ELSEVIER

INTEGRATION, the VLSI journal 30 (2000) 1–11

INTEGRATION
theVLSI journal

www.elsevier.com/locate/vlsi

A fast hypergraph min-cut algorithm for circuit partitioning

Wai-Kei Mak[a], D.F. Wong[b,*,1]

[a]*Department of Computer Science and Engineering, ENB 118, University of South Florida, Tampa, FL 33620-5399, USA*
[b]*Department of Computer Sciences, TAY 2.124, University of Texas at Austin, Austin, TX 78712-1188, USA*

Received 24 January 2000

Abstract

Circuit partitioning is one of the central problems in VLSI system design. The primary objective of circuit partitioning is to minimize the number of interconnections between different components of the partitioned circuit. So the circuit partitioning problem is closely related to the minimum cut problem. Recently, two very fast algorithms for computing minimum cuts in graphs were reported (Nagamochi and Ibaraki; SIAM J. Discrete Math. 5(1) (1992) 54; Stoer and Wagner, J. ACM 44(4) (1997) 585). However, it is known that a circuit netlist cannot be accurately modeled by a graph, but only by a hypergraph. In this paper, we present the fastest algorithm known today for computing a minimum cut in a hypergraph which is a non-trivial extension of the result in Stoer and Wagner (J. ACM 44(4) (1997) 586). Since the netlist of a circuit can be modeled naturally as a hypergraph, this opens the opportunity for finding very-high-quality solutions for the circuit partitioning problem. Unlike most minimum cut algorithms which rely on flow computations in a network, ours is a non-flow-based algorithm.

Keywords: Circuit partitioning; Minimum cut; Hypergraph; Min-cut partitioning; Flow-based algorithm; Non-flow-based algorithm

1. Introduction

Circuit partitioning is a central problem in VLSI system design [1]. A good partitioning tool is required to divide a large system into smaller, more manageable components. As a result, many years of research has been done to develop fast algorithms to find partition solutions of better quality. The development of research in partitioning in the past two decades can be found in a comprehensive survey by Alpert and Kahng [2].

* Corresponding author.
E-mail addresses: wkmak@csee.usf.edu (W.K. Mak), wong@cs.utexas.edu (D.F. Wong).

[1] The work of this author was partially supported by the National Science Foundation under grant CCR-9912390 and by the Texas Advanced Research Program under Grant No. 003658288.

0167-9260/00/$ - see front matter
PII: S0167-9260(00)00008-0

Because of the difference between on-chip and off-chip signal delays, a good partitioning should limit the number of signals travelling off-chip to ensure high system performance. So min-cut partitioning that minimizes the number of interconnections between different chips is desired.

As a net can connect multiple modules, the natural representation for a circuit netlist is a hypergraph where the nodes correspond to the modules and the hyperedges correspond to the interconnections. And the cost of each hyperedge reflects the cost of the corresponding interconnection. There are many algorithms for solving the minimum cut problem for graphs known in the literature [3]. Most of these algorithms rely on maximum flow computations motivated by the famous max-flow min-cut theorem[2] by Ford and Fulkerson [4]. More recently, several researchers [5–7] presented some very efficient algorithms for computing minimum cut without using any flow computation. The fastest algorithms known today are by Nagamochi and Ibaraki [6], and Stoer and Wagner [7]. Their algorithms have a running time of $O(mn + n^2 \log n)$ where m is the number of edges and n is the number of nodes in the graph. Unfortunately, these algorithms are designed for finding minimum cuts in graphs but not in hypergraphs.

