Maximum Flow Techniques for Network Clustering

Konstantinos Tsioutsiouliklis

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

June 2002

Abstract

The problem of clustering a data-set according to certain optimization criteria is of great theoretical interest, but also of major practical importance. The classification of large data collections (i.e. web-pages, scientific literature, etc.) requires methods that produce clusters of high quality and are efficient in practice, as well.

This thesis focuses on network clustering, based on maximum flow techniques. Central notion in our methods are various definitions of a community within the network, and key tool for extracting commmunities is the minimum cut tree (or Gomory-Hu tree). We study the properties of the produced clusters and present experimental results for real-world data.

We conclude that the maximum flows of a network provide strong relations between vertices, and allow for clustering algorithms of high quality. From a theoretical point of view we can prove strong performance bounds under several settings. Experimentally, the algorithms are relatively simple to implement and perform well in practice.

Acknowledgments

Later.

to my family - *στην οικογενεια μου*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many clustering algorithms exist for both general data (e.g. directed, undirected graphs) and application-specific data (e.g. pictures, program-modules, etc.). Many of these algorithms are based on optimization criteria such as k-median, minimum sum, minimum diameter, etc. Other algorithms may be more intuitive and are usually justified by empirical results on application specific data-sets. Regardless of the approach, there are two important questions every clustering algorithm must address:

a) What constitutes a good clustering?

b) How good are the clusters produced by the algorithm?

The first question is hard to answer because of its subjectiveness. Different people might consider different clusterings of the same data-set to be better or worse. Different data-sets have different importance criteria, and even the level of locality may yield totally different results. (Here, *locality* refers to how detailed or general the clustering is. The terms *resolution* and *levels* have also been used for the same notion, e.g. in [57]). Figure 1.1 shows an example of points in two dimensions, where at a higher level there are two clusters and at a lower level six.

The second question is easier to answer once the first has been defined, but still

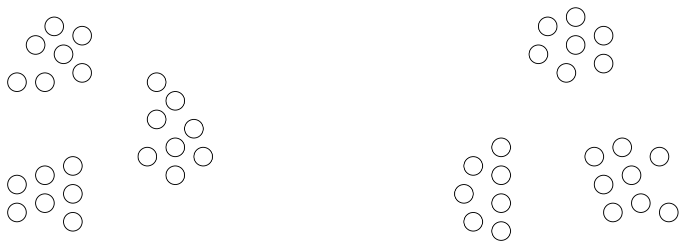

Figure 1.1: Clusters of points on the plane

many of the clustering criteria can be hard to compute. This often results in approximate solutions, heuristics and agglomerate methods, which relax the bounds in the analysis for simpler algorithms and faster running times.

In our research we focus on clustering algorithms based on maximum flows. Maximum flow techniques are well studied, and there is a plethora of algorithms for solving the maximum flow or minimum cut problem. Also many properties of maximum flows have been studied over the years, resulting in ever-faster algorithms and powerful structures that represent certain information about the underlying network. The minimum cut tree [45] is such a structure. It contains sufficient information about the minimum cut between all pairs of nodes in a very compact format.

We use the minimum cut tree, and other properties of maximum flows, to develop a series of clustering algorithms. The objective of these algorithms is to optimize certain clustering criteria, depending on the definition of a community within the network.

Besides studying these algorithms from a theoretical point of view, we have also applied them to a number of data-sets. Many of these data-sets stem from previous experimental studies, others we have developed ourselves, and yet others are real-world data from several sources (e.g. the internet, large databases, etc.). Real-world data is probably the most interesting among the above classes, not only because it is easier to relate to, but also because it shows if and how well the algorithms apply in practice. The fact that real-world data often contains a lot of noise, and thus makes it hard to group into high-quality clusters, together with the large size of the data-sets we used, made the experimental study quite challenging. We report on our experiments at various points throughout the thesis, but have also devoted the final chapter entirely to experimental results.

Structure

This thesis consists of four chapters, besides the Introduction. Chapter 2 contains definitions and previous work. Basic concepts about the maximum flow and minimum cuts are described, as well as a brief review on clustering algorithms.

Chapter 3 is about the minimum cut tree, an important structure for the chapters to follow. The chapter contains two sections, the first of which describes the properties and two older algorithms ([45] and [48]) for calculating a minimum cut tree. The second section contains an experimental study of minimum cut trees we did with A. Goldberg ([43] and [44]).

Chapter 4 contains four clustering methods, based on maximum flows and the minimum cut tree. Section 4.1 describes a basic clustering algorithm. Section 4.2 contains previous work ([78]), that wasn't ours, but fits very well in our general framework, so we decided to make a brief reference to it, as well. Section 4.3 defines and analyzes esoteric communities. Parts of this section were presented in [26], which was in collaboration with G. Flake and S. Lawrence. Section 4.4 describes the cut-clustering algorithm, which was joined work with G. Flake and R. Tarjan.

Finally, Chapter 5 contains experimental studies conducted on two data-sets: the citeseer database [81] and a large subset of the open directory project [82]. Most of this chapter is unpublished, except of on paper ([38]) with E. Glover, S. Lawrence, D. Pennock, and G. Flake, which studies algorithms used for the naming of clusters.

The thesis concludes with an extended bibliography, with special focus on maximum flow techniques and graph clustering algorithms.

Chapter 2

Definitions and Background

2.1 General Notions on Networks and the Maximum Flow

This chapter contains general notation and definitions on networks and the maximum flow problem, as used throughout this thesis.

2.1.1 Basic Definitions on Networks

The clustering algorithms in this thesis apply to networks or graphs. A graph $G(V, E)$ contains $|V| = n$ nodes (or vertices) and $|E| = m$ edges (or arcs) connecting the nodes. (Sometimes we change slightly the definition of a vertex, e.g. in Section 3.2, but always are very explicit when that is the case.)

The set of edges E is defined as $E : V \times V \to S$. If $S = \{0, 1\}$, we say that the graph is unweighted. But usually the edges have real weights ($S = \mathbb{R}^+$), and sometimes they are integers ($S = \mathbb{N}$). We often represent an edge either by naming an element from E (e.g. $e \in E$), or by naming its end-nodes (e.g. $e = (v_1, v_2), v_1, v_2 \in V$). To refer to the weight (or capacity) of edge e we write $w(e)$ or $c(e)$.

In the general case $w(v_1, v_2) \neq w(v_2, v_1)$, and we say that the graph is directed. But when $w(v_1, v_2) = w(v_2, v_1)$, $\forall v_1, v_2 \in V$ the graph is undirected. Most of the clustering algorithms in this thesis apply to undirected graphs.

2.1.2 Maximum Flow and Minimum Cut

Cuts

Let $G(V, E)$ be an undirected network. We define a *cut* to be a partition of V into two nonempty sets. If S_1, S_2 are two nonempty sets, s.t. $S_1 \cup S_2 = V$ and $S_1 \cap S_2 = \emptyset$, then S_1 and S_2 define a cut which we denote as $C(S_1, S_2)$ or simply (S_1, S_2). We make sure that the distinction between a single edge (v_1, v_2) and a cut (S_1, S_2) is always clear. Also, we sometimes omit one of the two sides of a cut, writing e.g. $C(S_1)$. In that case the other side of the cut is implied to be $\overline{S_1} = V - S_1$, and $C(S_1) = C(S_1, \overline{S_1})$.

Every cut has a value, and it is equal to the sum of the capacities of the edges crossing that cut. We say that cut $C(S_1, S_2)$ has value $c(S_1, S_2)$. Let $s, t \in V$ be two nodes of the graph, and let $C(S, T), s \in S, t \in T$ be a cut in G separating s and t. If the cut (S, T) is of minimum value among all cuts separating s and t, we say that (S, T) is a minimum cut between s and t, or equivalently, a minimum s, t-cut.

Flows

A flow in graph G is defined over the edges of G and in terms of a source node s and a sink node t. An s, t-flow f assigns a flow value $f(u, v)$ to every edge $(u, v) \in V \times V$. In order for a flow to be valid it has to satisfy the following constraints:

a) Capacity constraint: $f(u, v) \leq c(u, v)$, $\forall (u, v) \in V \times V$.

b) Antisymmetry constraint: $f(u, v) = -f(v, u)$, $\forall (u, v) \in V \times V$.

c) Conservation constraint: $\sum_{v \in V} f(u,v) = 0$, $\forall v \in V - s, t$.

The value of flow f is $\sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)$. The maximum flow between s and t is an s,t-flow of maximum value. The max-flow min-cut theorem [29] states that the maximum flow between s and t is equal to the minimum cut between s and t.

2.2 The Clustering Problem

Clustering is a very broad and well-studied problem. There are hundreds of clustering algorithms that address the problem under several different setting and optimization goals.

The books by Everitt [23],Andenberg [4], Aldenderfer and Blashfield [3], and Jain and Dubes [57], are good introductory and advanced references. A nice study on four recent clustering techniques was done by Fasulo [24].

The two most common, general approaches to clustering are k-clustering and hierarchical clustering. k-clustering algorithms take as input a set S of objects and an integer parameter k, and output a partition of S into k subsets. Usual optimization criteria for this category include *minimum diameter, k-median, k-center*, and *minimum sum.* See [65] and [46] for important older work, [8] and [52] for surveys, and [13], [14], [21], [56], and [58] for recent works in this area. It is interesting that most of these problems raised are NP-complete, and thus much of the above literature focuses on approximation algorithms. k-clustering algorithms are also very popular in the data mining community ([61], [22], [1]).

Hierarchical clustering is different from k-clustering in that its goal is not to globally partition the data into k sets, but instead to produces a series of clusterings at multiple levels. There are two main types of hierarchical clustering. The first is top-down, or *divisive*, which partitions S recursively into smaller clusters. The recursion bottoms out when it reaches singletons. The second is bottom-up, or *agglomerate*, which starts with singletons and merges smaller clusters into larger ones. Either way, hierarchical clustering algorithms produce a tree $T(S)$, in which nodes correspond to subsets of S. The root of the tree corresponds to the entire set S, and the leaves to singletons. Moving up and down the tree gives larger and smaller clusters, respectively.

Classical algorithms for the hierarchical clustering method are those by Ward [77], Sibson [75], and Defays [18]. Some more recent works are [76], [17], [66].

In this thesis we will study examples of both methods, but most of the algorithms will be k-clustering algorithms. Our work is closely related to other link-based clustering algorithms, e.g. [62], [12], and [35], except that methodologically our approach is not based on spectral graph partitioning or eigenvectors; instead, clusters are formed by minimum cuts in the network.

Minimum cut and maximum flow techniques have also a very rich history, and a minimum cut inherently contains a partition of the graph into heavy connected components. Previous works in that direction are by Wu and Leahy [78], and Hartuv and Shamir [51], but the bounds they achieve are not particularly strong. (We will focus more extensively on Wu and Leahy's work, together with Saran and Vazirani's [74], in Section 4.2, since it is very closely related to our clustering algorithms.) Another work that points out strengths and weaknesses of maximum flows techniques for clustering, is that by Kannan et.al. [59], where in addition strong, general clustering criteria are suggested. We will refer to their work several times in this thesis, esp. in Section 4.4, when studying the quality of the cut clustering algorithm.

Chapter 3

Minimum Cut Trees

3.1 Properties of minimum cut trees and algorithms

In this section we briefly summarize the theory behind minimum cut trees (or Gomory-Hu trees) and present two algorithms ([45] and [48]) for computing a minimum cut tree of a given undirected network.

Proofs of the theorems are omitted, since they can be found in the works of Gomory and Hu ([45]), and Gusfield ([48]). They serve in this thesis as background knowledge for the chapters that follow.

Multiterminal Maximum Flows

The notion of *minimum cut tree* was introduced by Gomory and Hu [45]. Let $G(V, E)$ be an undirected graph, $V = \{v_1, ...v_n\}$ its nodes, and $f(v_i, v_j)$, or simply f_{ij}, the maximum flow between any two nodes $v_i, v_j \in V$.

The following theorem (*realizability*) can be shown about the maximum flow values of a network:

Theorem 3.1.1 *A set of nonnegative numbers $f_{ij} = f_{ji}$, $i, j \in \{1, ..., n\}$, can be valid maximum flow values of a network G if and only if*

$$f_{ik} \geq \min\{f_{ij}, f_{jk}\}, \forall i, j, k \in \{1, ..., n\} \tag{3.1}$$

By induction we get:

Corollary 3.1.2 *For any sequence $v_1, v_2, ..., v_n$ of nodes in a network,*

$$f_{1n} \geq \min\{f_{12}, f_{23}, ..., f_{(n-1)n}\} \tag{3.2}$$

Furthermore, for any undirected network G there exists a network tree T that inherently contains the maximum flow values of that network:

Theorem 3.1.3 *The maximum flow value between any two nodes v_i and v_j of the original network G is equal to $\min\{f_{ia}, f_{ab}, ..., f_{dj}\}$, where $v_{ia}, v_{ab}, ..., v_{dj}$ are the weights of the edges that form the unique path between v_i and v_j in T.*

Tree T is called a *maximum flow equivalent tree.* There may exist multiple different flow equivalent trees for a given network, but Gomory and Hu showed that among those there is always at least one that has the additional property that the removal of the edge of minimum weight on the path between v_i and v_j also yields the minimum cut between v_i and v_j in G, for any v_i, v_j. The two sides of the cut consist of the nodes of the two connected subtrees produced after the removal of the lightest edge. A flow equivalent tree with this additional property is called a *Gomory-Hu tree*, *minimum cut tree*, or simply *cut tree.*

Synopsizing, for every undirected, weighted network $G(V, E)$ there exists a (not necessarily unique) minimum cut tree T over the nodes V that satisfies the following condition:

For every two nodes $v_i, v_j \in V$, let $P(v_i, v_j)$ be the unique path between v_i and v_j in T, and let e_{min} be the edge of $P(v_i, v_j)$ with smallest weight. Then the minimum cut between v_i and v_j in G has value equal to the weight $w(e_{min})$ of e_{min} and the removal of e_{min} from T yields the two sides of the minimum cut.

Gomory-Hu Algorithm

Besides defining a minimum cut tree and proving its existence for every undirected network, Gomory and Hu also provided an algorithm for computing such a minimum cut tree.

The general outline of the algorithm can be described in two steps ([45]):

STEP 1. Perform a maximum flow computation between two nodes on a network, which is usually smaller than the original, since subsets are getting merged into a single node. Go to step 2.

STEP 2. The maximum flow value from Step 1 is used to construct an edge in the minimum cut tree with weight equal to that value. The algorithm ends when $n-1$ edges have been constructed. Now, select a new source and sink for Step 1 in the next iteration, and contract certain subsets of the original network. The new network will be used for the next maximum flow computation. Go to Step 1.

Steps 1 and 2 are only general outlines of the actual algorithm. Figure 3.1 shows the first two iterations of the algorithm. The initial network is shown on the left hand side, and the minimum cut tree to be constructed on the right hand side. In the first iteration of the algorithm, any two nodes are selected and a maximum flow, of value f_1, between them is computed. At that point, the node corresponding to the minimum cut tree gets split into two nodes, which are connected by an edge of weight f_1. Each of these two nodes corresponds to one of the two sides of the minimum cut found in the original network.

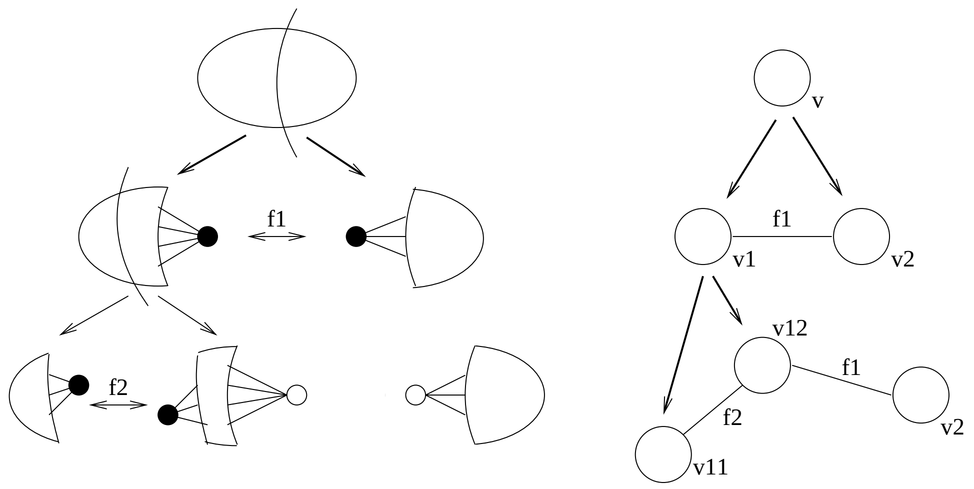

Figure 3.1: Gomory-Hu algorithm

In the next iteration, we work on the network defined by either of the two sides of the minimum cut. This new network consists of the network produced from the original one by contracting the other side of the minimum cut into a single node. The source and sink nodes in this new network must be chosen from the original nodes of the network (not nodes produced by contractions). The maximum flow of the second iteration produces the second edge of the minimum cut tree: the node of the current minimum cut tree that corresponds to the network for which we calculated the maximum flow gets split into two nodes and the edge that connects them gets weight equal to the value of the maximum flow. The other node of the minimum cut tree must now be connected to one of the newly produced nodes. Which one it is, depends on which side of the minimum cut the contracted node ended up: it gets connected to that node which corresponds to the side of the cut which contains the contracted node.

The algorithm continues in this fashion, until every node of the minimum cut tree corresponds to a subgraph of the initial network that contains exactly one original node (and indifferent number of contracted nodes). This will happen after exactly

$n-1$ iterations of the algorithm. Gomory and Hu showed that the minimum cut tree produced is correct.

Gusfield's Algorithm

As a simpler alternative to the Gomory-Hu algorithm, Gusfield ([48]) proposed an algorithm that finds the minimum cut tree of a network without any node contractions. It still requires $n-1$ maximum flow computations, but these are performed each time on the initial network. The minimum cut tree starts off as a star graph, and iteratively its edges get shifted between nodes, according to the maximum flows.

Gusfield suggested the algorithm of Figure 3.2 and proved its correctness.

```
GusCutTree($G(V,E)$)
    Initialize:
        Number all nodes in $V$ from 1 to $n$
        Let $p$ be an $n$ length vector initialized to 1
            /* $p$ corresponds to star tree $T$, s.t. every node $s$ points to $p[s]$ */
        For $s = 2$ to $n$ do
            Compute a minimum cut between nodes $s$ and $t = p[s]$ in $G$
            Let $X$ be the set of nodes on the $s$ side of the cut
            Let $f(s,t)$ be the value of the minimum cut found
            Set $fl[s] = f(s,t)$
            For $i = 1$ to $n$ do
                If $((i \neq s)$ and $(i \in X)$ and $(p[i] == t))$
                    Set $p[i] = s$
            If $(p[t] \in X)$
                Set $p[s] = p[t]$
                Set $p[t] = s$
                Set $fl[s] = fl[t]$
                Set $fl[t] = f(s,t)$
        Output $p$ and $fl$ /* $p$ corresponds to the minimum cut tree $T$
            with edges defined between all nodes $i$ and $p[i]$; the weight of edge
            $(i, p[i])$ is equal to $fl[i]$.
```

Figure 3.2: Gusfield's algorithm for minimum cut trees

The numbering of the nodes between 1 and n is arbitrary, but once these have been set the algorithm is not flexible in picking the next source and sink. Also, it is immediate that structure-wise this algorithm is much simpler than Gomory-Hu's. The main question that follows is: Which algorithm is faster? Asymptotically, they have the same running time of $O(|V| * MAX\_FLOW(G(V,E)))$ for graph $G(V,E)$, where $MAX\_FLOW(G(V,E))$ is equal to the running time for one max-flow computation in G. But how do they compare in practice? The next section, 3.2, addresses this question.

3.2 Experimental study of minimum cut trees

In this section we present an experimental study of algorithms for the cut tree problem. We study the Gomory-Hu and Gusfield's algorithms as well as heuristics aimed to make the former algorithm faster. We develop an efficient implementation of the Gomory-Hu algorithm. We also develop problem families for testing cut tree algorithms. In our tests, the Gomory-Hu algorithm with a right combination of heuristics was significantly more robust than Gusfield's algorithm.

3.2.1 Introduction

Cut trees, introduced by Gomory and Hu [45] and also known as *Gomory-Hu trees*, represent the structure of all s-t cuts of undirected graphs in a compact way. The cut trees have many applications.

All known algorithms for building cut trees use a minimum s-t cut subroutine. The most efficient currently known way to find a minimum s-t cut is using a maximum flow algorithm. See [41] for the currently known maximum flow bounds. Gomory and

Hu [45] showed how to solve the tree problem using $n - 1$[1] minimum cut computations and graph contractions, as we have seen in the previous section. Gusfield [48] proposed an algorithm that does not use graph contraction; all $n-1$ minimum s-t cut computations are performed on the input graph. Gusfield's algorithm is very simple and can be implemented by adding a few lines to a maximum flow code.

Computational performance of algorithms for closely related problems, the maximum flow problem and the (global, e.g. over all s,t pairs) minimum cut problem has been studied extensively; see e.g. [5, 16, 19, 39, 68] for computational studies of the former problem and [15, 63, 64, 67, 69] for the latter. Both problems can be solved well in practice: most problems that fit in RAM of a modern computer can be solved in a few minutes. The cut tree problem appears more difficult, for one needs to solve $n - 1$ minimum s-t cut problems.

Therefore, computational performance of cut tree algorithms is of great interest. Implementations of cut tree algorithms exist – for example, as subroutines of TSP codes [6, 47]. However, we are not aware of any published computational studies of cut tree algorithms. In this section we undertake such a study.

We describe how to implement Gomory-Hu and Gusfield's algorithms efficiently (which is nontrivial for the former). We also introduce and study heuristics aimed at improved computational performance of these algorithms. Our computational experiments lead to a good understanding of practical performance of the cut tree algorithms.

3.2.2 Definitions and Notation

The input to the cut tree problem is an undirected graph $G = (V, E)$ and a capacity function $c : E \Rightarrow \mathbf{R}^+$. We denote $|V| = n$ and $|E| = m$. A *cut* (X, Y) in G is a

[1]We denote the number of vertices and edges in the input graph by n and m, respectively.

partitioning of V into two nonempty sets. We say that an edge *crosses* the cut if its two endpoints are on different sides of the cut. *Capacity of a cut* is the sum of capacities of edges crossing the cut.

