithm

We will show how to find a hypergraph mincut much faster by taking a direct approach. The simple hypergraph minimum cut algorithm we describe here consists of $|V| - 1$ phases:

MinimumCutPhase(G, w, a)
$A \leftarrow \{a\}$
while $A \neq V$
    add to A the most tightly connected vertex
store the cut-of-the-phase and shrink G by merging the two vertices added last

A subset A of the hypergraph's vertices grows starting with an arbitrary single vertex until A is equal to V. In each step the vertex outside of A *most tightly connected* with A is added. Formally, we add a vertex

$$z \notin A \text{ such that } w(A, z) = \max\{w(A, y) \mid y \notin A\}$$

where (A, y) is the set of the hyperedges e with $y \in e$ and $A \cap e \neq \emptyset$. $w(A, y)$ is the sum of the weights of the hyperedges in (A, y). At the end of each such phase the two vertices added last are *merged*, i.e., the two vertices are replaced by a new vertex. Hyperedges containing just the two merged nodes are removed.
The cut of V that separates the vertex added last from the rest of the hypergraph is called the *cut-of-the-phase*. The lightest of these cuts-of-the-phase is the result of the algorithm, the desired minimum cut. So overall the hypergraph mincut algorithm can be described as:

MinimumCut(G, w, a)
while $|V| > 1$
    MinimumCutPhase(G, w, a)
    if the cut-of-the-phase is lighter than the current minimum cut
        then store the cut-of-the-phase as the current minimum cut

Notice that the starting vertex a stays the same throughout the whole algorithm.

4 Correctness

The core of the proof of correctness is the following somewhat surprising lemma. An s-t cut is a cut that separates the two vertices s and t.

Lemma *Each cut-of-the-phase is a minimum s-t cut in the current hypergraph, where s and t are the two vertices added last in the phase.*

Proof: The run of a MinimumCutPhase orders the vertices of the current hypergraph linearly, starting with a and ending with s and t, according to their order of addition to A. Now, we look at an arbitrary s-t cut C of the current hypergraph and show, that it is at least as heavy as the cut-of-the-phase.

We call a vertex $v \neq a$ *active* (with respect to C) when v and the vertex added just before v are in the two different parts of C. Let $w(C)$ be the weight of C, A_v the set of all vertices added before v (excluding v), C_v the cut of $A_v \cup \{v\}$ induced by C, and $w(C_v)$ the weight of the induced cut, i.e., the sum of the weights of the hyperedges containing vertices from both parts of the induced partition. We show that for every active vertex v

$$w(A_v, v) \leq w(C_v)$$

by induction on the set of active vertices:
For the first active vertex the inequality is satisfied with equality. Let the inequality be true for all active vertices added up to the active vertex v, and let u be the next active vertex that is added. Then we have

$$w(A_u, u) = w(A_v, u) + w((A_u \setminus A_v, u) \setminus (A_v, u)) =: \alpha$$

Now, $w(A_v, u) \leq w(A_v, v)$ as v was chosen as the vertex most tightly connected with A_v. By induction $w(A_v, v) \leq w(C_v)$. All hyperedges in $(A_u \setminus A_v, u) \setminus (A_v, u)$ contribute to $w(C_u)$ but not to $w(C_v)$. So

$$\alpha \leq w(C_v) + w((A_u \setminus A_v, u) \setminus (A_v, u)) \leq w(C_u)$$

As t is always an active vertex with respect to C we can conclude that $w(A_t, t) \leq w(C_t)$ which says exactly that the cut-of-the-phase is at most as heavy as C. $\square$

Using the lemma we can show as in [SW94] by a simple case distinction, that the smallest of these cuts-of-the-phase is indeed the (unrestricted) minimum cut we are looking for:

Theorem *The smallest of the $|V| - 1$ cuts-of-the-phase considered during the algorithm is the minimum cut of the hypergraph.*

5 Running Time

As the running time of the algorithm MinimumCut is essentially equal to the added running time of the $|V| - 1$ runs of MinimumCutPhase, which is called on hypergraphs with decreasing number of vertices and hyperedges, it suffices to show that a single MinimumCutPhase needs at most $\mathcal{O}(||E|| + |V| \log |V|)$ time. This yields an overall running time of $\mathcal{O}(|V| \cdot ||E|| + |V|^2 \log |V|)$.
The key to implementing a phase efficiently is to make it easy to select the next vertex to be added to the set A, the most tightly connected vertex. During execution of a phase, all vertices that are not in A reside in a priority queue based on a key field. The key of a vertex v is the sum of the weights of the hyperedges connecting it to the current A, i.e., $w(A, v)$. Whenever a vertex v is added to A we have to perform an update of the queue. v has to be deleted from the queue, and for every hyperedge e that contains v and a vertex w, not in A, the key of w has to be increased by the weight of the hyperedge, if e is cut for the first time. Thus, the increase has

to performed if and only if v is the *first* vertex of e added to A. Else the weight of e is already counted in the key of w. We mark a hyperedge when its first vertex is moved to A. So, if a vertex V moves to A we have to perform changes of the keys as follows:

for all hyperedges e with $v \in e$ **do**
  if e is not marked **then**
    mark e
    for all $u \in e \setminus \{v\}$ **do** IncreaseKey by $w(e)$

In order to do this fast we need the hypergraph to be stored in the following way: Every vertex is linked to the hyperedges it is contained in, every hyperedge is linked to the vertices it contains and can be marked as "already touched".
The marker prevents us from using the links between the edges and the vertices more than once in each direction.

Overall we have to perform $|V|$ ExtractMax and $||E||$ IncreaseKey operations. Using Fibonacci heaps [FT87], we can perform an ExtractMax operation in $\mathcal{O}(\log |V|)$ amortized time and an IncreaseKey operation in $\mathcal{O}(1)$ amortized time.
Thus the time we need for this key step that dominates the rest of the phase, is $\mathcal{O}(||E|| + |V| \log |V|)$.

References

[FT87] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM **34** (1987) 596–615

[IWW93] E. Ihler, D. Wagner and F. Wagner, Information Processing Letters **45** (1993) 171–175

[L90] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, Chichester-Stuttgart (1990)

[MWW95] R. H. Möhring, D. Wagner and F. Wagner, *VLSI Network Design*, in: Handbook in Operations Research and Management Science, Volume 8 Networks, Elsevier Science, Amsterdam (1995) 625–712

[NI92b] H. Nagamochi and T. Ibaraki, *Computing edge-connectivity in multigraphs and capacitated graphs*, SIAM Journal on Discrete Mathematics **5** (1992) 54–66

[Q95] M. Queyranne, *A Combinatorial Algorithm for Minimizing Symmetric Submodular Functions*, Proceedings of the 6th ACM-SIAM Symposium on Discrete Mathematics (1995) 98–101

[SW94] M. Stoer, F. Wagner *A Simple Min Cut Algorithm*, Proceedings of the 2nd Annual European Symposium on Algorithms, Lecture Notes in Computer Science **855** (1994) 141–147