Some researchers have tried to find ways to model a hypergraphs by a graph. One model often used is the clique-model where each hyperedge is modeled by a clique [8]. (A clique of k nodes is a complete graph with an edge connecting each pair of the k nodes.) Ideally, the cost of cutting the clique anywhere should be equal to the cost of cutting the corresponding hyperedge. Many different cost assignment methods have been proposed, however, none of them can preserve the cut values of different partitions after the transformation. Indeed, Ihler et al. [9] proved that there is no such perfect transformation to model hypergraphs by graphs with the same min-cut properties.[3] That means any strategy that relies on hypergraph to graph transformation and the use of a minimum cut algorithm for graphs can at best find approximate solutions to an optimal partition of the hypergraph. But because of the importance of the partitioning problem, we have to seek an optimal algorithm for solving the hypergraph minimum cut problem exactly.

The hypergraph minimum cut problem was studied in [10,11]. Hu and Moerder first considered the problem in [10]. They modeled each net x as a star node (Fig. 1) in a node-capacitated flow network where each star node has a capacity equals to the cost of the corresponding net and all other nodes have infinite capacity. They presented a simple flow augmenting-type algorithm to compute the minimum node separator of the network. Since any non-star node has infinite capacity, the minimum node separator cannot contain any non-star node. Thus, a minimum node separator must be a subset of the star nodes which corresponds to the set of nets in a minimum cut of the hypergraph. However, the node-capacitated flow network can be readily transformed into an arc-capacitated network using a technique due to Lawler [12]. Fig. 2(a) shows how the star in Fig. 1 is transformed. After transforming to an arc-capacitated network, more efficient flow algorithms can be applied. Later, Yang and Wong [11] presented another transformation technique that uses less nodes and arcs in the resultant flow network. Fig. 2(b) shows how the star in Fig. 1 is transformed using this new technique.

[2] The max-flow min-cut theorem states the dual relationship between a maximum flow between two nodes s and t and a minimum s–t cut that separates s and t.

[3] To be precise, it was shown in [9] that there cannot be a perfect transformation that preserves the same min-cut properties unless negative edge weights are allowed and dummy nodes are added. However, the problem of computing a minimum cut in a graph with negative weight edges is NP-complete since the maximum cut problem which is a known NP-complete problem can be reduced to it by multiplying all edge weights by -1.

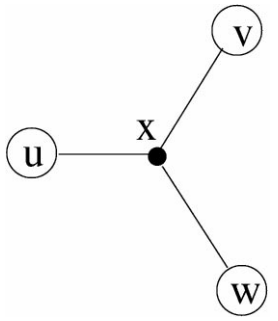


Fig. 1. A net x modeled as a star node connecting all modules in the net.

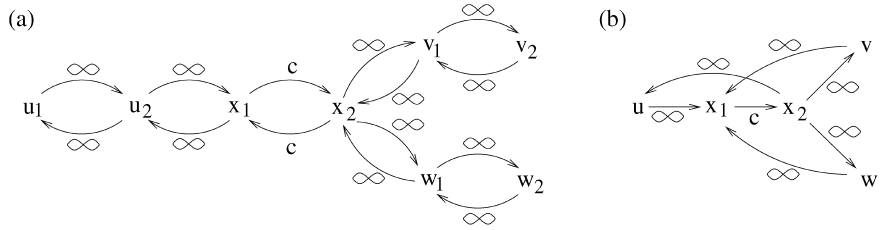


Fig. 2. (a) Transforming a star (c is the cost of star node x). (b) A more compact transformation.

The fastest known algorithms for computing a minimum s–t cut that separates two fixed nodes in a network are flow-based algorithms [3] which takes $\Omega(n'm')$ time where n' and m' are the numbers of nodes and arcs in the network. Using the transformation of [11], the network constructed for computing a hypergraph minimum cut has $n + 2m$ nodes and $m + 2p$ arcs where m, n, and p are the numbers of hyperedges, nodes, and terminals in the hypergraph. So, computing a hypergraph minimum s–t cut using the flow-based approach takes $\Omega((n + 2m)(m + 2p))$ time. We note that a global minimum cut can be found by computing $n - 1$ minimum s–t cuts. (The difference between a global minimum cut and a s–t cut is explained in Section 2.)