We distinguish between *vertices* and *nodes.* We refer to the elements of V as vertices. Nodes correspond to subsets of vertices. (A node can be a single-element subset.) We need the distinction because we use contraction operations.

For $s, t \in V$, an s-t cut is a cut such that s and t are on different sides of it. A *minimum s-t cut* is an s-t cut of minimum capacity. A *(global) minimum cut* is a minimum s-t cut over all s, t pairs.

A *cut tree* is a weighted tree T on V with the following property. For every pair of distinct vertices s and t, let e be a minimum weight edge on the unique path from s to t in T. Deleting e from T separates T into two connected components, X and Y. Then (X, Y) is a minimum s-t cut. Note that T is not a subgraph of G, i.e. edges of T do not need to be in E.

Gomory-Hu Algorithm

In this subsection we outline again briefly the Gomory-Hu algorithm and its efficient implementation. We also discuss heuristics that may improve algorithm's performance in practice. We provide only the details of the algorithm needed to describe the implementation and the heuristics. For a complete description, see e.g. [29, 45, 54].

The Gomory-Hu algorithm is recursive. It distinguishes between two kinds of nodes: original and contracted. A vertex of the input graph is an *original node.* If there is more than one original node, the algorithm picks two, s and t, finds a minimum s-t cut (S, T), and forms two graphs, G_s by contracting S into a contracted node, and G_t by contracting T. Then it recursively builds cut trees in G_s and G_t and puts these trees together. One can see that the algorithm maintains the following invariant: a trivial cut around a contracted node is a minimum cut between this node and some other node in the graph. If there is only one original node, the algorithm has enough information to construct a cut tree; in this case the recursion bottoms out.

Because of the contraction operations, efficient implementation of the Gomory-Hu algorithm is nontrivial. (This was the main motivation behind Gusfield's algorithm.) A naive implementation of contractions allocates new memory for a contracted node and its edges. This may result in $\Omega(n^2)$ memory allocation even for a sparse graph. We describe an implementation that uses $O(\log n)$ extra node records and $O(n)$ extra edge records. These records are allocated as a block at the beginning of the computation, avoiding expensive run-time memory allocation and improving locality of reference.

We maintain the following information at every step of the computation. Recall that the computation is recursive. For each recursive call currently in progress, we maintain information about the cut computed at this level and the graphs obtained by contracting one of the cut sides. When the first recursive call returns, we mark node and edge records of the corresponding subgraph as free. We also maintain information about the contracted nodes (on which side of the cut they are each time), since this determines the structure of the final cut tree. Finally, we maintain a data structure that builds the cut tree according to the cuts found so far.

Our implementation first recurses on the subgraph with a smaller number of nodes and uses fewer additional nodes and edges because of this. We analyze these numbers next.

For the second recursive call, we can reuse the nodes and the edges of the already processed subgraph and use no additional storage. The number of extra nodes we need is determined by the longest sequence of left branches in a root-to-leaf path in the recursion tree (corresponding to the first recursive calls), which is $\lceil \log_2 n \rceil$ because we recurse on the smaller subgraph first. Similarly, the number of extra edges needed is determined by the maximum, over all root-to-leaf paths, of the sum of sizes of subproblems corresponding to left branches. The maximum is bounded by n.

Note that our implementation destroys the input graph. If this is not desirable, one can make a copy of the graph before running the algorithm. All our codes are superlinear, and the time to make the copy would be negligible except for small graphs.

When implemented as described above, direct overhead of contraction operations is small; contraction usually costs less than the corresponding minimum cut computation. However, there is also indirect cost: locality of the input graph representation suffers because of the contractions, reducing the number of cache hits somewhat.

At high level, two major factors determine the computational performance of the algorithm. The first one is the balance (e.g., the ratio of the number of nodes) of the cuts found by the algorithm. In the worst case, one side of every such cut contains one node. In the best case, the cuts are balanced. In the latter case, assuming that minimum cut computations are superlinear, the first one dominates the total running time. The second factor is the hardness of the minimum cut subproblems. Heuristics that lead to more balanced cuts or simpler subproblems improve the algorithm performance.

The *balance* heuristic aims at keeping the cuts balanced. Assume we have at least four original nodes. First we compute all minimum cuts between two such nodes, a and b, and take the most balanced cut. If the cut is sufficiently balanced (e.g. the ratio of the number of nodes of the larger and the smaller parts does not exceed a threshold), we proceed. Otherwise, we pick two nodes, c and d, on the bigger side of the cut. We compute all minimum cuts between c and d, take the most balanced one, compare it to the most balanced minimum cut between a and b, and choose the best. We may have to compute twice as many cuts, so the worst-case loss is about a factor of two. The best-case gain can be much larger.

We can also use both cuts, since according to [45], if the cuts are crossing, we can always find non-crossing cuts. We use this technique, although it usually does not lead to a big speedup: most often one of the cuts is quite unbalanced, and the computation to find non-crossing cuts is relatively expensive.

The *mincut heuristic* makes use of the Hao-Orlin algorithm [49] for finding global mincuts. This algorithm uses the push-relabel method to find a minimum cut between the source and the sink. Then it contracts the source and the sink, and selects a new sink. Hao and Orlin show that with a careful implementation of many push-relabel algorithms, the asymptotic worst-case time bound for these $n-1$ minimum s-t cut computations is the same as that for one minimum s-t cut computation of the underlying algorithm.

Note that the first cut found by the Hao-Orlin algorithm is a minimum s-t cut in the input graph. Also, the algorithm finds a minimum cut, which is a minimum s-t cut for any s, t on the opposite sides of it. We prove a lemma that allows to use several cuts found by the algorithm in the cut tree construction.

The Hao-Orlin algorithm has the following property. Let s be the initial source and let S be the set of vertices contracted into the source at some point of an execution of the algorithm. Let λ be the capacity of the smallest cut found up to this point (initially $\lambda = \infty$). Then for any $x \in S$, the capacity of a minimum s-x cut is at least λ.

Lemma 3.2.1 *Suppose that t is the next sink and the minimum S-t cut has value $\lambda' \leq \lambda$. Then this cut is also a minimum cut between s and t in G.*

Proof. Suppose that there is a smaller cut between s and t. This cut cannot separate

s from a vertex $x \in S$ because s and x are λ-connected. Thus the cut separates S and t. This contradicts the definition of λ'. ■

The mincut heuristic uses the above lemma and finds several minimum s-t cuts with one Hao-Orlin computation. This number is usually small, so we use this heuristic together with the balance heuristic to obtain one or two cuts — the most balanced ones.

The *source selection heuristic* is aimed at both making minimum cut computations simpler and making balanced cuts more likely. This heuristic uses the fact that any original node can be chosen as the source for the next minimum cut computation. After choosing a sink for the computation, we choose an original node that is furthest away from the sink as the source. (All distances are with respect to a unit length function.) Note that we use an implementation of the push-relabel method [42] based on that of [16]. This implementation computes distances to the sink during the initialization, so the source selection heuristic adds essentially no overhead.

As part of the source selection heuristic, we choose the source/sink to be the heaviest nodes of the graph (e.g. nodes with the highest total capacity of adjacent edges), since this sometimes leads to more balanced cuts.

Our Implementations

After studying different ways of incorporating heuristics into the Gomory-Hu algorithm, we report on three implementations. The GH code uses no heuristics and picks the next source/sink pair at random. The GHs code uses the source selection heuristic. The GHg code uses the mincut heuristic in the following way: Initially, it picks the two heaviest nodes as the source and the sink of the Hao-Orlin algorithm.[2] As soon as it finds a cut in the decreasing sequence which is more balanced than the

[2]A random choice was much less robust in our tests.

first cut found, it splits the graph according to both this cut and the first one. Our experience shows that using the mincut heuristic is the best way to find balanced cuts at low expense.

Gusfield's Algorithm

Like the Gomory-Hu algorithm, Gusfield's algorithm [48] consists of $n - 1$ iterations of a minimum cut subroutine and bookkeeping that puts the resulting cuts together. Gusfield's algorithm, however, does not contract vertices and works with the original graph, making it easy to implement. At each of the $n - 1$ iterations of Gusfield's algorithm, a different vertex is chosen as the source. This choice determines the sink.

Low-level operations of this algorithm are efficient because of its simplicity and the fact that the algorithm takes advantage of locality of the input graph representation. However, all minimum cut subproblems are as big as the original graph. Furthermore, the algorithm has less flexibility for adding heuristics. The only flexibility is the choice of the next source. We choose the next source at random. We refer to the resulting implementation as GUS.

Experimental Setup

For our computations, we used a SUN Sparc Ultra-2 workstation with 256MB memory running SunOS 5.5.1. All the code is written in C and compiled with 'gcc' and optimization option -O4. Our implementations are written in the same style and are derived from the Hao-Orlin algorithm implementation of [15]. We attempted to make all implementations as efficient as possible.

For our tests we use problem families from the previous minimum cut studies [15, 64, 67, 69], but instead of finding a minimum cut of a graph, we build a cut tree. We omit the description of the problem families. Detailed descriptions appear in [64].

Problem family	Generator	# nodes	# edges	Other parameters
BIKEWHE	bikewheelgen	32,64,...,1024	$2n-3$	
CYC1	cyclegen	64,...,4096	n	
DBLCYC	dblcyclegen	64,...,1024	$2n$	
IRREG	irregulargen	1000	4000-5000	$k=8,9,W\in[0\dots1000]$
NOI1	noigen	100-800	density: 50%	$P=n,k=1$
NOI2	noigen	100-800	density: 50%	$P=n,k=2$
NOI3	noigen	500	6000-124000	$P=1000,k=1$
NOI4	noigen	500	6000-124000	$P=1000,k=2$
NOI5	noigen	500	62000	$P=1000$ $k=1,3,...,100,500$
NOI6	noigen	500	62000	$P=5000,2000,...,10,1$ $k=2$
PATH	pathgen	2000	20000	$P=1,000$ $k\in[1\dots2000]$
PR1	prgen	200,400,...,1000	density: 2%	$k=1$
PR5	prgen	200,400,...,1000	density: 2%	$k=2$
PR6	prgen	200,400,...,1000	density: 10%	$k=2$
PR7	prgen	200,400,...,600	density: 50%	$k=2$
PR8	prgen	200,400,...,600	density: 100%	$k=2$
REG1	regulargen	301	301,...,90300	
REG2	regulargen	50,100,...,800	$50n$	
TREE	treegen	800	density: 50%	$k\in[1\dots800]$
TSP	tsp-instances	500-13000	$\approx n$	
WHE	wheelgen	64,128,...,1024	$2n-2$	

Table 3.1: Problem families reported on in the experimental study of minimum cut trees.

We do not report on PR2–PR4 problem families because the results are very close to those for the PR1 family, and on REG3–REG4 families because the results are very close to those for the REG1 and REG2 families. We also use two new problem families produced by two generators, PATHGEN and TREEGEN, described below. A summary of the problem families we use appears in Table 3.1. We experimented with more families, but do not report on some where the results were similar to the ones we include.

The PATHGEN generator works as follows. Given a parameter k, it builds a path of $k-1$ "heavy" edges and connects the remaining $n-k$ vertices to the path vertices by heavy edges, at random. Then it adds "light" edges at random to achieve the desired number of arcs and to make the minimum cut problems more difficult. This generator takes the following parameters:

- n, the number of vertices;
- d, the density of the graph as a percentage;
- k, the path length;
- P, the path arc capacity parameter;
- S, the seed.

Heavy edge capacities are chosen uniformly at random from the interval $[1, \ldots, 100 \cdot P]$ and light edge capacities from $[1, \ldots, 100]$.

The value of k determines the path shape. For example, if $k = n$ then we get one heavy path through all the nodes; if $k = 1$, then the graph is a star. We use PATHGEN to produce the PATH problem family. We use $n = 2,000$, $d = 10$, $P = 1,000$, and k changing from 1 to $2,000$.

The TREEGEN generator works as follows. Given a parameter k, it builds a tree by connecting vertex i, $2 \leq i \leq n$, to a randomly chosen vertex in $[1, \min(i-1, k)]$. The tree edges are heavy. Then it adds "light" edges at random to achieve the desired number of arcs and to make the minimum cut problems more difficult. This generator takes the following parameters:

- n, the number of vertices;
- d, the density of the graph as a percentage;
- k, the shape parameter mentioned above;
- P, the path arc capacity parameter;

	GUS	GH	GHS	GHG
BIKEWHE	$\bigcirc$	$\bigcirc$	$\odot$	$\bigcirc+$
CYC1	$\bigcirc+$	$\bigcirc$	$\bigcirc$	$\bigcirc$
DBLCYC	$\bullet$	$\otimes$	$\otimes$	$\bigcirc+$
IRREG1	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\bigcirc$
NOI1	$\bigcirc+$	$\bigcirc$	$\bigcirc$	$\bigcirc$
NOI2	$\odot$	$\bigcirc+$	$\bigcirc$	$\bigcirc$
NOI3	$\bigcirc+$	$\bigcirc$	$\bigcirc$	$\bigcirc$
NOI4	$\odot$	$\bigcirc$	$\bigcirc$	$\bigcirc$
NOI5	$\bullet$	$\odot$	$\bigcirc+$	$\bigcirc$
NOI6	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\bigcirc$
PATH	$\otimes$	$\bullet$	$\bigcirc$	$\bigcirc$
PR1	$\bigcirc+$	$\bigcirc$	$\bigcirc$	$\bigcirc$
PR5	$\bigcirc$	$\bigcirc$	$\bigcirc+$	$\bigcirc$
PR6	$\odot$	$\bigcirc$	$\bigcirc$	$\bigcirc$
PR7	$\odot$	$\bigcirc+$	$\bigcirc$	$\bigcirc$
PR8	$\odot$	$\bigcirc+$	$\bigcirc$	$\bigcirc$
REG1	$\bigcirc+$	$\bigcirc$	$\bigcirc$	$\bigcirc$
REG2	$\bigcirc$	$\bigcirc$	$\bigcirc$	$\bigcirc$
TREE	$\otimes$	$\bullet$	$\bigcirc$	$\odot$
TSP	$\otimes$	$\bigcirc$	$\bigcirc$	$\bigcirc$
WHE	$\odot$	$\odot$	$\odot$	$\bigcirc$

Table 3.2: Summary of algorithm performance. $\bigcirc$ means good, $\odot$ means fair, $\otimes$ means poor, and $\bullet$ means bad. + marks the fastest code(s).

- S, the seed.

The generator chooses heavy edge capacities uniformly at random from the interval $[1, \ldots, 100 \cdot P]$ and light edge capacities from $[1, \ldots, 100]$.

The value of k determines the shape of the tree. For example, if $k = 1$ then the tree is a star. If $k = n - 1$, then a tree is obtained by connecting each vertex except the first one to a randomly chosen preceding vertex.

We use TREEGEN to produce the TREE problem family.

Experimental Results

In this section we describe our experimental results. Table 3.2 summarizes these results. Detailed data appears at the end of this section. As usual, our experimental results should be taken in the context of our study.

We use the following scoring system in the table. We normalize the times by that of the fastest code and use a factor of two as the threshold between adjacent scores. For example, if the fastest code runs in x seconds, a code running in $1.5x$ seconds is rated good, in $3x$ seconds – fair, in $7x$ seconds – poor, in $12x$ – bad. Our choice of the threshold makes it less likely that a code not rated good in our experiment would be the fastest under a different compiler and machine architecture combination. The scoring is done using instances with the biggest performance difference (usually, the largest instances) for a problem family. If a code is consistently faster than the other codes, we mark that code with a +. Several codes can be marked so if their performance is very close, and no codes can be marked if there is no consistent winner. Note that no code will get a good score on a problem family if every code performs relatively poorly for some parameter values.

This scoring system gives a general idea of relative performance of the codes and is robust with respect to many low-level implementation details and machine architecture variations. Note that in some cases larger problem sizes may amplify performance differences and thus change the scores.

Data tables at the end of this section give much more information than the above scores and can be used to explain performance differences. All our implementations are based on the push-relabel maximum flow method; we give counts of the push and relabel operations which give a machine-independent measure of performance. We also give the average size (the number of nodes and the number of edges) of the s-t cut problems solved. The average problem size is correlated with the algorithm performance. In addition to the total running time, we give the time spend computing minimum s-t cuts (CutTime) and the time spend on auxilary operations (ManipTime) such as building the cut tree, contracting nodes, etc. The total time is equal to the sum of the CutTime, ManipTime, initialization time, and postprocessing time.

3.2.3 Gusfield's Algorithm

The data shows that Gus is not robust. Although it is the fastest code on many problem families, in some cases it performs much worse than the Gomory-Hu algorithms.

Operation counts show that Gusfield's algorithm wins mostly due to its simplicity and better spatial locality resulting from the lack of contraction operations. The algorithm works on the original graph, so the average subproblem size is the original graph size. In contrast, Gomory-Hu algorithm wins when it gets balanced cuts which reduce the average size of the subproblems as well as reduces the number of push and relabel operations (which dominate the computation).

Note that if one assumes that contraction operations do not increase the number of push and relabel operations needed to solve a minimum s-t cut problem, the only reason Gusfield's algorithm may be faster than the Gomory-Hu algorithm is because of better locality and the lack of contraction operations. Since the work of the latter can be amortized, Gusfield's algorithm cannot win by more than a moderate constant factor. The Gomory-Hu algorithm can, and in some cases does, win by a wide margin.

3.2.4 The Gomory-Hu Implementations

Next we compare our implementations of the Gomory-Hu algorithm. Performance of these implementations depends on two factors: how balanced the "typical" cuts are, and how much work is involved in looking for more balanced cuts.

Recall that the GH implementation choses the next source-sink pair at random. This is a natural selection strategy to try. Somewhat surprisingly, in our tests this strategy was not as robust as the source selection heuristic used in GHs. On most problem families, GH performs similarly to GHs, but on a small number of families (in particular PATH and TREE), the former code is noticeably slower.

GHs and GHg are the most robust codes in our study, with the latter code being somewhat more robust. Receiving only one fair mark, GHg is the most robust code in our study. On some input classes (BIKEWHE, DBLCYC, WHE), it outperforms the other codes by a large margin. This is due to the fact that on these problem classes, GHg finds more balanced cuts and on the average works with smaller problems. One has to keep in mind, however, that the structure of these graph instances is quite special. They are very symmetric and have many cuts of the same value, which GHg takes advantage of. So, one has to be careful not to overestimate GHg performance in the general case. In general, the average problem size for GHg tends to be somewhat smaller than for GHs. However, the size is never much smaller, and often does not pay for the additional overhead. Thus on many graphs GHg is slightly slower than GHs.

The TREE family is the only problem family where GHg gets a fair score. As usual, on this family the average problem size for GHg is smaller than for GHs, but the minimum cut problems are very easy and GHs solves them very fast. GHg, finding several cuts for each problem, spent significantly more time on each of those problems.

Concluding Remarks

In this section we summarize our work and discuss heuristics that work as well as those that do not work.

Currently, the cut tree problems are substantially harder than the related maximum flow and minimum cut problems, both in theory and in practice. This is a good motivation for improving theoretical bounds for the problem and developing faster codes for it. Our study is a step towards the faster codes.

We get a good understanding of implementation issues for the existing cut tree algorithms, as well as a good understanding of computational performance of these algorithms. In particular, we show that with a careful low-level implementation, the Gomory-Hu algorithm is more robust than Gusfield's algorithm. This is because all subproblems solved by Gusfield's algorithm have the same size as the input problem. For the Gomory-Hu algorithm, however, the average problem size can be much smaller than the original problem size. The Gomory-Hu algorithm performance is less predictable, because the average problem size depends on the heuristics used.

Good heuristics reduce the average problem size and can substantially improve performance of the Gomory-Hu algorithm on some problems. One such heuristic is our source selection heuristic, which was the most robust in our tests. Random selection, although quite natural and easy to implement, does not work as well.

Other heuristics are based on the idea of finding several minimum cuts at every iteration of the Gomory-Hu algorithm and selecting the most balanced one. We experimented with the simple balance heuristic of selecting the best of two cuts at every recursive call of the algorithm. The resulting implementation was usually slower than GH, although not by much, and never significantly faster. This is because the best of the two cuts is usually not much more balanced than the first cut. The Hao-Orlin algorithm provides more opportunities for finding balanced cuts, and the GHG implementation was the most robust implementation in our tests. Further research may lead to even more effective heuristics, but we could not produce a more robust code.

Padberg-Rinaldi heuristics [69] proved very useful for certain classes of global minimum cut problems. One can use these heuristics (in a somewhat restricted form) to speed up s-t cut computations in the Gomory-Hu algorithm. However, on the problems these heuristics are effective, their use tends to lead to less balanced cuts and worse running times. This is because the heuristics tend to contract together large subsets of nodes. Although we invested substantial effort, we could not use the heuristics to consistently speed up our codes.

Further study of heuristics for the Gomory-Hu algorithm may provide significant improvements.

Data Tables and Plots

In the following tables and plots we present data for all the class instances we have considered in this study.

Notes:

- All the data reported has been averaged over multiple (5) runs of the algorithms for each input instance.
- N and M denote the number of vertices and edges respectively.
- $Aver.N$ and $Aver.M$ are equal to the average size (vertices and edges, respectively) over all subgraphs during the min-cut computations. For GUS these values are equal to N and M. For GH, GHS and GHG they are often significantly smaller.

- $CutTime$ corresponds to the total running time required to find all the cuts; this time is dominated by the max-flow computations.

- $ManipTime$ is the time required to manipulate the cuts. This includes the time to build the tree and for GH, GHs and GHg also includes the time for the contractions done. Moreover, for GHg it includes the time to perform double-splitting, according to the two most balanced cuts found so far.

- $Relabels$ and $Pushes$ account for the total number of relabels and pushes during the min-cut computation.

- $TotalTime$ is the total CPU-time for each algorithm. $TotalTime$ should be slightly bigger than $CutTime + ManipTime$, ($TotalTime$ also includes initialization and final output).

- Some tables contain individual parameters for certain problem families.

BIKEWHE									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	32	61	32.000	61.000	0.018	0.002	2590	4632	0.020
gh	32	61	32.000	79.000	0.016	0.000	2173	3704	0.016
ghs	32	61	32.000	77.000	0.020	0.004	3297	6533	0.024
ghg	32	61	14.097	36.113	0.006	0.002	1412	2526	0.008
gus	64	125	64.000	125.000	0.078	0.014	18428	39598	0.096
gh	64	125	64.000	159.000	0.076	0.016	16846	34441	0.098
ghs	64	125	64.000	157.000	0.122	0.018	23681	53183	0.140
ghg	64	125	27.603	74.738	0.054	0.018	10374	20601	0.074
gus	128	253	128.000	253.000	0.612	0.054	116305	276111	0.670
gh	128	253	128.000	319.000	0.588	0.060	117001	268674	0.652
ghs	128	253	128.000	317.000	0.818	0.076	160638	380200	0.902
ghg	128	253	52.504	144.008	0.322	0.050	60367	125806	0.372
gus	256	509	256.000	509.000	4.054	0.292	736057	1881693	4.370
gh	256	509	256.000	639.000	4.092	0.312	771031	1909737	4.420
ghs	256	509	256.000	637.000	6.048	0.336	1107079	2749403	6.396
ghg	256	509	98.929	275.508	2.102	0.214	357203	756738	2.332
gus	512	1021	512.000	1021.000	28.302	1.188	4597115	12473634	29.540
gh	512	1021	512.000	1279.000	27.458	1.276	4778584	12344741	28.754
ghs	512	1021	512.000	1277.000	44.138	1.464	7713890	19621858	45.624
ghg	512	1021	198.252	556.555	14.128	0.858	2220836	4848881	15.014
gus	1024	2045	1024.000	2045.000	193.438	6.364	30058422	83973336	199.882
gh	1024	2045	1024.000	2559.000	202.008	8.066	30431074	81759094	210.142
ghs	1024	2045	1024.000	2557.000	355.830	8.130	54754257	142345058	364.014
ghg	1024	2045	396.855	1113.205	107.632	5.492	15305374	33838972	113.172

Table 3.3: Data for BIKEWHE family

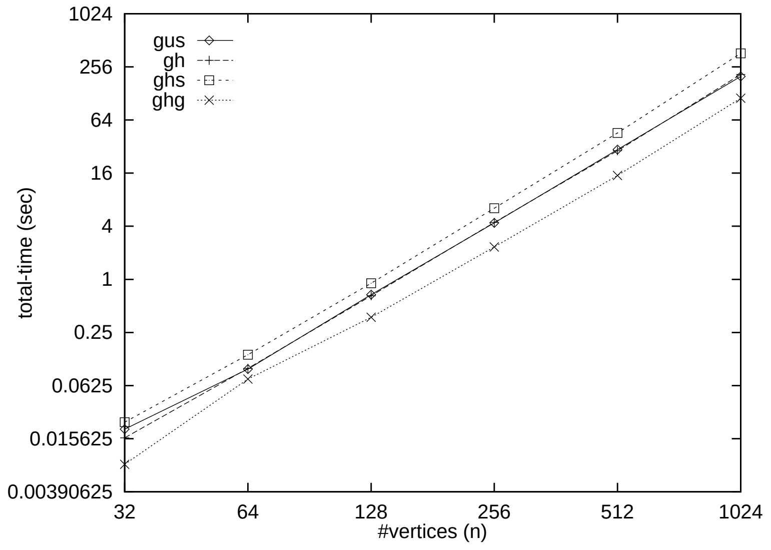

Figure 3.3: Running times for BIKEWHE family

CYC1									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	64	64	64.000	64.000	0.016	0.016	3782	3906	0.034
gh	64	64	64.000	98.000	0.010	0.030	3781	3906	0.044
ghs	64	64	64.000	96.000	0.010	0.004	3782	3906	0.026
ghg	64	64	64.000	96.000	0.030	0.012	3782	3906	0.052
gus	128	128	128.000	128.000	0.056	0.046	15750	16002	0.110
gh	128	128	128.000	194.000	0.036	0.060	15749	16002	0.104
ghs	128	128	128.000	192.000	0.058	0.062	15750	16002	0.122
ghg	128	128	128.000	192.000	0.080	0.060	15750	16002	0.144
gus	256	256	256.000	256.000	0.212	0.196	64262	64770	0.428
gh	256	256	256.000	386.000	0.224	0.230	64263	64770	0.460
ghs	256	256	256.000	384.000	0.218	0.230	64262	64770	0.466
ghg	256	256	256.000	384.000	0.362	0.228	64262	64770	0.610
gus	512	512	512.000	512.000	0.818	0.746	259590	260610	1.600
gh	512	512	512.000	770.000	0.808	0.986	259588	260610	1.822
ghs	512	512	512.000	768.000	0.902	0.980	259590	260610	1.902
ghg	512	512	512.000	768.000	1.620	0.852	259590	260610	2.502
gus	1024	1024	1024.000	1024.000	3.252	4.410	1043462	1045506	7.724
gh	1024	1024	1024.000	1538.000	3.064	6.592	1043462	1045506	9.698
ghs	1024	1024	1024.000	1536.000	3.724	5.678	1043462	1045506	9.456
ghg	1024	1024	1024.000	1536.000	5.788	6.174	1043462	1045506	12.006
gus	2048	2048	2048.000	2048.000	12.190	29.252	4184070	4188162	41.606
gh	2048	2048	2048.000	3074.000	14.530	35.908	4184071	4188162	50.528
ghs	2048	2048	2048.000	3072.000	14.824	34.104	4184070	4188162	49.028
ghg	2048	2048	2048.000	3072.000	26.068	35.952	4184070	4188162	62.148
gus	4096	4096	4096.000	4096.000	50.988	116.528	16756742	16764930	167.846
gh	4096	4096	4096.000	6146.000	66.034	154.422	16756740	16764930	220.680
ghs	4096	4096	4096.000	6144.000	63.144	142.724	16756742	16764930	206.070
ghg	4096	4096	4096.000	6144.000	106.782	152.694	16756742	16764930	259.732

Table 3.4: Data for CYC1 family

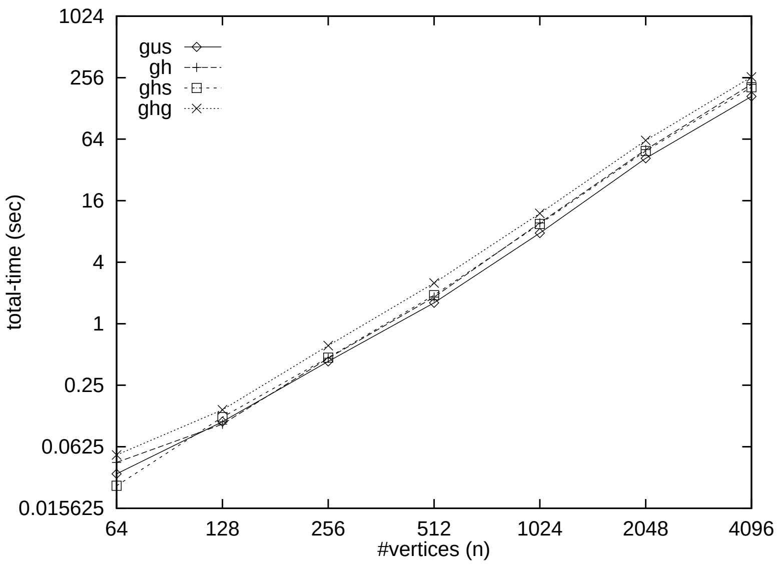

Figure 3.4: Running times for CYC1 family

DBLCYC									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	64	128	64.000	128.000	0.034	0.020	8904	13986	0.056
gh	64	128	64.000	162.000	0.038	0.018	9940	17248	0.058
ghs	64	128	64.000	160.000	0.086	0.016	19359	37934	0.108
ghg	64	128	22.571	55.540	0.026	0.006	4593	7450	0.040
gus	128	256	128.000	256.000	0.700	0.058	158022	302252	0.760
gh	128	256	65.118	164.843	0.134	0.048	36321	57223	0.190
ghs	128	256	46.874	116.925	0.130	0.052	30128	53221	0.184
ghg	128	256	15.913	38.413	0.040	0.056	5584	9436	0.104
gus	256	512	256.000	512.000	3.758	0.282	824857	1586501	4.050
gh	256	512	128.843	324.120	0.714	0.202	186897	308900	0.936
ghs	256	512	93.463	233.465	0.864	0.170	211112	381336	1.048
ghg	256	512	24.263	59.461	0.128	0.160	21506	35338	0.298
gus	512	1024	512.000	1024.000	44.272	1.282	9343902	18489463	45.600
gh	512	1024	257.411	645.535	4.494	0.898	1123988	1978619	5.404
ghs	512	1024	226.335	565.772	7.568	0.920	1787452	3390017	8.510
ghg	512	1024	43.129	106.592	0.478	0.620	85484	135.154	1.124
gus	1024	2048	1024.000	2048.000	180.620	6.206	36782759	72438517	186.904
gh	1024	2048	513.879	1286.702	34.576	5.368	7506538	13993665	39.990
ghs	1024	2048	399.940	999.699	54.382	5.216	11568074	22416182	59.648
ghg	1024	2048	91.745	228.161	2.174	3.906	407532	656551	6.120

Table 3.5: Data for DBLCYC family

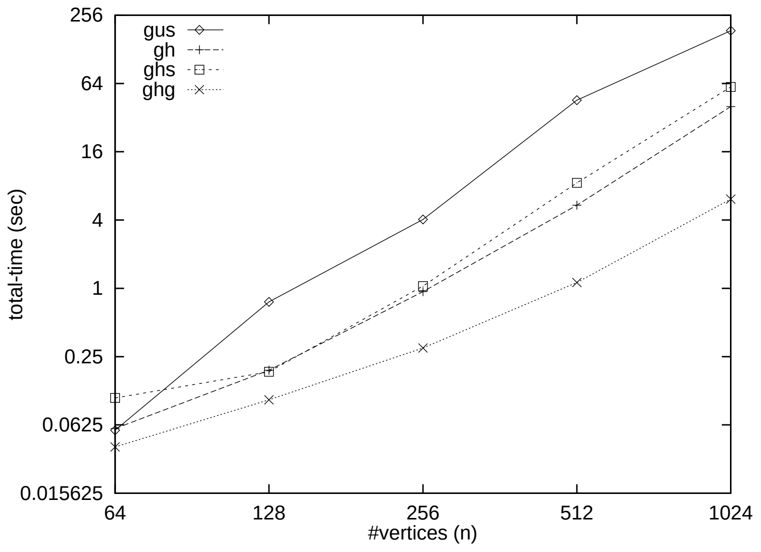

Figure 3.5: Running times for DBLCYC family

IRREG1											
	N	M	K	W	Aver.N	Aver.M	CTime	MTime	Relabels	Pushes	TotTime
gus	1000	4000	8	0	1000.00	4000.00	5.39	7.32	974166.2	1023545.0	12.792
gh	1000	4000	8	0	1000.00	4487.40	5.75	9.97	974336.6	1025379.0	15.778
ghs	1000	4000	8	0	1000.00	4485.40	6.06	9.42	964990.2	1012366.6	15.530
ghg	1000	4000	8	0	281.56	1777.62	4.31	5.38	507450.4	595453.4	9.746
gus	1000	4002	8	4	1000.00	4002.00	4.77	7.30	859142.2	892370.4	12.144
gh	1000	4002	8	4	1000.00	4489.40	5.78	9.96	978548.0	1031098.4	15.806
ghs	1000	4002	8	4	1000.00	4487.40	5.58	9.40	859329.0	890894.8	15.036
ghg	1000	4002	8	4	283.21	1774.54	4.30	5.32	513306.0	603358.2	9.682
gus	1000	4032	8	64	1000.00	4032.00	4.77	7.35	849120.2	879463.4	12.208
gh	1000	4032	8	64	1000.00	4519.20	5.91	10.04	1005170.4	1059059.4	16.024
ghs	1000	4032	8	64	1000.00	4517.20	5.68	9.54	860271.2	891862.2	15.296
ghg	1000	4032	8	64	308.11	1846.52	4.65	5.49	594861.4	693941.2	10.188
gus	1000	4128	8	256	1000.00	4128.00	5.38	7.52	965819.0	1012517.2	12.960
gh	1000	4128	8	256	1000.00	4614.60	6.18	10.22	1034272.2	1090457.6	16.460
ghs	1000	4128	8	256	1000.00	4612.60	6.22	9.95	924975.8	965052.4	16.230
ghg	1000	4128	8	256	351.77	2067.06	5.53	5.96	707328.8	824033.0	11.542
gus	1000	4384	8	768	1000.00	4384.00	5.59	7.74	970471.6	1017946.6	13.432
gh	1000	4384	8	768	1000.00	4868.60	6.31	10.78	997111.6	1049918.8	17.140
ghs	1000	4384	8	768	1000.00	4866.60	6.95	10.63	974138.8	1022270.4	17.642
ghg	1000	4384	8	768	306.32	1969.54	5.22	5.90	587228.6	690631.4	11.156
gus	1000	4500	8	1000	1000.00	4500.00	5.60	7.84	943306.0	989277.2	13.522
gh	1000	4500	8	1000	1000.00	4984.20	6.27	11.08	970817.6	1021374.2	17.408
ghs	1000	4500	8	1000	1000.00	4982.20	6.65	10.42	959151.8	1005409.8	17.126
ghg	1000	4500	8	1000	282.99	1987.57	5.08	5.86	521775.2	629682.6	10.990
gus	1000	4500	9	0	1000.00	4500.00	5.54	7.86	939854.8	983888.2	13.478
gh	1000	4500	9	0	1000.00	4984.40	6.31	10.99	971813.8	1022280.8	17.334
ghs	1000	4500	9	0	1000.00	4982.40	6.70	10.41	963312.2	1010213.4	17.160
ghg	1000	4500	9	0	290.21	2037.71	5.18	5.97	533194.8	637646.8	11.210
gus	1000	4502	9	4	1000.00	4502.00	5.09	7.85	861165.2	891509.0	13.024
gh	1000	4502	9	4	1000.00	4986.40	6.31	11.01	973293.2	1022636.8	17.378
ghs	1000	4502	9	4	1000.00	4984.40	6.33	10.45	891036.4	927130.4	16.816
ghg	1000	4502	9	4	286.36	1985.68	5.06	5.87	525460.0	623485.0	10.986
gus	1000	4532	9	64	1000.00	4532.00	5.38	7.81	907690.8	946118.2	13.248
gh	1000	4532	9	64	1000.00	5016.20	6.58	11.12	1007777.2	1061260.2	17.742
ghs	1000	4532	9	64	1000.00	5014.20	6.45	10.58	896267.4	932213.6	17.096
ghg	1000	4532	9	64	320.02	2116.92	5.62	6.15	622135.4	730064.6	11.840
gus	1000	4628	9	256	1000.00	4628.00	5.35	7.95	888729.4	924258.4	13.366
gh	1000	4628	9	256	1000.00	5111.80	6.83	11.34	1027583.6	1082463.2	18.222
ghs	1000	4628	9	256	1000.00	5109.80	6.89	11.19	895798.8	932278.8	18.146
ghg	1000	4628	9	256	327.07	2103.72	5.74	6.30	651540.4	764990.2	12.082
gus	1000	4884	9	768	1000.00	4884.00	5.71	8.16	906560.0	944295.6	13.928
gh	1000	4884	9	768	1000.00	5366.20	6.91	12.10	998317.4	1051006.4	19.060
ghs	1000	4884	9	768	1000.00	5364.20	7.50	11.87	902697.2	940658.8	19.418
ghg	1000	4884	9	768	314.30	2258.70	6.09	6.52	600495.2	703592.4	12.672
gus	1000	5000	9	1000	1000.00	5000.00	6.04	8.38	954539.8	998726.4	14.506
gh	1000	5000	9	1000	1000.00	5481.40	6.92	12.42	966768.8	1015155.8	19.392
ghs	1000	5000	9	1000	1000.00	5479.40	7.60	11.82	937960.2	979618.8	19.470
ghg	1000	5000	9	1000	293.30	2252.57	5.86	6.48	544862.8	660723.6	12.390

Table 3.6: Data for IRREG1 family

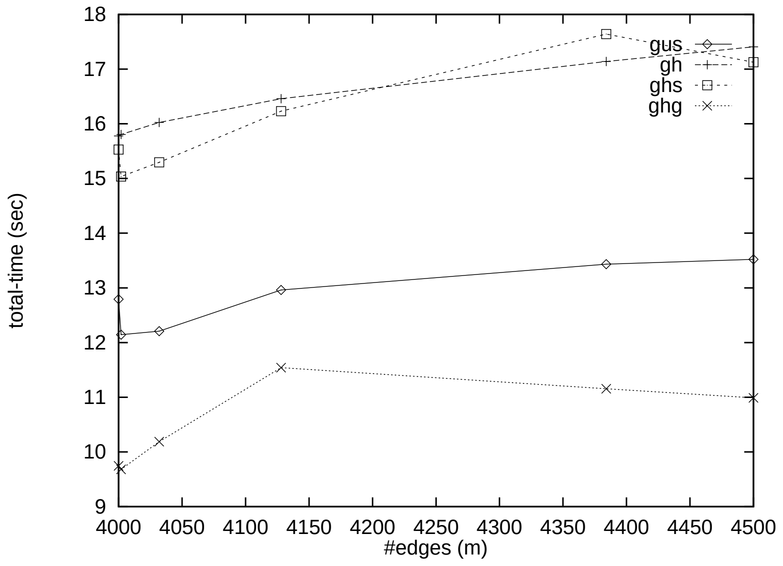

Figure 3.6: Running times for IRREG1 family ($K = 8$)

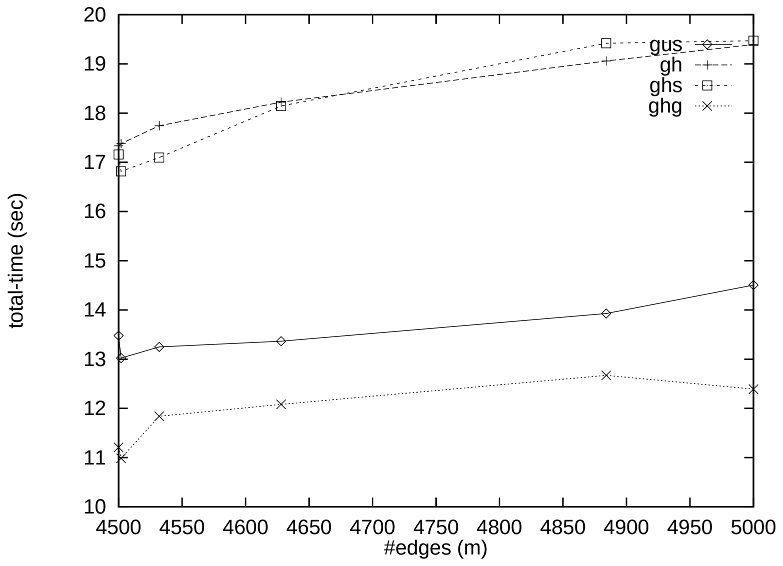

Figure 3.7: Running times for IRREG1 family ($K = 9$)

NOI1									
	N	M	Av.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	100	2475	100.00	2475.00	0.140	0.166	8072.400	20613.400	0.312
gh	100	2475	100.00	2001.80	0.162	0.236	10209.800	26180.200	0.404
ghs	100	2475	100.00	1999.80	0.184	0.224	7466.600	20015.800	0.414
ghg	100	2475	50.50	1010.00	0.306	0.164	15586.000	63282.000	0.474
gus	200	9950	200.00	9950.00	1.092	1.190	32507.200	80882.000	2.304
gh	200	9950	200.00	7934.80	1.390	1.932	40733.400	106244.600	3.330
ghs	200	9950	200.00	7932.80	1.406	1.978	30856.600	81365.200	3.394
ghg	200	9950	100.50	3986.33	2.630	1.446	63018.800	293091.600	4.084
gus	300	22425	300.00	22425.00	4.512	5.926	76437.200	190874.400	10.450
gh	300	22425	300.00	17784.20	5.340	9.338	91304.800	241702.000	14.692
ghs	300	22425	300.00	17782.20	6.140	9.192	73695.600	196178.600	15.338
ghg	300	22425	150.50	8920.83	11.194	6.900	146903.400	715597.600	18.098
gus	400	39900	400.00	39900.00	12.074	15.590	137977.000	345299.400	27.690
gh	400	39900	400.00	31615.40	13.810	25.774	161840.000	431264.800	39.606
ghs	400	39900	400.00	31613.40	17.056	25.244	134417.600	360402.600	42.318
ghg	400	39900	200.50	15846.31	27.858	19.100	245537.800	1216714.000	46.964
gus	500	62375	500.00	62375.00	25.572	30.642	220971.200	547866.400	56.256
gh	500	62375	500.00	49346.79	28.108	51.604	252343.000	675534.600	79.742
ghs	500	62375	500.00	49344.80	35.204	50.572	215844.600	581653.800	85.800
ghg	500	62375	250.50	24721.84	57.012	38.462	380651.800	1880512.600	95.490
gus	600	89850	600.00	89850.00	45.994	53.012	322304.800	793268.200	99.064
gh	600	89850	600.00	70937.80	50.120	89.686	362894.400	974327.800	139.832
ghs	600	89850	600.00	70935.80	62.702	87.972	316684.000	849284.000	150.706
ghg	600	89850	300.50	35527.11	102.768	66.756	553239.000	2772812.000	169.542
gus	700	122325	700.00	122325.00	74.892	84.312	440571.400	1091108.200	159.240
gh	700	122325	700.00	96604.79	80.788	143.050	493805.200	1331621.800	223.862
ghs	700	122325	700.00	96602.79	100.572	140.406	427978.400	1156990.000	241.020
ghg	700	122325	350.50	48370.50	166.184	106.776	754564.400	3916730.000	272.988
gus	800	159800	800.00	159800.00	113.326	126.118	572226.200	1417749.400	239.498
gh	800	159800	800.00	126152.79	123.284	214.504	646932.200	1751024.000	337.818
ghs	800	159800	800.00	126150.79	151.044	209.462	560811.600	1520480.600	360.544
ghg	800	159800	400.50	63154.34	264.006	159.752	1028238.600	5673518.400	423.786