However, in this paper, we present a non-flow-based algorithm to compute a global minimum cut in a hypergraph directly. Our algorithm runs in $O(np + n^2 \log n)$ time and it is the fastest hypergraph minimum cut algorithm known today. Note that the number of hyperedges m can be exponential in the number of nodes n in a hypergraph, so computing a global minimum cut using our algorithm is even faster than computing just one minimum s–t cut by the flow-based approach which takes $\Omega(mp)$ time. Our algorithm is a non-trivial extension of the elegant result by Stoer and Wagner [7] which works for graphs only.

The rest of the paper is organized as follows. Section 2 is the preliminaries. Our main algorithm is presented in Section 3. Then in Section 4, we discuss about the implementation of the algorithm and its complexity. The paper is concluded in Section 5.

2. Preliminaries

A *hypergraph* $H = (V, E)$ is defined by its node set V and hyperedge set E. While the edges of a graph connect exactly two nodes each, the hyperedges of a hypergraph can connect two or more

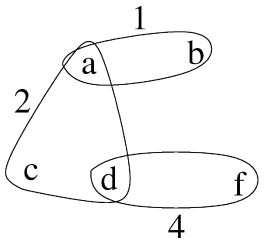


Fig. 3. A weighted hypergraph.

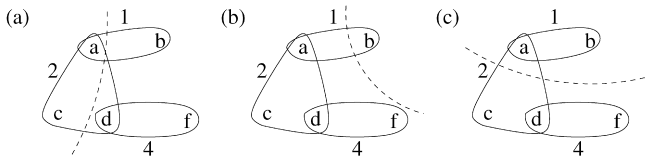


Fig. 4. (a) A cut. (b) A minimum cut. (c) A minimum s–t cut for $s = a$ and $t = c$.

nodes each. Fig. 3 shows a weighted hypergraph where there is a weight or cost associated with each hyperedge. The nodes connected by a hyperedge e are called the *terminals* of e. The hypergraph in Fig. 3 consists of three hyperedges, one of them has nodes a, c, and d as its terminals and has a weight equal to 2.

A *cut* $C = (X, \bar{X})$ is a partition of the set of nodes into two non-empty sets X and $\bar{X}(= V/X)$. A hyperedge that has terminals in both X and $\bar{X}$ is called a *cut hyperedge*. The *size* of a cut C is the total cost of all cut hyperedges, and is denoted by $w(C)$. For example in Fig. 4(a), $C = (\{a, c\}, \{b, d, f\})$ is a cut with $\{a, b\}$ and $\{a, c, d\}$ being the cut hyperedges, and $w(C) = 1 + 2 = 3$.

A *global minimum cut* (or simply *minimum cut*) of a hypergraph H is a cut of H that has the smallest cut size among all the possible cuts of H. A *minimum s–t cut* is a cut among the set of cuts that separate nodes s and t (i.e., $s \in X$ and $t \in \bar{X}$, or vice versa) with the smallest cut size. Referring to Fig. 4(b), cut $(\{b\}, \{a, c, d, f\})$ is a global minimum cut and has a size of 1. Referring to Fig. 4(c), cut $(\{a, b\}, \{c, d, f\})$ is a minimum s–t cut for $s = a$ and $t = c$, and has a size of 2.

3. Hypergraph minimum cut algorithm

The algorithm for computing a minimum cut of a hypergraph is based on the following observation. If we have a procedure P that can compute a minimum s–t cut for some nodes s and t quickly for any hypergraph, then we can compute a global minimum cut quickly by using the procedure $n - 1$ times where n is the number of nodes in the hypergraph.