Table 3.7: Data for NOI1 family

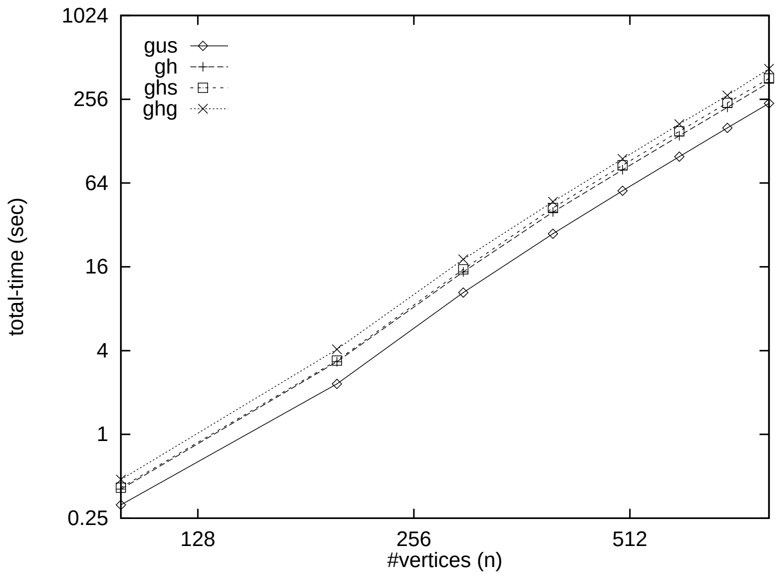

Figure 3.8: Running times for NOI1 family

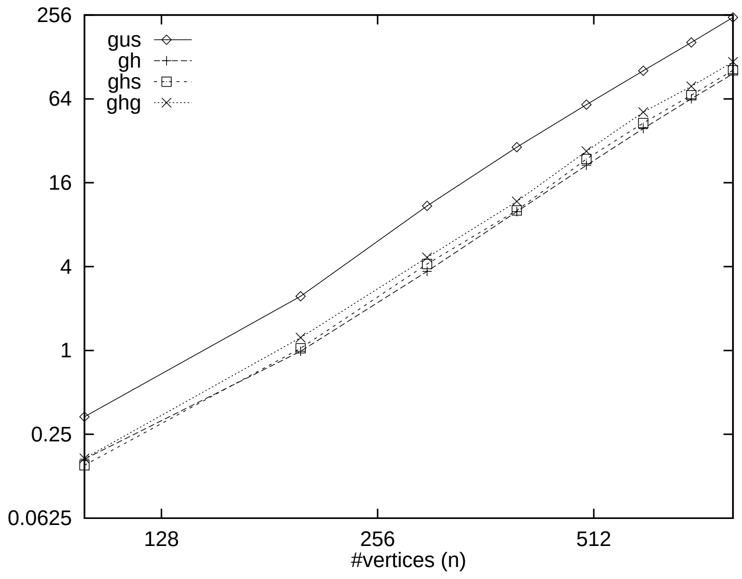

Figure 3.9: Running times for NOI2 family

NOI2									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	100	2475	100.000	2475.00	0.180	0.150	9405.200	29992.600	0.334
gh	100	2475	51.877	576.97	0.078	0.076	5715.800	17849.200	0.166
ghs	100	2475	52.632	606.60	0.060	0.088	3849.000	13698.600	0.150
ghg	100	2475	27.547	331.28	0.090	0.076	8295.800	33064.200	0.168
gus	200	9950	200.000	9950.00	1.288	1.154	37258.600	122806.800	2.448
gh	200	9950	101.851	2129.31	0.400	0.574	21855.400	71123.200	0.986
ghs	200	9950	104.755	2319.64	0.402	0.620	16094.600	56659.600	1.038
ghg	200	9950	52.853	1170.14	0.802	0.434	34968.200	159329.200	1.236
gus	300	22425	300.000	22425.00	5.018	5.868	83616.800	271293.400	10.898
gh	300	22425	151.963	4661.09	1.444	2.230	47927.800	157165.600	3.696
ghs	300	22425	157.648	5204.80	1.728	2.430	38543.000	134470.000	4.172
ghg	300	22425	78.567	2549.97	2.954	1.688	78217.200	366670.200	4.650
gus	400	39900	400.000	39900.00	13.194	15.578	150554.800	487276.200	28.800
gh	400	39900	201.916	8185.69	3.508	6.396	84492.400	281351.400	9.930
ghs	400	39900	202.461	8288.57	3.980	6.116	66213.000	231040.200	10.116
ghg	400	39900	102.461	4239.54	7.302	4.450	136564.000	662437.600	11.758
gus	500	62375	500.000	62375.00	27.312	30.870	237081.400	768829.400	58.206
gh	500	62375	252.429	12785.18	7.164	14.172	130190.800	437624.600	21.346
ghs	500	62375	258.742	13782.18	9.268	14.404	108203.400	374733.600	23.702
ghg	500	62375	129.830	6924.36	16.332	10.584	216869.800	1118776.200	26.922
gus	600	89850	600.000	89850.00	48.638	53.212	340552.200	1101504.000	101.882
gh	600	89850	302.181	18244.81	12.866	26.274	186688.600	628797.800	39.190
ghs	600	89850	307.650	19283.53	16.770	26.152	156110.800	542817.600	42.956
ghg	600	89850	157.417	10208.28	31.286	20.186	317276.800	1681369.600	51.486
gus	700	122325	700.000	122325.00	78.260	84.392	459326.200	1480889.200	162.702
gh	700	122325	352.168	24717.82	21.216	42.922	252435.800	854438.600	64.170
ghs	700	122325	353.985	25164.45	26.614	41.372	208409.600	727069.600	68.014
ghg	700	122325	178.165	12778.21	47.600	31.100	421098.600	2212331.600	78.718
gus	800	159800	800.000	159800.00	120.070	126.082	608897.400	1967348.800	246.202
gh	800	159800	402.020	32160.97	32.024	64.858	329014.800	1117358.800	96.922
ghs	800	159800	402.206	32281.82	40.684	61.830	277282.000	962146.200	102.540
ghg	800	159800	202.299	16321.77	71.264	46.570	536687.000	2785569.600	117.868

Table 3.8: Data for NOI2 family

NOI3									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	500	6237	500.000	6237.00	2.850	2.834	234598.600	330315.800	5.726
gh	500	6237	500.000	6328.00	4.034	4.686	322827.200	467052.800	8.744
ghs	500	6237	500.000	6326.00	3.288	4.496	223513.800	320174.400	7.802
ghg	500	6237	250.501	3169.33	6.838	3.250	460777.600	915968.600	10.102
gus	500	12475	500.000	12475.00	5.304	6.330	227950.200	365955.000	11.662
gh	500	12475	500.000	12119.60	6.814	9.696	289028.600	479595.800	16.536
ghs	500	12475	500.000	12117.60	6.516	9.576	210491.000	351918.600	16.116
ghg	500	12475	250.501	6070.94	13.196	7.082	441605.400	1136142.600	20.288
gus	500	31187	500.000	31187.00	13.546	17.132	223723.000	456445.400	30.712
gh	500	31187	500.000	27831.80	15.502	27.856	261829.600	564165.200	43.394
ghs	500	31187	500.000	27829.80	18.294	27.526	214188.800	464787.400	45.838
ghg	500	31187	250.501	13942.78	32.632	20.954	411799.400	1464214.000	53.604
gus	500	62375	500.000	62375.00	26.322	31.572	220971.200	547866.400	57.916
gh	500	62375	500.000	49346.79	28.852	52.654	252343.000	675534.600	81.534
ghs	500	62375	500.000	49344.80	36.056	51.928	215844.600	581653.800	88.012
ghg	500	62375	250.501	24721.84	58.980	39.994	380651.800	1880512.600	98.978
gus	500	93562	500.000	93562.00	37.748	44.144	220040.400	605356.000	81.934
gh	500	93562	500.000	66115.79	39.842	73.276	248833.400	744295.600	113.140
ghs	500	93562	500.000	66113.80	50.760	71.540	216279.600	649482.800	122.328
ghg	500	93562	250.501	33123.14	82.472	55.190	375962.800	2248610.600	137.678
gus	500	124750	500.000	124750.00	46.402	54.370	217498.000	630824.000	100.828
gh	500	124750	500.000	79109.40	48.900	90.230	247546.400	787930.200	139.156
ghs	500	124750	500.000	79107.39	62.470	87.550	214949.000	687242.800	150.030
ghg	500	124750	250.501	39632.96	95.706	67.000	356865.600	2446801.600	162.724

Table 3.9: Data for NOI3 family

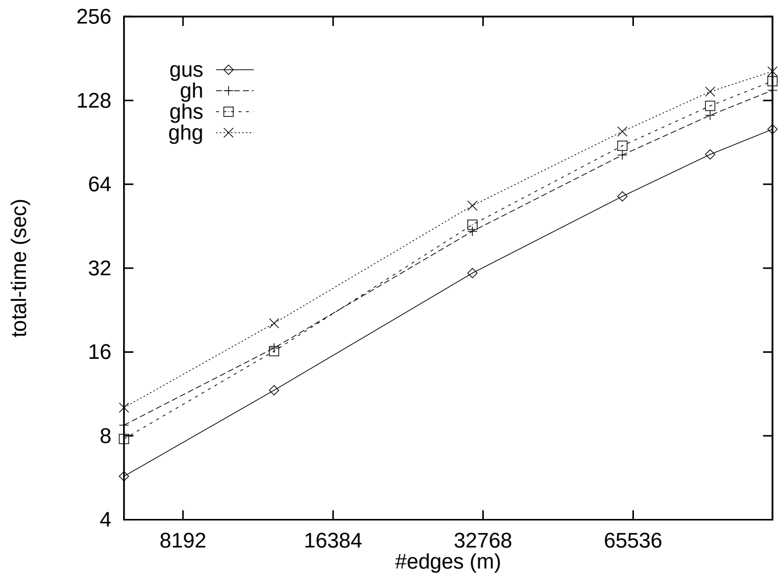

Figure 3.10: Running times for NOI3 family

NOI4									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	500	6237	500.000	6237.00	3.644	2.672	299293.600	554510.200	6.348
gh	500	6237	252.132	1932.21	1.658	1.690	202376.000	398719.400	3.370
ghs	500	6237	251.897	1933.07	1.370	1.664	126560.000	270927.800	3.062
ghg	500	6237	126.709	976.89	2.724	2.526	285626.000	670309.800	5.254
gus	500	12475	500.000	12475.00	5.700	5.956	256192.600	517099.000	11.692
gh	500	12475	252.653	3413.27	2.212	3.158	165373.000	365230.800	5.396
ghs	500	12475	253.184	3441.14	1.978	3.116	112728.600	267144.400	5.112
ghg	500	12475	127.600	1750.86	4.302	4.594	262285.000	722592.800	8.914
gus	500	31187	500.000	31187.00	14.142	16.606	241792.200	632689.800	30.794
gh	500	31187	252.846	7428.08	4.032	7.284	139160.800	389687.200	11.336
ghs	500	31187	253.974	7547.21	4.536	7.004	107543.200	313657.400	11.556
ghg	500	31187	128.498	3880.39	9.036	10.614	236752.400	894985.800	19.658
gus	500	62375	500.000	62375.00	27.098	30.666	234114.800	764217.000	57.798
gh	500	62375	252.846	12856.06	7.212	14.146	130832.600	441029.600	21.376
ghs	500	62375	257.453	13595.62	9.272	14.316	107365.200	374154.600	23.604
ghg	500	62375	130.971	7098.68	16.544	12.942	217833.000	1126862.800	29.492
gus	500	93562	500.000	93562.00	38.496	42.648	233186.800	846458.400	81.194
gh	500	93562	252.846	17110.38	9.862	20.260	127954.800	474673.000	30.158
ghs	500	93562	262.583	19145.87	14.016	21.210	108811.200	417675.800	35.248
ghg	500	93562	129.849	9244.61	22.144	15.234	205688.600	1287036.400	37.386
gus	500	124750	500.000	124750.00	47.552	52.964	230628.600	882537.200	100.560
gh	500	124750	252.550	20354.71	11.912	25.136	125878.000	496930.800	37.072
ghs	500	124750	267.743	24122.76	18.236	27.494	112151.400	460400.000	45.756
ghg	500	124750	131.624	11470.81	28.338	19.574	205917.400	1478475.600	47.924

Table 3.10: Data for NOI4 family

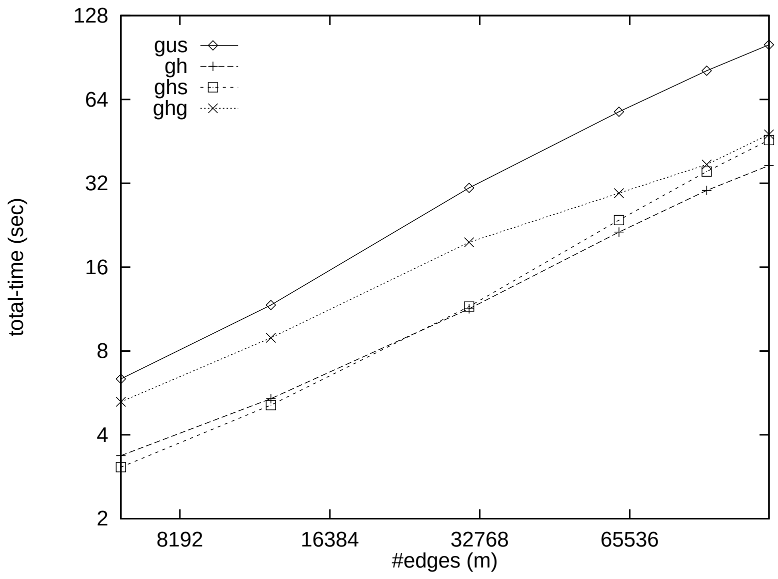

Figure 3.11: Running times for NOI4 family

NOI5										
	N	M	K	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	500	62375	1	500.000	62375.00	25.750	30.994	220971.2	547866.4	56.786
gh	500	62375	1	500.000	49346.79	28.122	51.388	252343.0	675534.6	79.552
ghs	500	62375	1	500.000	49344.80	35.646	51.256	215844.6	581653.8	86.934
ghg	500	62375	1	250.501	24721.84	57.232	38.588	380651.8	1880512.6	95.838
gus	500	62375	3	500.000	62375.00	28.770	30.862	247409.6	914064.0	59.654
gh	500	62375	3	171.527	6095.50	3.726	6.450	91066.8	356699.2	10.190
ghs	500	62375	3	181.966	7265.09	5.030	7.264	78182.2	350358.4	12.310
ghg	500	62375	3	88.705	3407.96	7.266	4.732	146781.6	731719.6	12.010
gus	500	62375	5	500.000	62375.00	31.186	30.954	260338.2	1137906.8	62.172
gh	500	62375	5	106.179	2511.96	1.832	2.912	59546.2	293668.8	4.766
ghs	500	62375	5	114.860	3321.28	2.518	3.446	50017.2	274181.8	5.992
ghg	500	62375	5	59.831	1920.63	4.346	2.854	101138.6	558784.2	7.214
gus	500	62375	10	500.000	62375.00	35.974	31.116	286834.4	1486421.4	67.122
gh	500	62375	10	64.779	1540.89	2.538	2.062	46341.4	388276.8	4.618
ghs	500	62375	10	68.250	1894.69	1.938	2.254	31876.40	231647.2	4.214
ghg	500	62375	10	38.011	1305.77	3.966	2.182	66670.2	464666.8	6.164
gus	500	62375	20	500.000	62375.00	44.764	31.082	329765.6	2039234.2	75.886
gh	500	62375	20	56.855	2333.32	4.920	3.006	59452.6	595017.0	7.952
ghs	500	62375	20	46.461	1625.09	2.238	2.134	26611.8	243525.6	4.402
ghg	500	62375	20	28.612	1249.94	5.054	2.162	57991.0	483311.2	7.232
gus	500	62375	33	500.000	62375.00	54.408	30.788	381868.6	2380417.8	85.228
gh	500	62375	33	71.868	4313.29	9.600	5.144	92870.4	859104.4	14.758
ghs	500	62375	33	40.293	1873.45	2.816	2.476	27230.8	260385.0	5.316
ghg	500	62375	33	27.572	1529.10	7.908	2.706	71487.8	631922.0	10.632
gus	500	62375	50	500.000	62375.00	56.012	30.814	392128.8	2379258.0	86.856
gh	500	62375	50	105.720	7861.90	16.824	9.212	143414.4	1167246.8	26.060
ghs	500	62375	50	49.181	2841.66	3.898	3.436	34782.2	297480.8	7.350
ghg	500	62375	50	32.944	2109.61	11.524	3.680	92422.0	858688.0	15.220
gus	500	62375	150	500.000	62375.00	41.260	30.714	321830.0	1317642.8	71.998
gh	500	62375	150	248.431	23843.52	35.484	27.596	254034.4	1465512.6	63.114
ghs	500	62375	150	176.290	15750.19	16.552	20.228	99246.8	466330.2	36.806
ghg	500	62375	150	96.215	8640.09	43.204	15.620	251426.2	2717844.0	58.842
gus	500	62375	300	500.000	62375.00	34.924	30.698	287580.2	941331.6	65.658
gh	500	62375	300	348.333	34539.50	37.862	38.226	283830.0	1305608.2	76.118
ghs	500	62375	300	289.413	28466.63	25.686	33.848	145519.4	524623.6	59.556
ghg	500	62375	300	149.117	14630.23	78.020	25.918	414365.6	4849200.2	103.962
gus	500	62375	500	500.000	62375.00	30.604	30.626	262441.4	760785.8	61.266
gh	500	62375	500	411.329	40854.05	36.796	43.902	291815.6	1132334.4	80.726
ghs	500	62375	500	369.372	36897.73	31.182	41.738	181231.2	572763.6	72.954
ghg	500	62375	500	187.771	18722.43	95.588	31.984	517790.2	5073999.2	127.576

Table 3.11: Data for NOI5 family

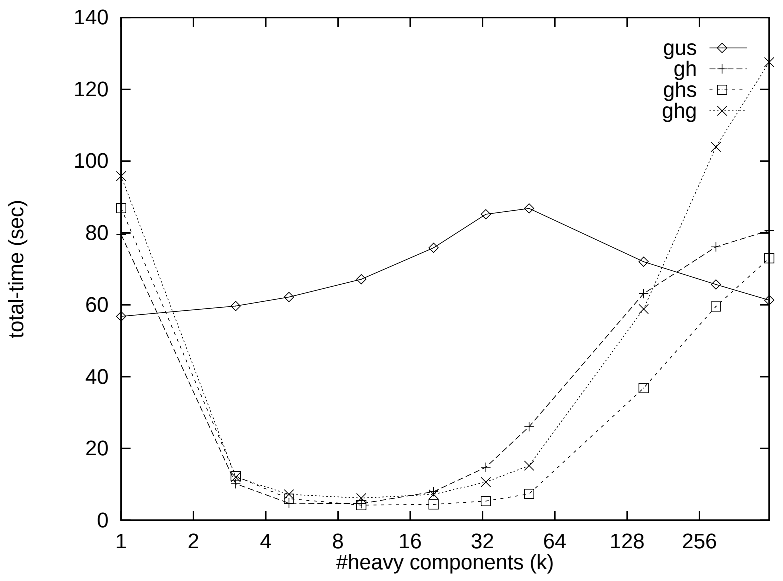

Figure 3.12: Running times for NOI5 family

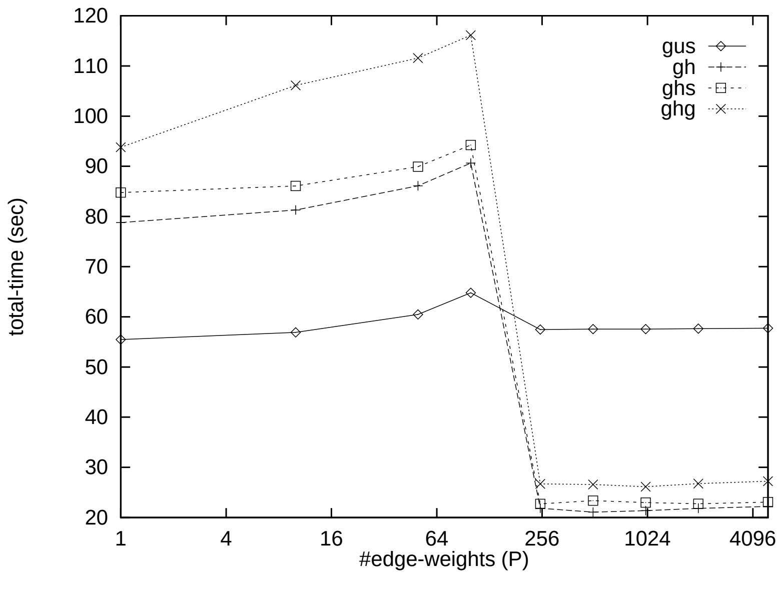

Figure 3.13: Running times for NOI6 family

NOI6										
	N	M	P	Av.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	500	62375	1	500.00	62375.00	25.038	30.406	219398.8	540051.0	55.466
gh	500	62375	1	500.00	49346.79	27.802	50.954	251919.2	669250.2	78.786
ghs	500	62375	1	500.00	49344.80	34.806	49.924	214554.8	571190.2	84.764
ghg	500	62375	1	250.50	24721.84	55.776	38.000	377032.2	1898937.0	93.786
gus	500	62375	10	500.00	62375.00	26.442	30.432	228514.8	889520.4	56.902
gh	500	62375	10	500.00	49346.79	30.182	51.078	269007.4	1053732.2	81.286
ghs	500	62375	10	500.00	49344.80	36.096	49.950	224058.6	933569.0	86.072
ghg	500	62375	10	250.50	24721.84	67.978	38.110	445558.4	2860203.4	106.110
gus	500	62375	50	500.00	62375.00	29.980	30.462	248876.2	2127012.4	60.482
gh	500	62375	50	500.00	49346.79	35.166	50.942	302002.4	2398172.0	86.122
ghs	500	62375	50	500.00	49344.80	39.934	49.964	246871.6	2211179.2	89.914
ghg	500	62375	50	250.50	24721.84	73.468	38.068	472712.2	5259467.4	111.558
gus	500	62375	100	500.00	62375.00	34.366	30.388	279440.2	3469834.2	64.786
gh	500	62375	100	500.00	49346.79	39.666	50.942	331127.6	3990786.0	90.636
ghs	500	62375	100	500.00	49344.80	44.158	50.036	274994.6	3585685.6	94.230
ghg	500	62375	100	250.50	24721.84	78.026	38.078	499262.4	6945915.8	116.128
gus	500	62375	250	500.00	62375.00	27.068	30.372	236391.8	788705.6	57.466
gh	500	62375	250	255.24	13199.00	7.556	14.256	132531.6	507056.6	21.832
ghs	500	62375	250	256.77	13486.43	8.802	13.864	106236.0	346881.8	22.698
ghg	500	62375	250	131.22	7143.68	15.984	10.708	212483.6	1071805.0	26.704
gus	500	62375	500	500.00	62375.00	27.094	30.406	237081.4	768829.4	57.544
gh	500	62375	500	252.42	12785.18	7.036	14.022	130190.8	437624.6	21.078
ghs	500	62375	500	258.74	13782.18	9.146	14.210	108203.4	374733.6	23.376
ghg	500	62375	500	129.83	6924.36	16.190	10.364	216869.8	1118776.2	26.570
gus	500	62375	1000	500.00	62375.00	27.162	30.360	236495.6	778270.4	57.554
gh	500	62375	1000	252.52	12799.72	7.476	13.880	135933.8	462982.0	21.372
ghs	500	62375	1000	257.09	13532.46	9.016	13.944	108466.8	392449.8	22.986
ghg	500	62375	1000	129.67	6900.01	15.768	10.356	214945.2	1081002.4	26.142
gus	500	62375	2000	500.00	62375.00	27.252	30.370	237338.2	788437.0	57.650
gh	500	62375	2000	252.42	12785.18	7.756	14.044	144343.6	490483.8	21.814
ghs	500	62375	2000	255.96	13364.02	8.894	13.826	109076.2	401186.4	22.738
ghg	500	62375	2000	130.52	7016.94	16.156	10.590	217461.2	1114206.0	26.758
gus	500	62375	5000	500.00	62375.00	27.358	30.338	237857.0	808932.4	57.730
gh	500	62375	5000	252.32	12770.47	8.136	14.050	153392.0	505017.0	22.208
ghs	500	62375	5000	256.84	13494.08	9.162	13.876	110165.4	412341.8	23.060
ghg	500	62375	5000	129.71	6903.91	16.840	10.368	226332.0	1164251.6	27.234