We can find a global minimum cut for a hypergraph with two nodes by applying procedure P once because a minimum s–t cut is also a global minimum cut for a hypergraph with only two nodes. If we know how to compute a global minimum cut for any hypergraph with $k - 1$ nodes, then we can compute a global minimum cut for a hypergraph H with $k(\geqslant 3)$ nodes as follows. Note

that for any pair of nodes s and t in H, either there exists a global minimum cut with s and t on opposite sides in which case any minimum s–t cut of H is also a global minimum cut of H, or there is no global minimum cut with s and t on opposite sides in which case we can merge s and t into a single node and the problem is reduced to finding a global minimum cut in the reduced hypergraph of $k-1$ nodes. So, a global minimum cut for a hypergraph with k nodes can be computed by finding a minimum s–t cut using procedure P, then computing a global minimum cut of the reduced hypergraph with nodes s and t merged, and finally picking the smaller of the two.

So, the main question is: how can we compute a minimum s–t cut for some nodes s and t quickly for any hypergraph? Note that we are free to choose s and t that are easy for us to compute a minimum s–t cut efficiently. It is shown in [7] how a minimum s–t cut can be computed quickly for a graph where nodes s and t are not given, here we show how it can be done for a hypergraph.

We start with some definitions.

Definition 1. For any node subset A, a node $v \notin A$ is tightly connected to A if there exists a hyperedge that only contains node v and some nodes in A but no node in $V/(A \cup \{v\})$.

Definition 2. For any node subset A and any node $v \notin A$, define $w(A, v)$ as the sum of the weights of all hyperedges that only contain node v and some nodes in A but no node in $V/(A \cup \{v\})$. ($w(A, v)$ is zero if v is not tightly connected to A.)

Definition 3. A node *most tightly connected* to node subset A is a node $v \notin A$ for which $w(A, v)$ is maximum.

For example, for the hypergraph in Fig. 5, node d is tightly connected to set $\{a, b, c\}$ because of hyperedge $\{d, a, b\}$, and node g is tightly connected to set $\{a, b, c\}$ because of hyperedge $\{g, c\}$. But node f is not tightly connected to set $\{a, b, c\}$ (though hyperedge $\{f, c, g\}$ connects f to set $\{a, b, c\}$, it does not make f tightly connected to set $\{a, b, c\}$). And $w(\{a, b, c\}, g) = 1$ (note that hyperedge $\{f, c, g\}$ is not counted in $w(\{a, b, c\}, g)$).

Note that, here, we introduced the notion of tightly connectedness which was not found in [7]. We distinguish between simple connectedness and tightly connectedness. For instance, in Fig. 5, f is connected but not tightly connected to $\{a, b, c\}$. Our notion of tightly connectedness leads to the major difference between our hypergraph minimum cut algorithm and the graph minimum cut

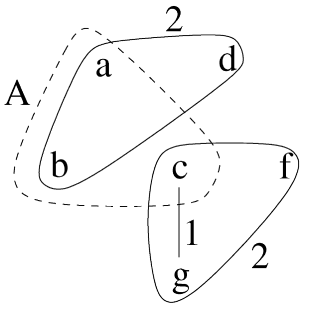


Fig. 5. Both nodes d and g are tightly connected to $A = \{a, b, c\}$ but node f is not.

algorithm in [7]. Defining $w(A, v)$ according to the notion of tightly connectedness is essential to establishing Lemma 1 on hypergraph.

We describe below a fast algorithm for computing a minimum s–t cut in a hypergraph where nodes s and t are not given.

Fast Minimum s–t Cut Algorithm

Input: A hypergraph $H = (V, E)$.
Output: A minimum s–t cut C^* of H and the choice of s and t.

```
x := an arbitrary node in H;
A := {x};
while A ≠ V do
      v := node in V/A most tightly connected to A, i.e., w(A, v) is maximum;
      A := A ∪ {v};
od;    /*A = V*/
s := 2nd last node added to A;
t := last node added to A;
C* := (A/{t}, t);
```

We will prove that the above algorithm correctly finds a minimum s–t cut in the input hypergraph H. We introduce some terminologies useful for the proof. We define a sequence $\{a_i\}$ from the set of nodes in H according to the order they are added to A by the algorithm, i.e., $a_i =$ the ith node added to A. Hence, $s = a_{n-1}$ and $t = a_n$ where n is the number of nodes in H. We use $pred(v)$ to denote the node immediately preceding node v in the sequence. And use A_v to denote the set of nodes in the prefix sequence $(a_1, a_2, a_3, \ldots, pred(v))$.