Table 3.12: Data for NOI6 family

PATH										
	N	M	K	Aver.N	Aver.M	CTime	MTime	Relabels	Pushes	TotTime
gus	2000	21990	1	2000.00	21990.00	2.738	71.926	11.0	39757.6	74.844
gh	2000	21990	1	2000.00	22864.00	53.860	111.120	3462058.2	5089280.4	165.076
ghs	2000	21990	1	2000.00	22862.00	19.528	112.792	11.0	39757.6	132.434
ghg	2000	21990	1	1000.50	11436.71	66.922	79.284	1674373.6	10123340.2	146.248
gus	2000	21990	4	2000.00	21990.00	14.940	71.836	757042.2	2022180.8	86.910
gh	2000	21990	4	1989.43	22709.44	80.414	110.346	5323845.6	9611058.8	190.858
ghs	2000	21990	4	504.14	2776.64	2.264	31.808	7482.6	73443.8	34.180
ghg	2000	21990	4	253.03	1398.47	5.224	16.604	254095.2	664068.2	21.886
gus	2000	21990	15	2000.00	21990.00	46.264	73.574	2950639.8	6537896.6	119.994
gh	2000	21990	15	1778.39	20386.26	130.034	100.612	7040216.0	16579151.4	230.760
ghs	2000	21990	15	143.34	628.79	1.196	24.778	30778.8	173520.2	26.070
ghg	2000	21990	15	75.21	354.88	2.288	11.702	118590.8	385170.6	14.042
gus	2000	21990	50	2000.00	21990.00	83.076	72.760	5057902.6	10189574.2	155.996
gh	2000	21990	50	1260.43	15504.24	136.430	81.156	6536603.2	18526364.4	217.706
ghs	2000	21990	50	60.54	381.10	2.194	23.872	90910.4	403747.6	26.186
ghg	2000	21990	50	38.56	294.20	3.900	11.676	177543.4	722060.4	15.636
gus	2000	21990	200	2000.00	21990.00	110.852	71.096	6804711.8	12232899.8	182.078
gh	2000	21990	200	266.61	3667.48	42.074	35.896	2037790.2	6330626.4	78.070
ghs	2000	21990	200	43.03	458.31	4.270	24.228	196448.2	636261.6	28.586
ghg	2000	21990	200	34.97	402.49	8.172	12.656	347545.6	1285304.8	20.882
gus	2000	21990	800	2000.00	21990.00	242.030	71.264	13343243.8	24378523.8	313.436
gh	2000	21990	800	124.97	1846.75	36.848	29.216	1658893.6	4227637.4	66.180
ghs	2000	21990	800	64.56	956.40	11.258	25.950	499685.4	1265666.4	37.322
ghg	2000	21990	800	50.95	731.54	32.882	15.852	1243339.8	3886804.2	48.794
gus	2000	21990	2000	2000.00	21990.00	554.889	70.765	28851668.8	50298108.4	625.804
gh	2000	21990	2000	126.65	2030.55	62.116	29.720	2671666.2	5836707.4	91.926
ghs	2000	21990	2000	103.92	1701.29	34.484	28.802	1501881.4	3255455.4	63.360
ghg	2000	21990	2000	67.00	1053.02	89.398	19.110	3371604.6	8772053.8	108.568

Table 3.13: Data for PATH family

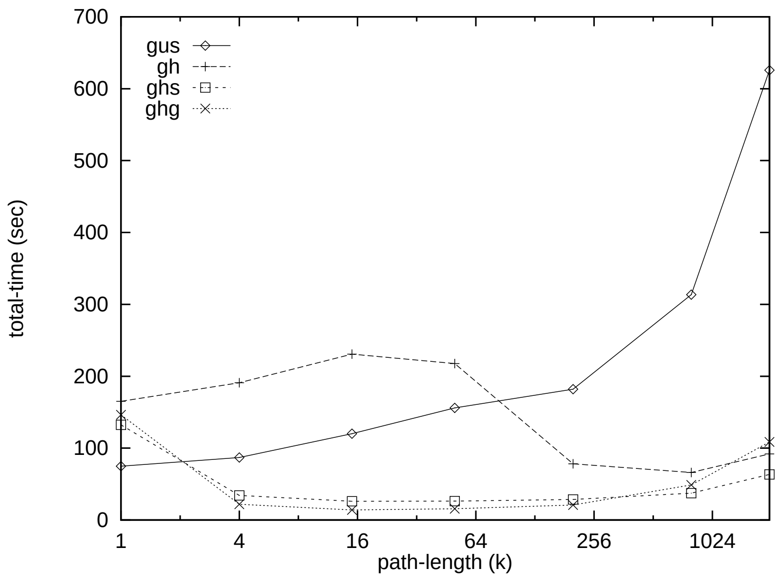

Figure 3.14: Running times for PATH family

PR1									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	200	583	200.000	583.400	0.198	0.184	38043.20	49857.20	0.390
gh	200	583	191.417	667.144	0.288	0.256	60242.60	80960.00	0.550
ghs	200	583	180.615	630.366	0.162	0.230	32129.40	41757.00	0.404
ghg	200	583	92.651	323.107	0.420	0.166	73554.00	124632.60	0.586
gus	400	1968	400.000	1968.000	0.844	0.964	145920.60	178316.20	1.848
gh	400	1968	399.895	2169.843	1.440	1.304	228495.00	287433.80	2.770
ghs	400	1968	399.645	2166.681	1.050	1.262	136425.20	166105.00	2.332
ghg	400	1968	200.179	1085.436	2.140	0.922	286573.20	465718.60	3.076
gus	600	4157	600.000	4157.000	2.548	2.644	306644.00	360422.20	5.230
gh	600	4157	600.000	4459.000	3.914	4.242	481212.20	584416.80	8.192
ghs	600	4157	600.000	4457.000	2.850	4.388	298350.40	352604.20	7.256
ghg	600	4157	300.387	2232.006	5.538	2.988	607139.20	956597.20	8.544
gus	800	7175	800.000	7175.200	5.878	7.832	549325.20	636626.00	13.748
gh	800	7175	800.000	7577.200	9.300	12.786	826197.40	986157.00	22.136
ghs	800	7175	800.000	7575.200	7.956	12.982	506020.00	591347.80	20.974
ghg	800	7175	400.375	3791.932	14.344	9.640	1046093.20	1620243.40	24.006
gus	1000	10923	1000.000	10923.400	9.976	15.514	865278.80	987098.80	25.580
gh	1000	10923	1000.000	11425.400	15.470	22.460	1252763.20	1470220.60	37.968
ghs	1000	10923	1000.000	11423.400	12.726	22.664	806876.20	931668.00	35.430
ghg	1000	10923	500.472	5717.315	22.302	15.438	1550385.20	2254644.00	37.766

Table 3.14: Data for PR1 family

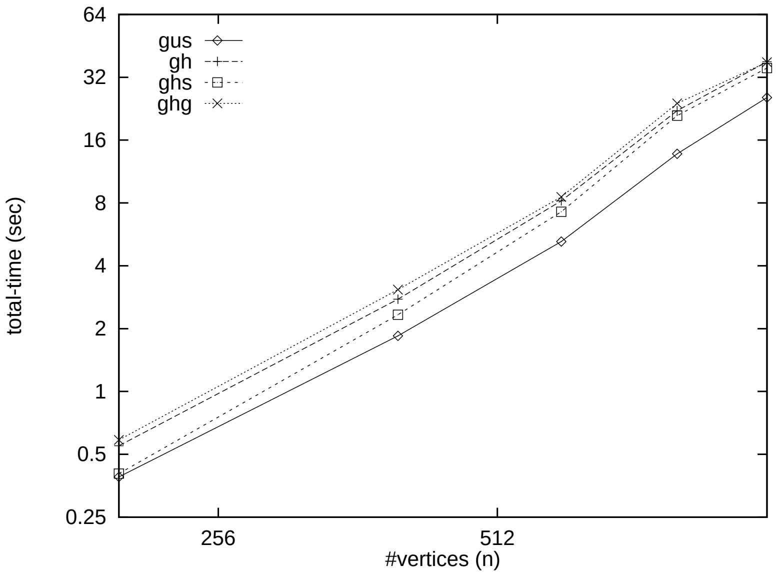

Figure 3.15: Running times for PR1 family

PR5									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	200	583	200.000	583.400	0.302	0.180	63118.40	95808.80	0.498
gh	200	583	79.653	273.606	0.168	0.148	34759.00	61253.40	0.328
ghs	200	583	61.963	214.261	0.090	0.128	15153.40	28268.60	0.228
ghg	200	583	34.907	119.939	0.196	0.068	30987.80	64216.20	0.266
gus	400	1968	400.000	1968.000	1.444	0.916	222721.00	331643.80	2.398
gh	400	1968	193.153	858.583	0.848	0.688	150966.20	262749.40	1.560
ghs	400	1968	182.832	816.417	0.580	0.670	81575.80	149125.20	1.260
ghg	400	1968	94.265	420.948	1.188	0.508	184163.60	374778.60	1.712
gus	600	4157	600.000	4157.000	3.668	2.938	470455.20	683390.80	6.650
gh	600	4157	299.150	1629.344	2.068	2.042	322511.20	555542.60	4.146
ghs	600	4157	295.839	1614.090	1.534	2.030	186272.80	338361.20	3.596
ghg	600	4157	149.336	815.801	3.330	1.288	432119.20	870631.60	4.636
gus	800	7175	800.000	7175.200	8.742	7.864	790617.00	1133614.00	16.668
gh	800	7175	400.874	2585.057	4.094	4.856	530152.20	916293.20	8.972
ghs	800	7175	400.393	2589.191	3.210	4.882	320745.80	579301.00	8.140
ghg	800	7175	201.171	1304.450	7.650	3.548	767179.00	1535033.80	11.216
gus	1000	10923	1000.000	10923.400	14.468	15.752	1220926.20	1729318.00	30.300
gh	1000	10923	501.456	3727.116	6.896	9.450	795712.80	1365773.00	16.400
ghs	1000	10923	501.725	3734.425	5.436	9.126	506897.40	919886.80	14.612
ghg	1000	10923	251.623	1876.382	11.292	5.824	1128825.20	2195287.20	17.144

Table 3.15: Data for PR5 family

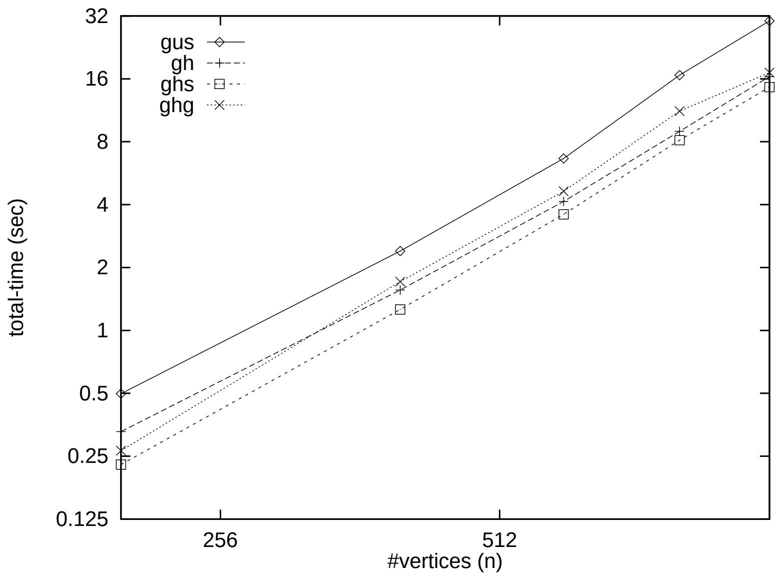

Figure 3.16: Running times for PR5 family

PR6									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	200	2172	200.000	2171.800	0.472	0.376	41935.600	85789.60	0.868
gh	200	2172	101.361	742.568	0.232	0.276	28228.600	61798.60	0.520
ghs	200	2172	102.459	761.375	0.188	0.238	16747.000	40648.60	0.434
ghg	200	2172	52.117	391.956	0.372	0.164	39328.400	104808.80	0.536
gus	400	8307	400.000	8307.200	3.122	2.802	162708.600	328246.20	5.960
gh	400	8307	201.997	2490.413	1.292	1.436	103221.200	229085.20	2.744
ghs	400	8307	202.097	2500.436	1.100	1.458	69329.400	166563.80	2.574
ghg	400	8307	101.849	1269.926	2.248	1.056	157521.000	429883.40	3.324
gus	600	18481	600.000	18481.400	10.906	12.670	360963.800	726273.20	23.624
gh	600	18481	301.499	5197.957	3.822	5.822	219253.400	490586.80	9.674
ghs	600	18481	302.198	5241.647	3.496	5.754	154055.000	366086.20	9.284
ghg	600	18481	152.351	2659.269	7.422	3.940	354662.400	991072.20	11.372
gus	800	32740	800.000	32740.000	28.196	32.610	633331.400	1258017.00	60.874
gh	800	32740	401.499	8946.378	8.898	15.186	378634.600	854925.80	24.124
ghs	800	32740	402.099	8999.726	8.768	14.792	277902.600	653849.60	23.602
ghg	800	32740	202.351	4549.897	18.868	10.568	626897.400	1753289.40	29.458
gus	1000	50959	1000.000	50959.200	56.828	62.902	992343.800	1970744.60	119.812
gh	1000	50959	501.500	13734.130	17.776	32.628	579732.800	1318336.40	50.452
ghs	1000	50959	502.399	13824.069	18.886	31.958	439993.200	1032512.80	50.906
ghg	1000	50959	252.351	6967.647	39.202	22.866	981210.200	2750153.80	62.100

Table 3.16: Data for PR6 family

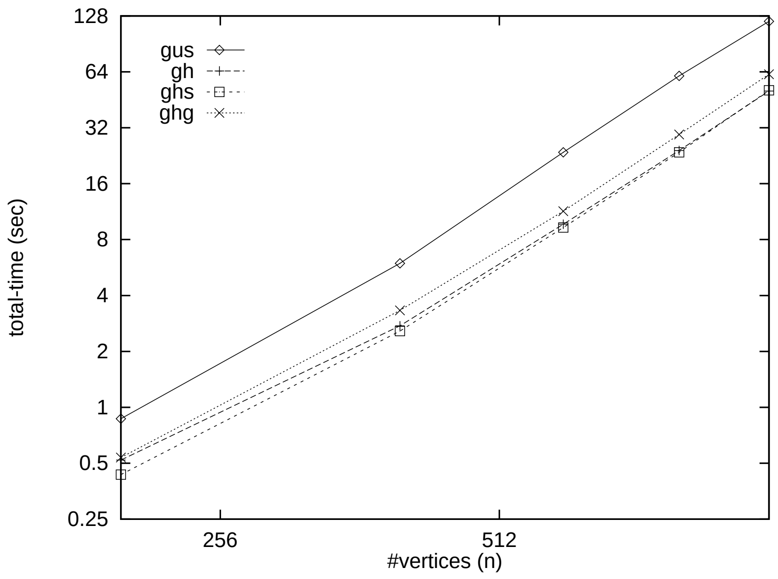

Figure 3.17: Running times for PR6 family

PR7									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	200	10053	200.000	10053.400	1.606	1.518	36113.400	128901.60	3.136
gh	200	10053	101.497	2693.447	0.500	0.714	20294.800	71946.00	1.230
ghs	200	10053	101.995	2753.068	0.564	0.684	15493.600	59643.60	1.256
ghg	200	10053	51.956	1416.748	0.900	0.506	30685.400	158949.60	1.414
gus	300	22564	300.000	22564.400	6.438	7.092	83126.600	311260.60	13.560
gh	300	22564	151.498	5917.411	1.750	2.840	45649.000	165996.80	4.614
ghs	300	22564	152.096	6019.273	1.860	2.772	35552.600	136054.00	4.646
ghg	300	22564	77.354	3101.000	3.420	1.922	71827.400	396020.80	5.358
gus	400	40047	400.000	40047.200	16.798	18.482	149747.200	569512.00	35.302
gh	400	40047	201.499	10343.649	4.336	8.018	80879.000	295589.60	12.368
ghs	400	40047	201.997	10465.661	5.106	7.688	65574.200	249304.60	12.812
ghg	400	40047	102.253	5346.542	8.788	5.636	127392.200	700936.60	14.434
gus	500	62596	500.000	62595.800	34.276	36.478	232183.800	870089.20	70.794
gh	500	62596	251.499	16100.493	9.172	17.994	126032.200	463793.60	27.192
ghs	500	62596	251.998	16253.585	11.084	17.460	102698.000	391509.60	28.558
ghg	500	62596	127.253	8268.730	19.106	12.946	200461.600	1150734.60	32.068
gus	600	90090	600.000	90090.400	61.468	63.182	337965.400	1294392.80	124.714
gh	600	90090	301.499	23028.875	16.372	33.302	181342.000	670525.60	49.704
ghs	600	90090	302.298	23280.924	21.198	32.222	152022.600	580170.00	53.452
ghg	600	90090	152.552	11844.242	36.840	24.326	297449.400	1771142.40	61.178

Table 3.17: Data for PR7 family

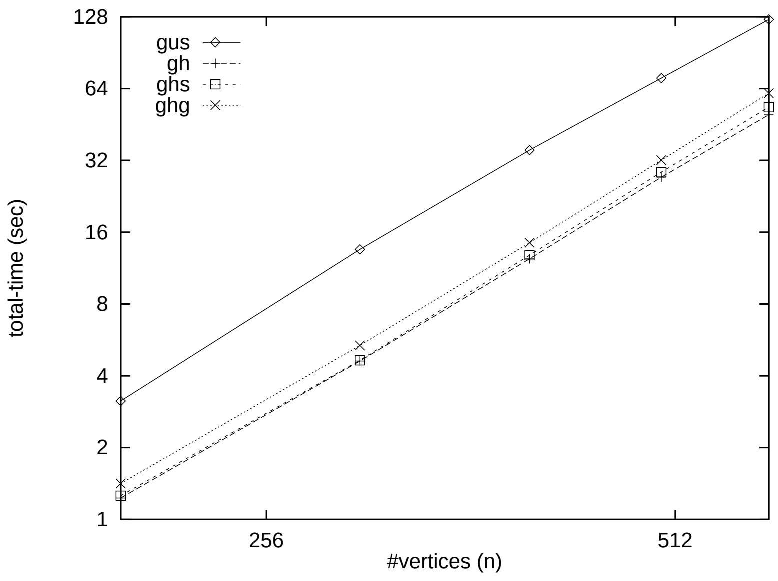

Figure 3.18: Running times for PR7 family

PR8									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotTime
gus	200	19694	200.000	19693.800	3.544	3.810	35768.000	165378.00	7.372
gh	200	19694	101.497	5074.955	0.970	1.448	19206.400	85271.00	2.430
ghs	200	19694	102.492	5269.992	1.022	1.494	15393.600	72885.40	2.528
ghg	200	19694	52.754	2784.239	1.706	1.106	28500.400	221103.00	2.820
gus	300	44397	300.000	44396.600	14.204	15.402	81803.200	374690.80	29.628
gh	300	44397	151.498	11324.736	3.458	6.488	43257.800	192910.00	9.968
ghs	300	44397	151.997	11507.513	4.170	6.250	35757.600	168378.80	10.434
ghg	300	44397	77.254	5921.627	6.554	4.664	62164.000	516311.00	11.226
gus	400	79002	400.000	79002.200	35.896	36.846	147182.000	694794.20	72.774
gh	400	79002	201.499	20051.519	9.154	18.444	77549.600	347151.60	27.628
ghs	400	79002	202.596	20474.217	11.956	17.758	66382.200	310587.60	29.740
ghg	400	79002	102.852	10549.703	18.462	13.772	114846.800	964515.80	32.242
gus	500	123495	500.000	123495.000	73.118	72.542	230464.400	1082868.80	145.686
gh	500	123495	251.499	31249.892	18.870	38.124	121357.800	548266.80	57.018
ghs	500	123495	252.597	31779.008	24.914	36.810	102977.800	482336.60	61.754
ghg	500	123495	127.451	16156.218	35.880	27.774	171577.200	1529135.20	63.666
gus	600	177903	600.000	177903.000	126.522	125.338	333352.800	1580193.20	251.906
gh	600	177903	301.499	44922.063	33.806	66.898	175337.600	789361.20	100.722
ghs	600	177903	302.497	45513.105	44.560	64.370	152255.000	709552.80	108.970
ghg	600	177903	152.751	23202.804	67.874	48.854	260187.200	2381038.00	116.736

Table 3.18: Data for PR8 family

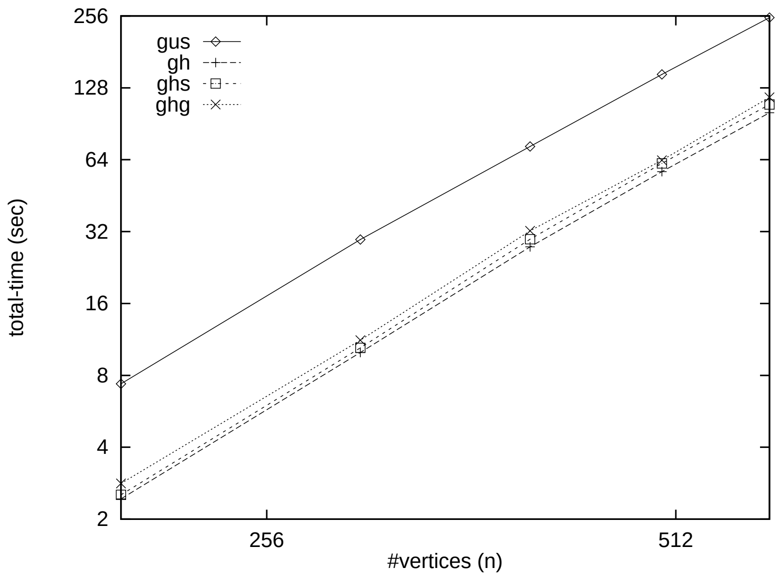

Figure 3.19: Running times for PR8 family

REG1									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	301	301	301.000	301.000	0.268	0.288	89102.000	89700.00	0.572
gh	301	301	301.000	453.500	0.316	0.314	89100.600	89700.00	0.648
ghs	301	301	301.000	451.500	0.306	0.330	89102.000	89700.00	0.664
ghg	301	301	301.000	451.500	0.472	0.350	89102.000	89700.00	0.834
gus	301	602	301.000	602.000	0.330	0.342	92460.600	96053.40	0.692
gh	301	602	301.000	752.100	0.362	0.452	90589.000	96899.60	0.824
ghs	301	602	301.000	750.100	0.386	0.476	82518.000	87639.20	0.882
ghg	301	602	79.731	237.356	0.238	0.264	41614.800	53610.60	0.516
gus	301	1505	301.000	1505.000	0.516	0.498	88339.200	96316.00	1.034
gh	301	1505	301.000	1638.900	0.492	0.756	86939.000	94634.20	1.264
ghs	301	1505	301.000	1636.900	0.614	0.734	87014.000	94640.20	1.364
ghg	301	1505	99.064	714.244	0.556	0.378	58392.000	82351.80	0.950
gus	301	4816	301.000	4816.000	1.220	1.138	87408.000	103524.80	2.386
gh	301	4816	301.000	4743.900	1.308	1.976	88418.200	106863.20	3.298
ghs	301	4816	301.000	4741.900	1.528	2.040	87735.400	104425.20	3.578
ghg	301	4816	107.843	2120.813	1.596	0.924	73879.400	149285.40	2.538
gus	301	15050	301.000	15050.000	3.310	3.530	88835.800	134363.20	6.872
gh	301	15050	301.000	12999.900	3.432	5.350	88676.600	143933.20	8.794
ghs	301	15050	301.000	12997.900	4.046	5.282	88771.200	136157.2	9.352
ghg	301	15050	126.786	5847.286	6.440	2.588	113434.800	487002.40	9.036
gus	301	49966	301.000	49966.000	10.778	12.360	88941.600	199804.80	23.154
gh	301	49966	301.000	30416.700	10.312	19.950	89063.400	223790.00	30.274
ghs	301	49966	301.000	30414.700	14.024	19.362	88995.000	209584.20	33.404
ghg	301	49966	136.869	12248.858	24.042	8.968	148442.200	1075426.20	33.036
gus	301	90300	301.000	90300.000	15.122	17.150	89104.600	231682.80	32.282
gh	301	90300	301.000	39205.700	14.218	27.546	89147.800	259904.40	41.786
ghs	301	90300	301.000	39203.700	19.256	26.356	89147.000	246501.00	45.628
ghg	301	90300	170.299	18951.678	41.610	15.172	187139.000	1530682.40	56.792

Table 3.19: Data for REG1 family

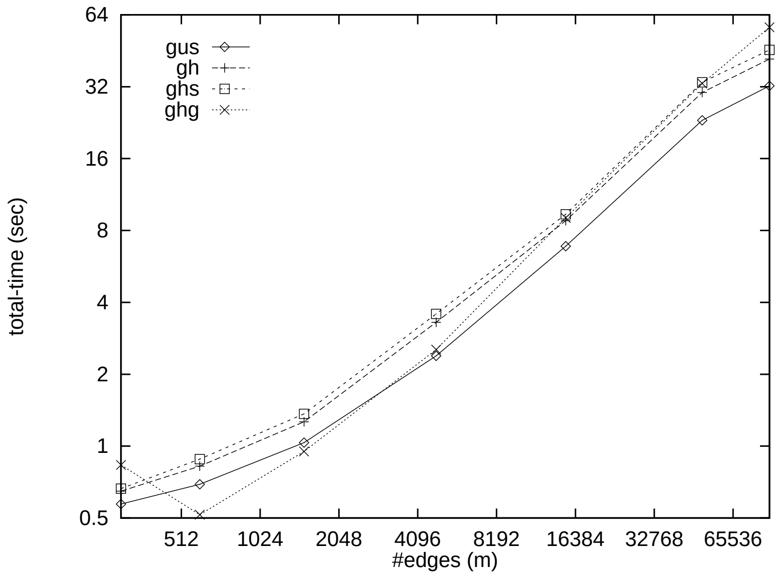

Figure 3.20: Running times for REG1 family

REG2									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	50	1250	50.000	1250.000	0.026	0.044	2247.800	4994.200	0.076
gh	50	1250	50.000	827.000	0.032	0.042	2252.000	5212.400	0.076
ghs	50	1250	50.000	825.000	0.038	0.052	2240.000	5097.000	0.090
ghg	50	1250	23.082	326.418	0.056	0.026	3150.800	11222.000	0.082
gus	100	2500	100.000	2500.000	0.152	0.168	9438.800	16356.200	0.322
gh	100	2500	100.000	2036.000	0.190	0.212	9446.000	17180.400	0.402
ghs	100	2500	100.000	2034.000	0.204	0.222	9463.800	16765.800	0.430
ghg	100	2500	42.303	841.432	0.252	0.092	11547.800	41391.400	0.356
gus	200	5000	200.000	5000.000	0.740	0.710	38813.200	54391.800	1.462
gh	200	5000	200.000	4521.200	0.748	0.982	38775.200	56663.000	1.732
ghs	200	5000	200.000	4519.200	0.742	1.176	38704.800	54660.800	1.930
ghg	200	5000	82.338	2064.235	1.128	0.466	42566.200	122155.000	1.596
gus	400	10000	400.000	10000.000	3.372	3.300	157202.400	190449.400	6.692
gh	400	10000	400.000	9626.800	3.450	5.402	156654.000	195953.000	8.866
ghs	400	10000	400.000	9624.800	4.270	5.512	157578.600	192128.000	9.806
ghg	400	10000	144.792	4299.959	4.778	2.626	137136.200	310780.000	7.422
gus	800	20000	800.000	20000.000	15.212	19.480	632617.600	703019.400	34.750
gh	800	20000	800.000	19806.600	15.520	29.952	631304.800	715456.200	45.506
ghs	800	20000	800.000	19804.600	20.186	29.752	629028.200	701329.000	49.968
ghg	800	20000	257.503	8466.638	18.310	13.130	449468.200	766573.600	31.480

Table 3.20: Data for REG2 family

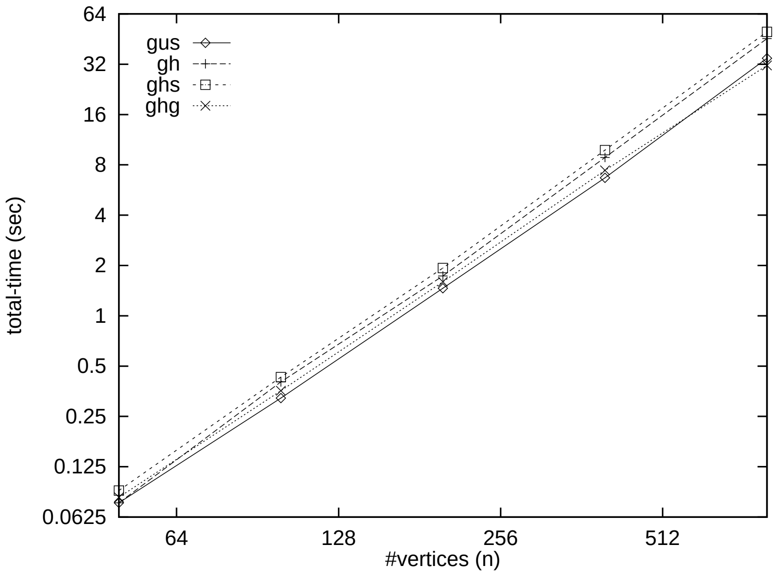

Figure 3.21: Running times for REG2 family

TREE										
	N	M	K	Av.N	Aver.M	CutTime	MTime	Relabels	Pushes	TotTime
gus	800	160600	1	800.00	160600.00	14.668	131.290	127.2	251171.0	146.026
gh	800	160600	1	800.00	126637.99	139.682	225.440	654496.8	3382659.2	365.164
ghs	800	160600	1	800.00	126635.99	51.730	215.438	127.2	251171.8	267.196
ghg	800	160600	1	400.50	63397.24	195.330	164.424	353357.4	31922655.0	359.770
gus	800	160600	3	800.00	160600.00	46.530	131.504	170829.8	794471.0	178.098
gh	800	160600	3	793.24	125034.00	167.616	224.196	765747.2	4583496.4	391.840
ghs	800	160600	3	403.76	38863.52	18.934	73.984	7531.6	284183.8	92.936
ghg	800	160600	3	202.33	19504.64	50.922	55.570	130895.8	8917101.4	106.522
gus	800	160600	5	800.00	160600.00	59.126	131.236	237671.4	1124891.4	190.412
gh	800	160600	5	792.88	124943.02	185.174	219.818	849349.4	5763163.0	405.030
ghs	800	160600	5	296.02	22990.66	11.962	44.134	11237.4	313122.4	56.118
ghg	800	160600	5	148.79	11610.35	28.988	32.786	91232.2	5175053.6	61.796
gus	800	160600	10	800.00	160600.00	80.516	131.440	349191.0	1602387.4	212.018
gh	800	160600	10	789.17	124159.16	219.236	220.068	968735.2	6947990.2	439.336
ghs	800	160600	10	194.26	11909.41	7.200	23.224	16703.4	355732.0	30.476
ghg	800	160600	10	98.26	6132.02	15.710	17.238	67740.8	2736431.0	32.962
gus	800	160600	20	800.00	160600.00	98.936	131.174	451982.8	2125038.4	230.176
gh	800	160600	20	776.14	121513.25	245.868	216.928	1055504.2	7735155.0	462.838
ghs	800	160600	20	131.73	7341.17	5.842	14.622	22917.2	408319.6	20.496
ghg	800	160600	20	68.12	3990.48	12.174	11.434	61921.6	1857187.8	23.626
gus	800	160600	50	800.00	160600.00	115.680	131.382	545664.4	2696860.8	247.110
gh	800	160600	50	727.92	112241.48	268.230	202.242	1109984.0	8832373.8	470.506
ghs	800	160600	50	99.22	6610.69	7.598	13.654	37106.6	507136.8	21.286
ghg	800	160600	50	54.42	3978.67	17.628	11.798	87508.6	1859914.0	29.450
gus	800	160600	100	800.00	160600.00	126.788	131.646	598343.8	3171652.0	258.488
gh	800	160600	100	664.09	101533.40	258.378	186.488	1054070.2	8568625.0	444.904
ghs	800	160600	100	102.32	8889.52	11.888	18.540	55722.2	616169.8	30.464
ghg	800	160600	100	60.02	5679.07	35.894	17.040	162158.4	3017374.2	52.946
gus	800	160600	200	800.00	160600.00	138.074	131.416	644957.2	3737396.0	269.544
gh	800	160600	200	594.63	91044.89	259.294	172.300	1005457.8	8267321.0	431.610
ghs	800	160600	200	121.06	13033.33	20.328	27.702	86009.0	798572.0	48.064
ghg	800	160600	200	75.15	8543.69	77.432	26.386	303894.2	5516712.8	103.842
gus	800	160600	400	800.00	160600.00	154.596	131.048	704840.2	4323066.8	285.720
gh	800	160600	400	508.54	78112.80	242.806	152.198	930439.0	7687778.2	395.044
ghs	800	160600	400	156.70	19788.66	32.374	42.148	123498.2	1073789.4	74.558
ghg	800	160600	400	98.21	12798.78	144.732	39.998	502969.0	9447691.2	184.744
gus	800	160600	800	800.00	160600.00	185.920	131.012	809714.6	5163665.6	316.990
gh	800	160600	800	449.86	69666.43	238.776	137.512	895883.2	7362390.4	376.320
ghs	800	160600	800	192.21	26378.75	46.742	55.636	166621.6	1424371.0	102.430
ghg	800	160600	800	115.63	16127.19	212.824	49.914	696121.4	12522017.2	262.764

Table 3.21: Data for TREE family

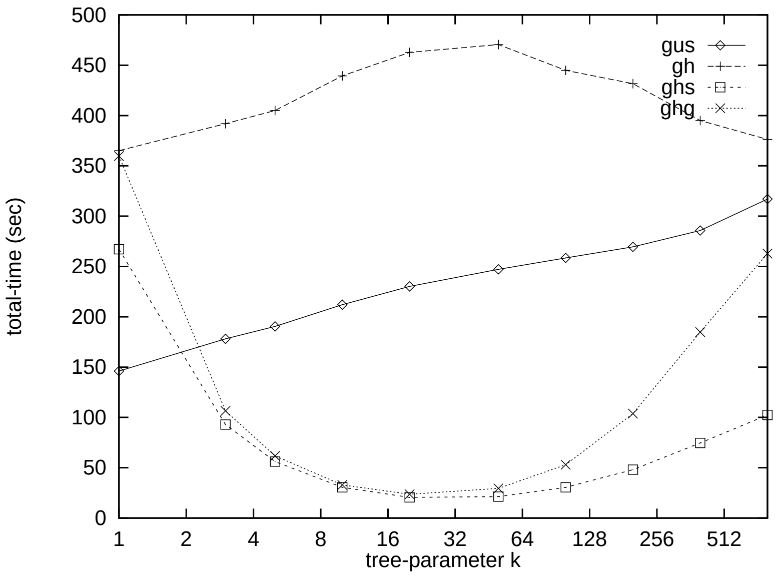

Figure 3.22: Running times for TREE family

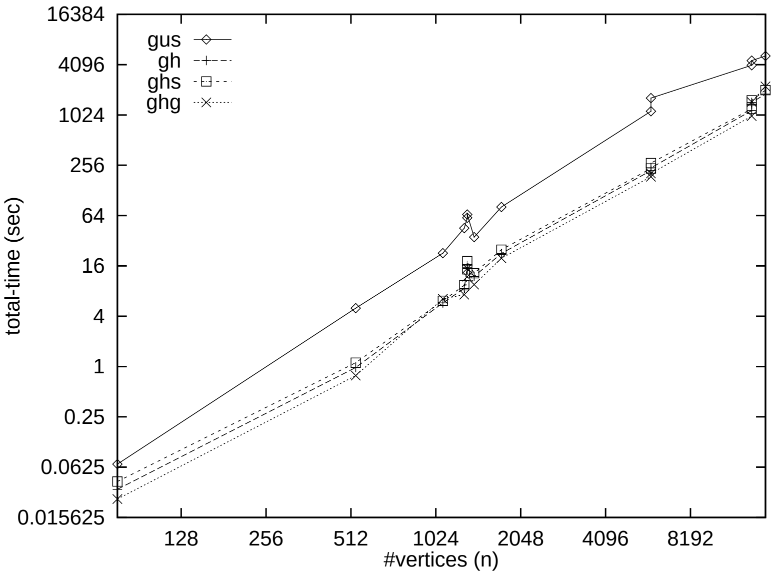

Figure 3.23: Running times for TSP family

TSP							
	Name	N	M	Aver.N	Aver.M	CutTime	ManTime
gus	tsp.pr76.x.2	76	90	76.000	90.000	0.046	0.018
gh	tsp.pr76.x.2	76	90	26.360	47.140	0.014	0.018
ghs	tsp.pr76.x.2	76	90	30.813	53.500	0.020	0.018
ghg	tsp.pr76.x.2	76	90	20.813	35.380	0.014	0.010
gus	tsp.att532.x.1	532	787	532.000	787.000	4.028	0.930
gh	tsp.att532.x.1	532	787	44.034	90.348	0.400	0.544
ghs	tsp.att532.x.1	532	787	52.652	104.667	0.512	0.560
ghg	tsp.att532.x.1	532	787	19.454	37.599	0.242	0.508
gus	tsp.vm1084.x.1	1084	1252	1084.000	1252.000	16.850	5.778
gh	tsp.vm1084.x.1	1084	1252	151.038	251.206	1.784	3.912
ghs	tsp.vm1084.x.1	1084	1252	148.443	244.495	2.174	3.898
ghg	tsp.vm1084.x.1	1084	1252	69.059	113.071	2.642	3.758
gus	tsp.d1291.x.1	1291	1942	1291.000	1942.000	34.922	10.210
gh	tsp.d1291.x.1	1291	1942	49.640	108.058	1.692	6.858
ghs	tsp.d1291.x.1	1291	1942	65.643	133.611	2.528	6.812
ghg	tsp.d1291.x.1	1291	1942	32.154	65.568	1.768	5.398
gus	tsp.rl1323.x.1	1323	2169	1323.000	2169.000	48.742	11.360
gh	tsp.rl1323.x.1	1323	2169	145.005	347.968	7.476	7.862
ghs	tsp.rl1323.x.1	1323	2169	179.663	400.577	6.494	8.012
ghg	tsp.rl1323.x.1	1323	2169	84.682	187.326	5.492	6.324
gus	tsp.rl1323.x.2	1323	2195	1323.000	2195.000	54.468	11.472
gh	tsp.rl1323.x.2	1323	2195	136.018	331.185	6.970	7.898
ghs	tsp.rl1323.x.2	1323	2195	161.665	367.155	10.320	7.904
ghg	tsp.rl1323.x.2	1323	2195	87.026	198.292	8.554	6.276
gus	tsp.fl1400.x.1	1400	2231	1400.000	2231.000	22.182	13.050
gh	tsp.fl1400.x.1	1400	2231	104.392	221.533	2.770	9.140
ghs	tsp.fl1400.x.1	1400	2231	153.761	296.658	3.902	9.102
ghg	tsp.fl1400.x.1	1400	2231	54.195	107.243	2.134	7.360
gus	tsp.vm1748.x.1	1748	2336	1748.000	2336.000	57.994	22.826
gh	tsp.vm1748.x.1	1748	2336	84.007	179.390	5.760	16.740
ghs	tsp.vm1748.x.1	1748	2336	146.673	276.722	7.700	17.180
ghg	tsp.vm1748.x.1	1748	2336	83.975	156.121	6.270	13.430
gus	tsp.r15934.x.1	5934	7287	5934.000	7287.000	872.524	259.342
gh	tsp.r15934.x.1	5934	7287	94.734	179.307	21.992	198.364
ghs	tsp.r15934.x.1	5934	7287	166.762	290.145	34.826	199.054
ghg	tsp.r15934.x.1	5934	7287	74.835	132.483	19.593	167.161
gus	tsp.r15934.x.2	5934	7627	5934.000	7627.000	1361.879	264.833
gh	tsp.r15934.x.2	5934	7627	124.668	248.837	38.140	199.346
ghs	tsp.r15934.x.2	5934	7627	160.541	294.561	72.360	199.152
ghg	tsp.r15934.x.2	5934	7627	75.964	136.789	36.728	167.710
gus	usa13509.xo.15631	13509	15631	13509.000	15631.000	2530.176	1502.263
gh	usa13509.xo.15631	13509	15631	214.049	390.342	104.680	1052.889
ghs	usa13509.xo.15631	13509	15631	381.401	662.432	147.000	1063.709
ghg	usa13509.xo.15631	13509	15631	109.505	187.916	99.529	892.320
gus	usa13509.xo.17494	13509	17494	13509.000	17494.000	2936.825	1636.194
gh	usa13509.xo.17494	13509	17494	509.419	1011.787	295.418	1148.741
ghs	usa13509.xo.17494	13509	17494	1029.467	1907.037	340.760	1189.828
ghg	usa13509.xo.17494	13509	17494	316.181	571.854	440.779	1023.730
gus	d15112.xo.19057	15112	19057	15112.000	19057.000	3268.710	1935.979
gh	d15112.xo.19057	15112	19057	765.822	1491.199	405.040	1415.519
ghs	d15112.xo.19057	15112	19057	1470.403	2678.758	564.729	1465.350
ghg	d15112.xo.19057	15112	19057	512.190	915.079	991.442	1254.776