Definition 4. Given a node sequence $\{a_i\}$ and a cut C, a node v is called an *active* node if v and $pred(v)$ are on opposite sides of the cut C.

Definition 5. Given a node sequence $\{a_i\}$ and a cut C, define C_v to be the set of cut hyperedges that contain only nodes in $A_v \cup \{v\}$ for any $v \in V$.

Consider a particular run of the algorithm and the fixed sequence $\{a_i\}$ associated with it. It is clear that the computed cut $C^* = (V/\{t\}, t)$ is an s–t cut of hypergraph H for $s = a_{n-1}$ and $t = a_n$. To prove that it is a minimum s–t cut for $s = a_{n-1}$ and $t = a_n$, we have to show that $w(V/\{t\}, t) \leqslant w(C)$ for any s–t cut C of H.

Consider an arbitrary s–t cut C of H, where $s = a_{n-1}$ and $t = a_n$, we want to show that $w(V/\{t\}, t) \leqslant w(C)$. Since $A_t = V/\{t\}$ and $C_t = C$ from the definitions, it is equivalent to prove that $w(A_t, t) \leqslant w(C_t)$. However, we will prove a more general result in Lemma 1, which says that $w(A_v, v) \leqslant w(C_v)$ for any active node v. We note that t is an active node because $prev(t) = s$, and by

assumption C is an s–t cut, so s and t are on opposite sides of C. Thus we are done if we can prove the lemma.

Lemma 1. *For any s–t cut C of hypergraph H, where $s = a_{n-1}$ and $t = a_n$, we have $w(A_v, v) \leqslant w(C_v)$ for every active node v in the sequence $\{a_i\}$.*

Proof. We will prove the lemma by induction. Suppose that we have a fixed s–t cut C, where $s = a_{n-1}$ and $t = a_n$.

Base case: If v is the first active node, then all nodes in A_v must be on one side of the cut C, and v must be on the other side. So, any hyperedge that has v as one of its terminals and has all other terminals in A_v must be cut by C, and hence in C_v. Thus we have $w(A_v, v) \leqslant w(C_v)$.

Inductive step: Suppose that the statement holds for active node u, and v is the next active node after u. We will show that the statement also holds for active node v.

By the definition of $w(A_v, v)$, it can be written as the sum of three terms. $w(A_v, v) = w(A_u, v) + w(A_v/A_u, v) + w(A_u, A_v/A_u, v)$ where $w(A_u, A_v/A_u, v)$ is the total cost of all hyperedges with v as a terminal that also have terminals in both A_u and A_v/A_u, but not in $V/(A_v \cup \{v\})$. And since

$$\begin{aligned} w(A_u, v) &\leqslant w(A_u, u) \quad \text{(as } u \text{ is the node most tightly connected to } A_u\text{)} \\ &\leqslant w(C_u) \quad \text{(by the induction hypothesis),} \end{aligned}$$

we have

$$\begin{aligned} w(A_v, v) &= w(A_u, v) + w(A_v/A_u, v) + w(A_u, A_v/A_u, v) \\ &\leqslant w(C_u) + w(A_v/A_u, v) + w(A_u, A_v/A_u, v). \end{aligned}$$

Because all nodes between u and v in the sequence are non-active nodes by the assumption, all nodes in A_v/A_u must be on one side of the cut C, while v must be on the other side as it is active. So any hyperedge with v as a terminal that have at least one terminal in A_v/A_u must be cut by C. Hence, any hyperedge counted in $w(A_v/A_u, v)$ or $w(A_u, A_v/A_u, v)$ must belong to C_v. Also, as C_u is a subset of C_v, any hyperedge counted in $w(C_u)$ must belong to C_v. Finally, notice that the set of hyperedges counted in $w(C_u)$, the set of hyperedges counted in $w(A_v/A_u, v)$, and the set of hyperedges counted in $w(A_u, A_v/A_u, v)$ are mutually disjoint by their definitions. So, we have $w(A_v, v) \leqslant w(C_u) + w(A_v/A_u, v) + w(A_u, A_v/A_u, v) \leqslant w(C_v)$.