Table 3.22: Data for TSP family

TSP - cont'd						
	Name	N	M	Relabels	Pushes	TotalTime
gus	tsp.pr76.x.2	76	90	11266	14127	0.068
gh	tsp.pr76.x.2	76	90	3349	4188	0.034
ghs	tsp.pr76.x.2	76	90	4535	5709	0.042
ghg	tsp.pr76.x.2	76	90	3003	4422	0.026
gus	tsp.att532.x.1	532	787	1063351	1562818	4.990
gh	tsp.att532.x.1	532	787	113526	163446	0.972
ghs	tsp.att532.x.1	532	787	153862	209975	1.110
ghg	tsp.att532.x.1	532	787	41882	72700	0.778
gus	tsp.vm1084.x.1	1084	1252	4620731	6759490	22.722
gh	tsp.vm1084.x.1	1084	1252	692992	791288	5.760
ghs	tsp.vm1084.x.1	1084	1252	729703	837499	6.126
ghg	tsp.vm1084.x.1	1084	1252	585930	1019210	6.462
gus	tsp.d1291.x.1	1291	1942	8799141	13794299	45.244
gh	tsp.d1291.x.1	1291	1942	468269	722254	8.596
ghs	tsp.d1291.x.1	1291	1942	698900	1090299	9.396
ghg	tsp.d1291.x.1	1291	1942	354765	633638	7.236
gus	tsp.rl1323.x.1	1323	2169	11847414	18843579	60.212
gh	tsp.rl1323.x.1	1323	2169	1982525	3171867	15.410
ghs	tsp.rl1323.x.1	1323	2169	1674102	2568434	14.568
ghg	tsp.rl1323.x.1	1323	2169	1184215	2043948	11.884
gus	tsp.rl1323.x.2	1323	2195	13154067	21379535	66.032
gh	tsp.rl1323.x.2	1323	2195	1858710	2970391	14.936
ghs	tsp.rl1323.x.2	1323	2195	2587602	4283288	18.274
ghg	tsp.rl1323.x.2	1323	2195	1809131	3182704	14.888
gus	tsp.fl1400.x.1	1400	2231	5611206	7733388	35.336
gh	tsp.fl1400.x.1	1400	2231	882686	1215741	11.982
ghs	tsp.fl1400.x.1	1400	2231	1178439	1543050	13.072
ghg	tsp.fl1400.x.1	1400	2231	440222	722726	9.562
gus	tsp.vm1748.x.1	1748	2336	14378206	21863109	80.952
gh	tsp.vm1748.x.1	1748	2336	1690678	2613124	22.568
ghs	tsp.vm1748.x.1	1748	2336	2141324	3188744	24.974
ghg	tsp.vm1748.x.1	1748	2336	1292352	2156314	19.782
gus	tsp.rl5934.x.1	5934	7287	215146904	339530461	1132.362
gh	tsp.rl5934.x.1	5934	7287	6548048	9811932	220.642
ghs	tsp.rl5934.x.1	5934	7287	10326695	15343457	234.178
ghg	tsp.rl5934.x.1	5934	7287	3690344	6859845	186.579
gus	tsp.rl5934.x.2	5934	7627	323352112	539711410	1627.174
gh	tsp.rl5934.x.2	5934	7627	11642651	16912327	237.774
ghs	tsp.rl5934.x.2	5934	7627	21090031	33203500	271.778
ghg	tsp.rl5934.x.2	5934	7627	7753845	14153522	204.682
gus	usa13509.xo.15631	13509	15631	531347283	678321694	4033.327
gh	usa13509.xo.15631	13509	15631	30634681	39561542	1158.174
ghs	usa13509.xo.15631	13509	15631	45549472	53759023	1211.342
ghg	usa13509.xo.15631	13509	15631	20050422	34115253	992.468
gus	usa13509.xo.17494	13509	17494	549530523	709759847	4574.013
gh	usa13509.xo.17494	13509	17494	69743866	96070354	1444.972
ghs	usa13509.xo.17494	13509	17494	81149905	103254869	1531.428
ghg	usa13509.xo.17494	13509	17494	80970759	138601854	1465.180
gus	d15112.xo.19057	15112	19057	646836745	816539734	5205.931
gh	d15112.xo.19057	15112	19057	96301486	128562817	1821.206
ghs	d15112.xo.19057	15112	19057	130209542	170710980	2030.944
ghg	d15112.xo.19057	15112	19057	191025627	326933722	2247.079

Table 3.23: Data for TSP family - cont'd

WHE									
	N	M	Aver.N	Aver.M	CutTime	ManipTime	Relabels	Pushes	TotalTime
gus	64	126	64.000	126.000	0.090	0.020	20093	36092	0.110
gh	64	126	64.000	160.000	0.084	0.012	18083	30519	0.098
ghs	64	126	64.000	158.000	0.102	0.014	20186	38086	0.116
ghg	64	126	26.317	63.333	0.054	0.004	9851	16521	0.064
gus	128	254	128.000	254.000	0.612	0.052	132799	239193	0.676
gh	128	254	128.000	320.000	0.586	0.072	128092	219885	0.662
ghs	128	254	128.000	318.000	0.632	0.064	125855	251667	0.704
ghg	128	254	51.992	127.500	0.338	0.046	63320	111598	0.388
gus	256	510	256.000	510.000	4.330	0.228	826401	1622279	4.586
gh	256	510	256.000	640.000	3.866	0.342	803582	1429288	4.228
ghs	256	510	256.000	638.000	4.404	0.328	741497	1548958	4.744
ghg	256	510	102.863	254.655	1.982	0.240	367431	673441	2.238
gus	512	1022	512.000	1022.000	28.624	1.016	5062917	9804981	29.674
gh	512	1022	512.000	1280.000	28.906	1.302	5355418	10046681	30.228
ghs	512	1022	512.000	1278.000	27.272	1.346	4573090	10065115	28.636
ghg	512	1022	203.014	505.063	12.500	1.034	1989225	3994170	13.554
gus	1024	2046	1024.000	2046.000	205.388	6.568	32290667	68441212	212.014
gh	1024	2046	1024.000	2560.000	221.048	7.666	35834725	67668296	228.762
ghs	1024	2046	1024.000	2558.000	215.368	7.968	29847073	67803180	223.390
ghg	1024	2046	409.289	1020.676	80.816	5.912	11480901	23882469	86.784

Table 3.24: Data for WHE family

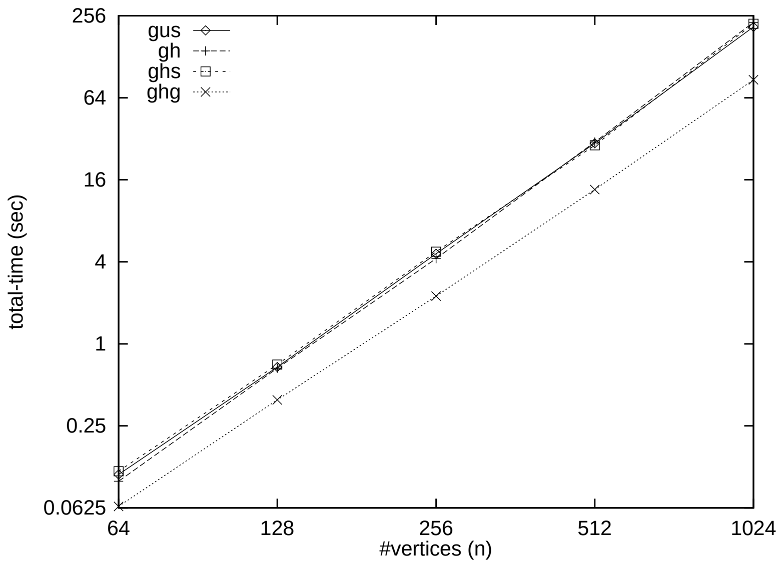

Figure 3.24: Running times for WHE family

Chapter 4

Maximum Flow Techniques for Graph Clustering

4.1 Basic minimum cut tree algorithm

4.1.1 Properties

The minimum cut tree T of a network G provides not only information about the minimum cuts within G, but also about its structure. The following two lemmata follow directly from the definition of a minimum cut tree:

Lemma 4.1.1 *Let u, v be two nodes of the minimum cut tree T, and let $P(u, v)$ be the (unique) path that connects them in T. Then, for any node w on that path, the maximum flow $f(u, w)$ between u and w is no less than the maximum flow $f(u, v)$ between u and v. Symmetrically, $f(w, v) \geq f(u, v)$.*

Lemma 4.1.2 *For u, v two nodes and $P(u, v)$ the path between them in the minimum cut tree T, the length $p = |P(u, v)|$ of that path is equal to the number of unique (i.e. different) minimum s, t-cuts in the graph.*

The minimum cut tree tends to put together nodes that are relatively heavily connected in the graph. But the fact that it tries to do so for all nodes can be conflicting between the neighborhoods created and may result in minimum cut trees of poor overall structure (e.g. a circular unit-capacity graph has an exponential in its size number of minimum cut trees, and there are at least two trees very different from each other).

But often the minimum cut tree does follow the structure of the initial graph close enough, and the algorithm of this section exploits this characteristic. We describe a basic clustering algorithm that clusters according to the structure of the minimum cut tree, and provide examples for queries from the WWW, which justify the potential of the algorithm to produce clusters of high quality. In later sections we shall improve upon this algorithm, by imposing stronger clustering criteria and produce clusters of high quality more consistently.

4.1.2 The algorithm

Let $G(V, E)$ be the graph to be clustered, and T its minimum cut tree. The algorithm starts off with identifying the centroid v_c of T. The centroid of a tree T with nodes V is defined as follows: Let $v \in V$ and $T_1, T_2, ..., T_d$ be the connected components induced by removing v from T. If $|T_i|$ denotes the size of subtree T_i, define for each node the following measure:

$$N(v) = \max_{1 \leq i \leq d}\{|T_i|\} \tag{4.1}$$

Then, a centroid of tree T is defined ([50]) as the node v_c, which minimizes $N(v)$ over all nodes in V, or

$$N(v_c) = \min_{v \in V}\{N(v)\} \tag{4.2}$$

It is easy to show that T may have up to two centroids, in which case we pick one arbitrarily.

Once the algorithm has located the centroid v_c, let T be rooted at v_c. The subtrees under v_c partition G and correspond to the clusters returned by the algorithm. The only node that gets not clustered this way is the centroid itself. Let e_c be the adjacent edge of v_c with maximum weight. We assign v_c to the cluster which corresponds to the tree under e_c. Figure 4.1 shows a minimum cut tree and the clusters given by the algorithm.

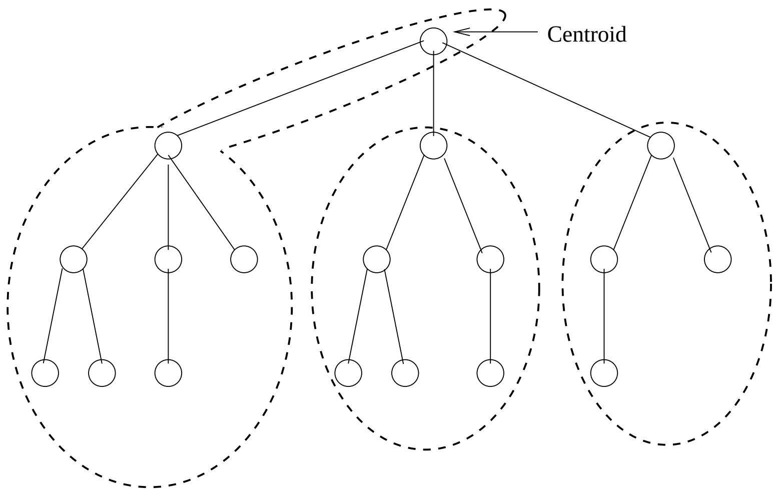

Figure 4.1: Clusters of basic clustering algorithm

The algorithm also allows for hierarchical clustering. Let the clusters produced by the above procedure be the clusters of highest (i.e. first) level. We recursively apply the same algorithm to each of the clusters in order to get clusters of lower levels. So, for the second iteration, let the nodes adjacent to the initial v_c be the new roots (notice that we don't recalculate new centroids for the subtrees), and let the subtrees of these roots be the clusters of second level. Figure 4.2 shows the clusters for the top two levels (roots omitted).

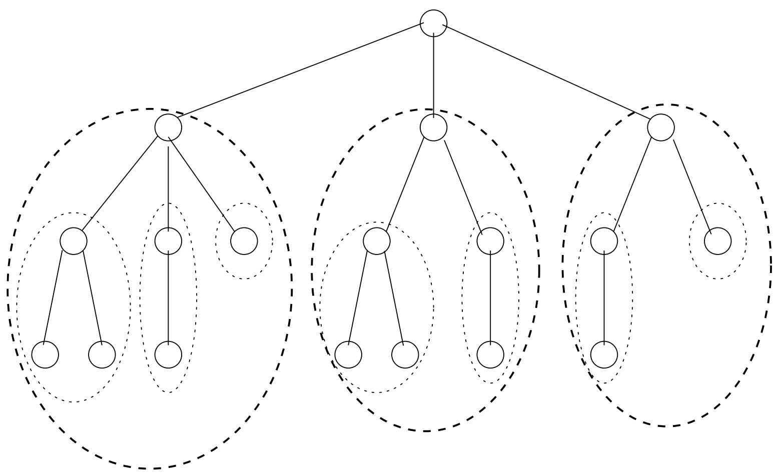

Figure 4.2: Clusters of Basic Clustering Algorithm

4.1.3 Edge normalization

The minimum cut tree, as defined by Gomory and Hu, applies only to undirected graphs. In fact, Benczúr [7] showed that no minimum cut tree can be defined for directed graphs. On the other hand, many of the data-sets we experimented with are directed by nature, e.g. web-pages and the links among them, scientific literature and their references, etc. In order to transform these directed graphs into undirected, we normalize over the outgoing edges.

Let $G(V, E)$ be a directed graph, and $v \in V$ one of its nodes. Let $E_{in}, E_{out} \subseteq E$ be respectively the set of edges pointing inwards to and outwards from v. For an edge $(x, y) \in E$, let $w(x, y)$ be its weight. We normalize the outgoing edges from v as follows: Let $(u, v) \in E_{out}$. Make (u, v) undirected and set its new weight $w'(u, v)$ according to:

$$w'(u,v) = \begin{cases} \frac{w(u,v)}{\sum_{x \in V} w(u,x)} & \text{if } |E_{out}| > 0; \\ 0 & \text{if } |E_{out}| = 0. \end{cases}$$

Every node normalizes the weights of its outgoing edges only, and since every edge is outgoing for some node, applying the above procedure to every node and all its adjacent outgoing edges transforms G into an undirected graph with all its edges normalized.

This normalization technique loses some information, but is not arbitrary. A similar approach is being used in *pagerank* [11], and also corresponds to the first iteration of Kleinberg's *hubs and authorities* algorithm [62]. One improvement over normalizing only over outgoing edges is to repeat the same normalization for the incoming edges, and possibly iterate in both directions several times until the edge-weights are close enough to their fixed point. (Which is also what Kleinberg does for hubs and authorities.) We have tried this approach as well, but report only on results for a single iteration (over outgoing links), since they were satisfying enough.

4.1.4 Data from the WWW

For the experiments on the basic clustering algorithm, we start off by choosing a search term and use one of the existing search engines (`alltheweb.com`, `altavista.com`, `google.com`, etc.) to get an initial seed set of web-pages relevant to the search term. Then, to create our initial graph of web-pages, we look at all the in- and out-going links of that seed-set. The nodes of the graph correspond to the web-pages (both the initial seed-set and those at distance 1), and the edges correspond to the links between the web-pages.

Given this graph, we first normalize it and then apply the algorithm of Subsection 4.1.2 to compute its clusters.

Authorities

Next, we report on results for four search terms, which are the exact same ones used in [62]. Also, we follow the same procedure in obtaining the seed set and expanding it one link away. This allows for a direct comparison of our results to those from [62]. One difference is in ranking the results we get. Since our algorithm produces hierarchical clusters of nodes, instead of a linear ranking function, we assign to each node a rank value which depends on the distance (i.e. number of links) between that node and the centroid. To compare two nodes not in the same cluster, we look at the underlying subtrees in their clusters respectively, and sort according to their sizes.

The four search terms are: `java`, `censorship`, `search engine`, and `gates`. In all cases we get our initial seed-set from `altavista.com` and truncate it down to the top 200 results. Then we look at all nodes within distance 1 away from that seed-set. This will be in each case our initial graph, and this is exactly identical to what Kleinberg did for his experiments [62].

Also, next to each web-page, there are two numbers, indicating the size of the subtree rooted at that node (i.e. the size of the cluster for which this node is the centroid/root), and how strongly connected it is to the rest of the graph (i.e. the min-cut value). The larger the min-cut value, the more connected that cluster is to the rest of the graph. The smaller the min-cut value, the more isolated that cluster.

SIZE: 20987 – TOP RESULTS FOR **java**:

1. `http://www.javaboutique.com/` (2425, 2615)
   Java applets, downloads,...
2. `http://www.gamelan.com/` (676, 1720)
   Gamelan portal on java
3. `http://www.devworld.apple.com/java/` (346, 3033)
   Developer - java
4. `http://www.davecentral.com/java.html` (593, 1341)
   Java software, freeware,...
5. `http://www.mindprod.com/gloss.html` (310, 2954)
   Java and internet glossary

Figure 4.3: Results for search term 'java'

For the search term `java` (Figure 4.3), we see that our results are similar to

Kleinberg's. Entries in positions 2 and 3 are expected, since they are among the most authoritative ones. Our algorithm differs from Kleinberg's in entries 1, 4, and 5, where all three web-pages correspond to large authoritative directories on java-programming and java-applications.

SIZE: 4787 – TOP RESULTS FOR **censorship**:
1. `http://www.eff.org` (528, 2087)
   Electronic Frontier Foundation
2. `http://www.ncac.org/index.html` (265, 2827)
   National Coalition Against Censorship
3. `http://www.canada.ifex.org/` (769, 1812)
   International Freedom of Expression eXchange
4. `http://www.projectcensored.org/intro.htm` (237, 957)
   Project Censored
5. `http://www.booksatoz.com/censorship/index.htm` (55, 2212)
   Information on Censorship of the Written Word

Figure 4.4: Results for search term 'censorship'

For the term `censorship` (Figure 4.4), all five webpages are very strong and closely related to the search term. This community is much smaller (over 4 times) than the java community above, and there are fewer, thus stronger, authorities.

SIZE: 10722 – TOP RESULTS FOR **search engine**:
1. `http://www.yahoo.com/` (10722, 6130)
2. `http://searchenginewatch.com/` (922, 6130)
3. `http://www.beaucoup.com/` (626, 4688)
4. `http://www.netstrider.com/search/` (357, 2647)
5. `http://www.google.com` (6, 11019)

Figure 4.5: Results for search term 'search engine'

For search term `search engine`, Kleinberg's top 5 consisted of the most popular search engines. We also get two search engines in the top 5, but looking at the top 10-20 results, we see that they are dominated by directories about search engines, and meta-search engines about search engines, including those that support multiple, simultaneous searches (from several sources) at once. Individual search engines did not rank as high, compared to these web-pages. Depending on the criteria, both top results make sense. Our results seem more general, Kleinberg's seem more focused, but again, this is very subjective.

SIZE: 5723 – TOP RESULTS FOR `gates`:
1. `http://www.microsoft.com/billgates/` (874, 1657)
   Bill Gates' Web Site
2. `http://www.gatesfoundation.org/` (866, 760)
   B. and M. Gates Foundation
3. `http://www.quuxuum.org/~evan/bgnw.html` (241, 2195)
   Gates Net Worth
4. `http://www.zpub.com/un/bill/` (121, 2414)
   The Unofficial B. Gates
5. `http://gates.theinfo.org` (103, 2044)
   Gates Dollars

Figure 4.6: Results for search term 'gates'

The search term `gates` is very broad, so the results we get cover a wide range of different topics. It is interesting though that all top results are about Bill Gates, and that there are only a couple of strong authorities in this community: his home-page and the Gates Foundation.

Tree-structure

The next example shows not only the centroids/roots at the highest level, but for the top three levels. It is an illustrative case, where the simple minimum cut tree produced a high quality multi-level clustering, indicating that the minimum cut tree can follow the structure of a graph in a very natural way.

Figure 4.7 shows the results for the search term `jaguar`. Again, we start off with a seed set of size 200 from `altavista.com`, which we expand by following one link away, then normalize, and find the clusters from the minimum cut tree, as described in Section 4.1.2.

Jaguar* (3129)
- **Automobile** `http://www.carlynx.com/mak/jag.htm` (1887)
  - **Jaguar Cars Official Home Page** `http://www.jaguarcars.com/` (588)
  - **Jaguar Racing** `http://www.jaguar-racing.com/` (370)
  - **US Jaguar Sites** `http://www.jaguarvehicles.com/us/` (166)
  - **Jaguar Clubs and Events** `http://www.jagweb.com/` (143)
  - **International Jaguar Sites** `http://autopedia.com/html/MfgSitesJagua.html` (78)
  - **Jaguar Parts** (22)
  - **Jaguar Magazines** (12)
- **NFL Football Team** `http://cnnsi.com/football/nfl/teams/jaguars/index.html` (322)
  - **Home page** `http://jaguars.jacksonville.com/` (100)
  - **NFL page on Jaguars** `http://www.nfl.com/jaguars/` (95)
- **Atari Jaguar machine** `http://www.angelfire.com/nv/jaguartop50/` (121)
  - **Jaguar portal** `http://jaguar.holyoak.com/` (70)
- **Other clusters - more isolated or smaller**
  - **Webdesign company** `http://www.jaguarwoman.com/` (277)
  - **Drug company** `http://www.schrodinger.com/` (261)
  - **Diving company** `http://www.divejaguarreef.com/` (57)
  - **Starship and military models** (42)
  - **Animal Jaguar** `http://www.bluelion.org/jaguar.htm` (15)

Figure 4.7: Example for the basic minimum cut tree algorithm

We conclude from the examples of this section that the subtrees of the minimum cut tree, with respect to the centroid, often correspond to high quality clusters. In the next sections we will continue to use the minimum cut tree as the basic underlying structure, but the clusters will be calculated by different, stronger criteria, that will guarantee for higher quality clusters.