From the above, we have $w(A_v, v) \leqslant w(C_v)$ for any active node v with respect to the cut C. As C can be any arbitrary s–t cut of H, the lemma is proved. □

We will illustrate how the Fast Minimum s–t Cut Algorithm works with an example. Consider the hypergraph in Fig. 6. Suppose node a is picked as the arbitrary node to put into A first. Then A becomes $\{a\}$, and the node most tightly connected to A is b since $w(\{a\}, b) = 2$ while $w(\{a\}, e) = 1$ and $w(\{a\}, v) = 0$ for any other node v. So the second node added to A is b and A becomes $\{a, b\}$. Now, there are two nodes most tightly connected to A, namely, nodes c and d, since $w(\{a, b\}, c) = w(\{a, b\}, d) = 3$, $w(\{a, b\}, e) = 1$, and $w(\{a, b\}, f) = 0$. Suppose we choose to add node

c into A, then A becomes $\{a, b, c\}$. It can be checked that d is the node most tightly connected to $\{a, b, c\}$, hence node d is added to A next, followed by node e, and finally node f. So, we get $C^* = (\{a, b, c, d, e\}, \{f\})$, $s = e$ and $t = f$. The cut size of C^* is 3. It can be checked that C^* is indeed a minimum s–t cut for $s = e$ and $t = f$.

The global minimum cut of the hypergraph in Fig. 6 is found by five minimum s–t cut computations. Fig. 7(a) shows the cut computed by the Fast Minimum s–t Cut Algorithm assuming the nodes are added to A in the order a, b, c, d, e, f. The cut size is 3 and the cut separates e and f. So nodes e and f are merged before the second minimum s–t cut computation. Fig. 7(b) shows the cut computed by the Fast Minimum s–t Cut Algorithm assuming the nodes are added to A in the order ef, a, b, d, c. The cut size is 4 and the cut separates d and c. So nodes d and c are merged before the third minimum s–t cut computation. Fig. 7(c) shows the cut computed by the Fast Minimum s–t Cut Algorithm assuming the nodes are added to A in the order cd, b, a, ef. The cut size is 5 and the cut separates a and ef. So nodes a and ef are merged before the fourth minimum s–t cut computation. Fig. 7(d) shows the cut computed by the Fast Minimum s–t Cut Algorithm assuming the nodes are added to A in the order aef, b, cd. The cut size is 7 and the cut separates b and cd. So nodes b and cd are merged before the fifth minimum s–t cut computation. Fig. 7(e) shows the cut. And the cut size is 3. The minimum of the five cuts computed above is a global minimum cut. In this example, the first and the fifth computed cuts are two different global minimum cuts.

The pseudo-code for computing a global minimum cut in a hypergraph is given below.

Fast Minimum Cut Algorithm

Input: A hypergraph $H = (V, E)$.
Output: A global minimum cut C^* of H.

```
H' := H;
cutsize := ∞;
for i = 1 to n − 1 do        /* n = |V| */
    (C, s, t) := minimum s–t cut(H');
    if w(C) < cutsize then
        C* := C;
        cutsize := w(C);
    fi;
    H' := H' with nodes s and t merged;
rof;
return C*;
```

---

4. Implementation and complexity

Let n be the number of nodes in hypergraph H. Let p be the total number of terminals in all the hyperedges of H. The Fast Minimum s–t Cut Algorithm can be implemented to run in $O(p + n \log n)$ time.

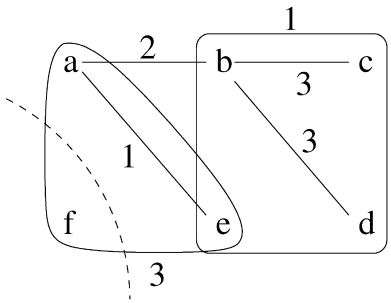


Fig. 6. A weighted hypergraph.

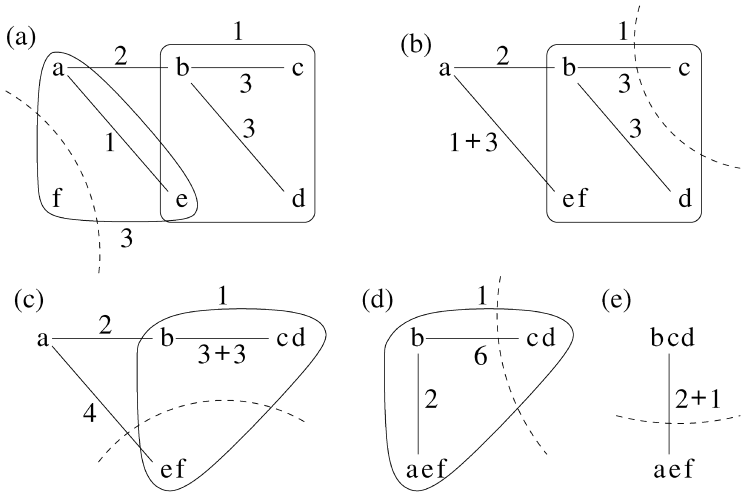


Fig. 7. Computing a global minimum cut.

In the Fast Minimum s–t Cut Algorithm, we need to keep track of the value of $w(A, v)$ for all node v not in the current set A. And we have to find a node v not in the current A with the maximum $w(A, v)$ value quickly. We can put the nodes into a priority queue using the value of $w(A, v)$ as the key of node v. When we add a node v to A, we will remove it from the priority queue, and update $w(A, u)$ for any node u that becomes the last terminal of a hyperedge to remain outside of A (i.e., u and v are terminals of some hyperedge e and they are the only two terminals of e not yet put into A just prior to the addition of v to A). We can identify such last node of a hyperedge by keeping a counter for each hyperedge. The value of the counter of a hyperedge e is set to the number of terminals of the hyperedge initially, and is decremented by 1 each time a terminal of e is added to A. Every time the counter for a hyperedge e becomes 1, we scan all the terminals of e to find the last remaining node u outside A and increase the value of $w(A, u)$ by $w(e)$. During the whole algorithm, the total time spent for decrementing the counters and scanning whenever a counter becomes 1 is $O(p)$. Assume Increase Key(u) is the operation to increase the key value of a node u, and Extract Max is the operation to find and remove the node u with the maximum key value from the priority

queue. There are m Increase Key operations where m is the number of hyperedges in H, and n Extract Max operations. If we implement the priority queue using Fibonacci heaps, the amortized times for performing an Increase Key operation and an Extract Max operation are O(1) and O($\log n$), respectively. Hence, the total running time of the Fast Minimum s–t Cut Algorithm is $O(p + m + n \log n) = O(p + n \log n)$.

We can compute a global minimum cut in a hypergraph in $O(np + n^2 \log n)$ time by applying the Fast Minimum s–t Cut Algorithm $n - 1$ times as in the Fast Minimum Cut Algorithm. For VLSI circuits, usually $p = cn$ for some c around 3 and 4. So the time becomes $O(n^2 \log n)$.

5. Conclusions

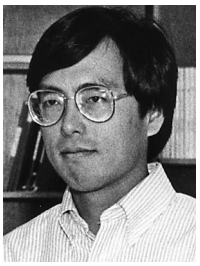
Because of the importance of the hypergraph minimum cut problem in finding good partitions for VLSI circuits, many researchers have studied this problem. Previously, the best reported approach to find a minimum cut in a hypergraph requires transforming the hypergraph into a larger flow network and then applying a flow-based algorithm on the network. However, in this paper we presented a more efficient non-flow-based approach to compute a minimum cut in a hypergraph directly. Our algorithm is the fastest hypergraph minimum cut algorithm known today.

References

[1] B. Preas, M. Lorenzetti, Physical Design Automation of VLSI Systems, Benjamin/Cummings, Menlo Park, CA, 1988.
[2] C.J. Alpert, A.B. Kahng, Recent directions in netlist partitioning: a survey, Integration VLSI J. 19 (1–2) (1995) 1–81.
[3] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[4] J.R. Ford, D.R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, 1962.
[5] D.R. Karger, C. Stein, A new approach to the minimum cut problem, J. ACM 43 (4) (1996) 601–640.
[6] H. Nagamochi, T. Ibaraki, Computing edge-connectivity in multigraphs and capacitated graphs, SIAM J. Discrete Math. 5 (1) (1992) 54–66.
[7] M. Stoer, F. Wagner, A simple min-cut algorithm, J. ACM 44 (4) (1997) 585–591.
[8] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Wiley–Teubner, New York–Stuttgart, 1990.
[9] E. Ihler, D. Wagner, F. Wagner, Modeling hypergraphs by graphs with the same mincut properties, Inform. Process. Lett. 45 (4) (1993) 171–175.
[10] T.C. Hu, K. Moerder, Multiterminal flows in a hypergraph, VLSI Circuit Layout: Theory and Design, IEEE Press, New York, 1985, pp. 87–93.
[11] H. Yang, D.F. Wong, Efficient network flow based min-cut balanced partitioning, Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1994, pp. 50–55.
[12] E.L. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart & Winston, New York, 1976.

Wai-Kei Mak received the B.S. degree in Computer Science from the University of Hong Kong, Hong Kong, China, in 1993. He received the M.S. degree and the Ph.D. degree in Computer Science from the University of Texas at Austin in 1995 and 1998, respectively.

Currently, Dr. Mak is an Assistant Professor in the Computer Science and Engineering Department of the University of South Florida. From 1994 to 1998, he was a research/teaching assistant in the Computer Sciences Department of the University of Texas at Austin. He worked as a CAD engineer at Intel Corporation, Santa Clara, for a summer internship in 1997. His research interests include computer-aided design of VLSI circuits, design automation of multi-FPGA systems, and design and analysis of combinatorial optimization algorithms.



D.F. Wong received the B.Sc. degree in Mathematics from the University of Toronto (Canada) and the M.S. degree in mathematics from the University of Illinois at Urbana-Champaign. He obtained the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign.

Dr. Wong is currently Professor of Computer Sciences at the University of Texas at Austin. His main research interest is CAD of VLSI. He has published over 200 technical papers and has graduated 22 Ph.D. students in this area. He is a coauthor of "Simulated Annealing for VLSI Design" (Kluwer Academic Publishers, 1988) and two invited articles in the Wiley Enclycopedia of Electrical and Electronics Engineering (1999).

Dr. Wong received the 2000 IEEE CAD Transactions Best Paper Award for his work on interconnect optimization. He also received best paper awards at DAC-86 and ICCD-95 for his work on floorplan design and FPGA routing, respectively. Dr. Wong was the General Chair of the 1999 ACM International Symposium on Physical Design (ISPD-99) and was the Technical Program Chair of the same conference in 1998 (ISPD-98). He also has served on the technical program committees of many other VLSI CAD conferences (e.g., ICCAD, ISPD, DATE, ISCAS, FPGA). Dr. Wong has served as an Associate Editor for IEEE Transactions on Computers and Guest Editor of two special issues on physical design for IEEE Transactions on Computer-Aided Design. He is on the Editorial Boards of ACM Transactions on Design Automation of Electronic Systems and International Journal on Applied Mathematics.