4.2 Minimum k-cut clustering algorithm

4.2.1 Properties

The minimum k-cut clustering problem is that of dividing a network into k disjoint sets, s.t. the total sum of edges between the sets is minimum. This problem, though NP-complete, has a $(2-\frac{2}{k})$-approximation that makes use of a minimum cut tree [74]:

Lemma 4.2.1 *Let T be the minimum cut tree for an undirected network $G(V, E)$. Removing the k-1 edges with minimum weight in T yields k connected components, which partition G into k clusters, s.t. the sum of the edges between these clusters are a $(2 - \frac{2}{k})$ approximation to the minimum k-cut clustering problem.*

Wu and Leahy [78] apply the same methodology to image segmentation. In their paper they show that the clusters produced have the following property: Let v_1, v_2 be two nodes from the same cluster, and let w_1, w_2 be two nodes from different clusters. Then the maximum flow between v_1 and v_2 is always at least as great as the maximum flow between w_1 and w_2 in the original graph. In other words,

Lemma 4.2.2 *All intra-clusters cuts (i.e. cuts between nodes of the same cluster) are at least as large as any inter-cluster cuts (i.e. between nodes of different clusters) in G.*

4.2.2 Performance

The reason we mention the above algorithm is that it is closely related to our topic of clustering a graph by means of a minimum cut tree. We will not analyze further the exact properties of this algorithm, but it is worthwhile to mention its advantages and disadvantages, as we recorded them.

Advantages:

Lemma 4.2.2 provides a simple way to partition the nodes of G into clusters, so that nodes with large maximum flows between them get clustered together. This is a natural way to classify the nodes according to the maximum flows of the graph, and it has been shown to perform well in practice for image segmentation [78].

Disadvantages:

We have observed that the minimum cut tree tends to place heavier edges towards the center and lighter edges at the perimeter. In fact, for most of the graphs we experimented with, the lightest edges were between leaves (external nodes) and the rest of the graph. Thus, a clustering algorithm based on removing these edges would result in highly unbalanced clusters, mostly singletons. This is exactly what we have experienced for our data-sets.

4.3 Community clustering algorithm

In this section we focus on a clustering algorithm that is based on the following definition of a community:

Definition 4.3.1 *An* esoteric community *is a set S, such that* $\forall s \in S$,

$$\sum_{v \in S} w(s,v) \geq \sum_{v \in V-S} w(s,v) \tag{4.3}$$

That is, all community members predominantly link to other community members.

Our goal is to find a clustering that is non-trivial and covers the entire graph, that is, a partitioning of the graph into esoteric communities.

Before presenting the algorithm, we will show that this problem is in fact NP-complete. Yet, for many instances the minimum cut tree (on which our algorithm is based) provides a simple tool for extracting such communities, and we will study under which conditions it succeeds to cluster the graph and under which it doesn't.

4.3.1 Esoteric community

Let $G(V, E)$ be a weighted, undirected graph with n nodes and m edges. Let $p \geq 1/2$ be a *connectedness* parameter that defines an *esoteric* community $S \subset V$ in the following way:

Definition 4.3.2 *S is a p-esoteric community if $\forall s \in S$,*

$$\sum_{v \in S} w(s, v) \geq p \sum_{v \in V} w(s, v) \tag{4.4}$$

or equivalently,

$$(1-p) \sum_{v \in S} w(s, v) \geq p \sum_{v \in V-S} w(s, v) \tag{4.5}$$

That is, the sum of the weights of the edges that connect s with $S - s$ is at least a fraction p of the total weight of all adjacent edges of s. Also, when we refer to an esoteric community without specifying p, we imply that $p = 1/2$, so that

$$\sum_{v \in S} w(s, v) \geq \sum_{v \in V-S} w(s, v) \tag{4.6}$$

as defined earlier.

The notion of an esoteric community has been used before in a similar way for graph clustering ([25]).

4.3.2 NP-completeness of esoteric community problem

For graph G, we define the following partitioning problem:

PROBLEM: **PARTITION INTO ESOTERIC COMMUNITIES**

INSTANCE: Undirected graph $G(V, E)$, real parameter $p \geq 1/2$, integer k.

QUESTION: Can the vertices of G be partitioned into k disjoint sets $V_1, V_2, ..., V_k$, such that for $1 \leq i \leq k$, the subgraph of G induced by V_i is an esoteric community?

Lemma 4.3.1 *PARTITION INTO ESOTERIC COMMUNITIES, or simply ESOTERIC COMMUNITIES, is NP-complete.*

We will prove the above lemma by reducing BALANCED PARTITION, a restricted version of PARTITION, to ESOTERIC COMMUNITIES.

Here are the definitions for PARTITION (from [33]) and BALANCED PARTITION.

PROBLEM: **PARTITION**

INSTANCE: A finite set A and a *size* $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

QUESTION: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

PROBLEM: **BALANCED PARTITION**

INSTANCE: A finite set A and a *size* $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

QUESTION: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$, and $|A'| = |A|/2$?

Both PARTITION and BALANCED PARTITION are well known NP-complete problems ([60] and [33]). Now we can prove Lemma 4.3.1:

Proof. We will reduce BALANCED PARTITION to ESOTERIC COMMUNITIES.

First of all, it is easy to see that if we are given a solution C of ESOTERIC COMMUNITIES, we can verify in polynomial time weather each node is connected to nodes of the same cluster by at least a fraction p of its total adjacent edge-weight.

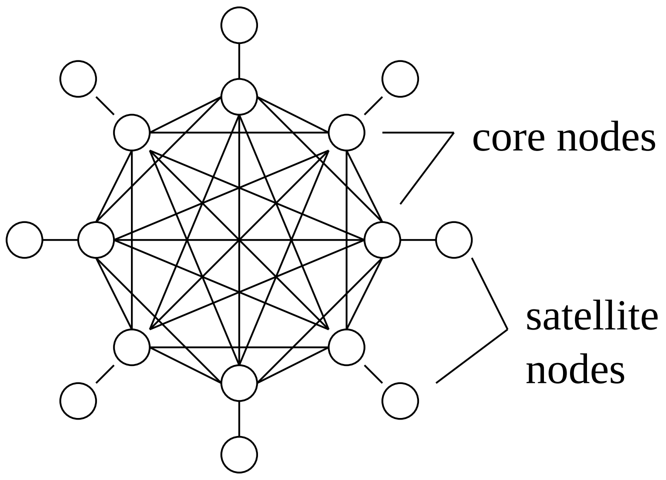

Figure 4.8: Core and satellite nodes forming the core graph

For the proof, we will transform the input of BALANCED PARTITION to that of ESOTERIC COMMUNITIES. The input set C has cardinality $n = 2k$. We will construct an undirected graph G with $2n + 4$ nodes as follows. First, n of G's nodes will form a complete graph K_n with all edge-weights equal to w, s.t. $\frac{s(a_{min})}{n} > w > 0$, where a_{min} is the smallest among all elements a_i of A. Lets call these nodes the *core* of the graph. Then, we connect each node of the core to a single *satellite* node respectively, with edge-weight $w + \epsilon$, s.t. $w > 2\epsilon > 0$. The n satellite nodes have all degree 1 (Figure 4.8).

Now, we add two more nodes to the graph, which we call *basic*. Each basic node is connected to every core node by an edge of weight $s(a_i)$. We make sure that all $s(a_i)$'s of A are used as weights by the adjacent edges of the basic nodes, but also that every core node is connected to both basic nodes by the exact same weight $s(a_i)$. Finally, we add two more satellite nodes, one to each basic node, by edges of weight

ϵ. The final graph looks as in Figure 4.9.

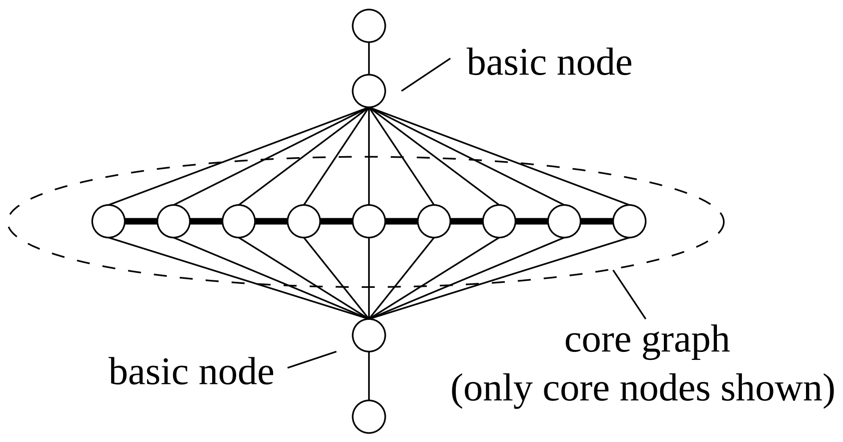

Figure 4.9: Final graph for esoteric community proof

Now, lets assume that ESOTERIC COMMUNITY is solvable in polynomial time. We transform an input of BALANCED PARTITION as mentioned above, and set $p = 1/2$ and $k = 2$.

Assume that ESOTERIC COMMUNITY gives a solution for this instance. Then, that solution must divide the core nodes into two sets, S_1 and S_2, for if this were not true, one of the two sets, say S_1 must consist only of basic and/or satellite nodes. But every satellite node requires its adjacent node to be included as well in order to form an esoteric community, and thus there must be at least one basic node in S_1. Including a basic node in S_1 also requires including at least one core node, because otherwise that basic node cannot be connected heavily enough within S_1. So, the partitioning must split the core nodes into two sets.

Also, the partitioning cannot put both basic nodes in the same set, because then a core node in the other set will be connected to at most $n-2$ core nodes (and its satellite node), thus having adjacent edge-weight of at most $(n-1)w + \epsilon$ within its community. But its total adjacent weight is $nw + \epsilon + 2s(a_i) > 2((n-1)w + \epsilon)$. So, each partitioned set must contain exactly one basic node.

Now we can prove that the core nodes can actually only be split evenly. To see this, assume w.l.o.g. that S_1 contains $q < n/2$ core nodes. Then the adjacent weight of each of those core nodes is $qw + \epsilon + x_i$. But its total adjacent weight is $nw + \epsilon + 2s(a_i) > 2(qw + \epsilon + s(a_i))$. So, S_1 and S_2 must contain at least, thus exactly, $n/2$ nodes each. When that is the case, we can verify that the adjacent weight for each core node sums up.

Satellite nodes will always be in the same set as their adjacent nodes, so the only nodes left to consider are the basic nodes. Is it possible that their adjacent weight is less within their community? The answer depends on the way the core nodes get divided. If the core nodes are divided in such a way that the sum of their adjacent a_i values is equal for both sets, then it easy to verify that the basic nodes don't violate the esoteric community property. But in the opposite case, one of the two basic nodes will be connected heavier to the nodes of the other community than to the nodes of its own community. Thus, the only possible solution for ESOTERIC COMMUNITY must partition all a_i's into two sets of equal sum and cardinality. But this implies that any instance for the BALANCED PARTITION problem can be solved by ESOTERIC COMMUNITY. And since BALANCED PARTITION is NP-complete and we can build the graph G and transform the one instance to the other in polynomial time, ESOTERIC COMMUNITY is NP-complete too. ■

4.3.3 Structure of esoteric communities

Here, we mention some lemmata that provide a good intuition about the structure of esoteric communities:

Lemma 4.3.2 *If G is connected and C is a clustering of esoteric communities that covers G, then each esoteric community must contain at least two nodes of G.*

Proof. Immediate, since no singleton can form an esoteric community by itself in a connected graph. ■

Lemma 4.3.3 *If C is a clustering that covers G, then C' consisting of unions of communities of C is also a valid clustering of G.*

Proof. For each node in a community, the addition of more nodes to that community can only increase the weight of the adjacent edges within the community. ■

Lemma 4.3.4 *Let C be a minimal clustering that covers G. That is, the communities of C cannot be split into smaller sub-communities and still maintain the esoteric community invariant. Then there may exist another minimal clustering C' that covers G, yet is different than C. That is, there might not exist a unique minimum clustering for a graph, but more than one different minimal clusterings.*

Proof. Figure 4.10 depicts a graph and two of its clusterings that are minimal, yet different. In this example $p = 1/2$. ■

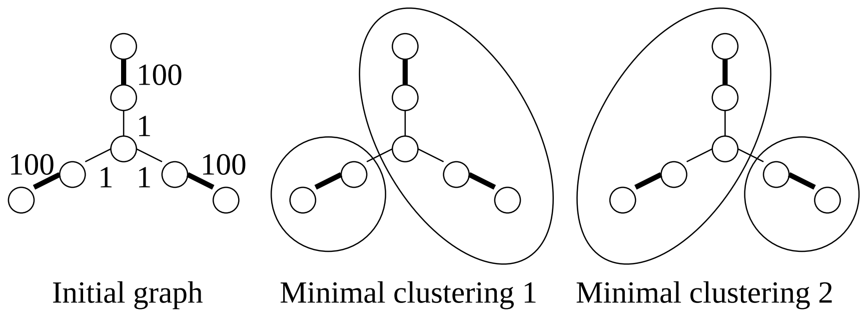

Figure 4.10: Two different minimal clusterings

So, by combining communities of lower levels we can create larger, more general ones, and thus build a hierarchy of communities of G. Also, there can be multiple different hierarchies, based on different minimal clusterings, at the lowest level.

Next we will present an algorithm that is based on minimum cut trees and finds esoteric communities in a graph.

4.3.4 Finding esoteric communities

In the rest of this section we will assume that $p = 1/2$.

Let s, t be any source and sink for graph G, and consider the minimum s,t-cut in G. Let S, T be the two sides of the cut, s.t. $s \in S$ and $t \in T$. If we move any node from $S - s$ to the other side of the cut, the cut-value can only increase, and the same holds for every node in $T - t$. Thus every node in S and T is heavier connected to the other nodes in its side of the cut, and thus S and T obey the esoteric community properties, for $p = 1/2$. Only exception might be the nodes s and t themselves. If by moving s or t to the other side of the cut, the value of the cut increases, then S and T are both esoteric communities. It follows directly that the global minimum cut of G always produces esoteric communities, unless of course one of the two sides is a singleton.

Based on the above observations we can find a minimum cut in G that gives two esoteric communities by using a minimum cut tree. We introduce *local minimum edges* in the minimum cut tree as follows:

Definition 4.3.3 *An edge e of a minimum cut tree T is a* local minimum edge*, if both ends v_1, v_2 of e have at least one adjacent edge in T with larger weight than that of e.*

Equivalently, let T be a minimum cut tree, and mark for every node its adjacent edge of largest weight. All unmarked edges are *local minimum edges.*

We show the following lemma:

Lemma 4.3.5 *Let T be a minimum cut tree for graph G, and let e be a local minimum edge in e. Then, the removal of e yields two esoteric communities.*

Proof. Let v_1, v_2 be the nodes of T adjacent to e. Edge e corresponds to the (v_1, v_2)-minimum cut in G. Thus, the two sides of the cut are esoteric communities if (a) neither of them contains only a singleton and (b) if moving v_1 or v_2 to the other side doesn't yield a cut of smaller value.

To show (a) assume that v_1 is the only node in its community. But then e would be marked as the heaviest (since unique) adjacent edge of v_1.

For (b), assume w.l.o.g. that moving v_1 to the v_2 side of the cut yields a cut of smaller value. Let e_{max} be the adjacent edge of v_1 with largest weight, and let v_{max} and v_1 be the two nodes adjacent to e_{max}. ($e_{max} \neq e$, for otherwise e would have been marked.) But moving v_1 to the other side of the cut creates a new cut that also separates v_{max} from v_1 and thus can't have weight less than that of e_{max}, which is larger than the weight of e.

Thus the esoteric communities created by removing e are valid. ■

Based on the previous lemma we develop an algorithm that finds esoteric communities in G. The idea is to first find two communities and contract one of them into a single node, say X. Subsequent iterations produce again two communities, and each time one of these communities gets contracted into X. The algorithm finishes when no local minimum edge can be found in the current minimum cut tree. Figure 4.11 contains the pseudocode.

Lemma 4.3.6 *The algorithm of Figure 4.11 produces valid $\frac{1}{2}$-esoteric communities.*

Proof. In each iteration, the cluster output corresponds to one of the two sides of a minimum cut, induced by the removal of a local minimum edge. Lemma 4.3.5 applies directly and guarantees for the validity of the esoteric clusters. ■

```
CommunityCluster($G(V, E)$)
    Let $T$ be the minimum cut tree of $G$
    If ($T$ has no local minimum edge)
        Output single, trivial cluster covering entire $G$ and return
    Else
        Let $X = \{\}$ be empty
        Let $e$ be a local minimum edge in $T$
        While ($e$ non-empty)
            Let $S_1, S_2$ be the two communities formed after removing $e$
            If ($X == \{\}$)
                Either output $S_1$ as next community and contract it into $X$
                    or output $S_2$ as next community and contract it into $X$
            Else if ($X \in S_1$)
                    Output $S_2$ as next community and contract it into $X$
            Else /* $X \in S_2$ */
                    Output $S_1$ as next community and contract it into $X$
            Calculate new minimum cut tree $T$
            Let $e$ be a local minimum edge in new $T$
        Return
```

Figure 4.11: Community clustering algorithm

Note that Lemma 4.3.5 allows for the removal of only one local minimum edge at a time. This is also the reason for outputting only one community in each iteration. If we delete multiple local minimum edges simultaneously, the resulting connected components are not necessarily esoteric communities. The reason we contract all output communities into X is to avoid crossing cuts that might cause the same node to be part of more than one communities. Also, note that the algorithm doesn't necessarily partition the entire graph; it only extracts esoteric communities from G.

4.3.5 Relaxing the conditions

Instead of iterating over newly found local minimum edges, as described in the previous algorithm, one idea is to remove all local minimum edges of the first minimum cut tree at once. Of course, as mentioned above, the clusters produced that way might not be esoteric communities, but we can still prove the following lemma:

Lemma 4.3.7 *Let T be the minimum cut tree of graph G, and let C be the clustering produced by removing all local minimum edges of T. Let v_1, v_2 be two nodes in the same cluster of C. Let $c(v_1), c(v_2)$ be the largest minimum cut values separating respectively v_1 and v_2 from any other node in G. Then, the for the minimum cut value $c(v_1, v_2)$ between v_1 and v_2 we have that*

$$c(v_1, v_2) \geq \min c(v_1), c(v_2) \tag{4.7}$$

Proof. Every cluster formed by removing the local minimum edges of T is connected in T. Thus, for any v_1, v_2 in the same cluster, say C_1, the nodes between v_1 and v_2 on the path $P = P(v_1, v_2)$ that connects them in T also belong to C_1.

Looking at the weights of the edges of P, we can see that there must exist a node, say $v \in P$, s.t. the edges from both v_1 and v_2 to v have non-decreasing weight, for otherwise some edge on the path is a local minimum edge and should have been removed. But v is different than either v_1 or v_2 or both. Assume w.l.o.g. that $v_1 \neq v$. Then the node adjacent to v_1, say v_3, didn't mark the edge (v_1, v_3) (because the edge-weights increase monotonically towards v), and thus it was marked by v_1, which means that the weight of edge (v_1, v_3) is equal to $c(v_1)$, the largest minimum cut value separating v_1 from all other nodes in G. But it is also an upper bound on the minimum cut value between v_1 and v_2, which proves the lemma. ■

We have applied the algorithm on several of the problem families of Section 3.2 and concluded that it produces clusters of high quality in most cases. We will elaborate in detail on experimental results involving esoteric communities in Chapter 5.

4.4 Cut-clustering algorithm

In this section we introduce a simple clustering method for undirected graphs. The clustering method uses maximum flow techniques on the link-structure of the graph. The quality of the produced clusters is bounded by strong minimum-cut and expansion criteria. We also present a framework for hierarchical clustering and apply it to real-world data. We conclude that the clustering algorithms satisfy strong theoretical criteria and perform well in practice.

4.4.1 Introduction

As mentioned above, in this section we present a new clustering algorithm which is based on maximum flow techniques. Maximum flow algorithms are relatively fast and simple, and have been used in the past for data-clustering (e.g. [78], [25]). Difficulties in the analysis and in practice have also been pointed out in [59]. The main idea behind maximum flow (or equivalently, minimum cut [29]) techniques is to create clusters that have small inter-cluster cuts (i.e. between clusters) and relatively large intra-cluster cuts (i.e. within clusters). This guarantees strong connectedness within the clusters and is also a strong criterion for a good clustering in general.

Our clustering algorithm is based on inserting an artificial sink to a network and connecting it to all nodes of the network. Maximum flows are then calculated between all nodes of the network and the artificial sink. A similar approach has been introduced in [25], which was also the motivation for our work. Here we analyze the minimum cuts produced, calculate the quality of the clusters produced in terms of expansion-like criteria, generalize the clustering algorithm into a multi-layered clustering technique, and apply it to real-world data.

Structure-wise, this section consists of five subsections, besides the Introduction.

In Subsection 4.4.2, we review basic notions necessary for the results and sections that follow. In Subsection 4.4.3, we focus on the previous work of Flake et.al. ([25]) and lay the foundations for our cut-clustering algorithm by inserting an artificial sink to the network, as mentioned before, and studying the minimum-cut properties induced. In Subsection 4.4.4, we generalize the method of Subsection 4.4.3 by defining a recursive clustering method on multiple levels. In Subsection 4.4.5, we present results of our method applied to a real world problem set and see how the algorithm performs in practice. We end with Subsection 4.4.6, which contains a summary of our results and final remarks.

4.4.2 Basic Notions and Terminology

Max-flows and the minimum-cut tree

Let $G(V, E)$ be an undirected network with $|V| = n$ nodes and $|E| = m$ edges. The edges are weighted, and each edge $e \in E$ has weight $w(e)$.

Let $s, t \in V$ be two nodes designated as source and sink respectively. We define a community of s in terms of a minimum cut between s and t. Specifically, we define the community of s in G with respect to t to be the set $S \subset V$, s.t. $s \in S$, $t \notin S$ and the cut between S and $V - S$ is minimum. We say that the cut $(S, V - S)$ has value $c(S, V - S)$. In the case of a tie between more than one communities with the same minimum cut value, we pick the one of smallest size. In that case, we know that the smallest community is unique ([45]). We can avoid ties in the first place by slightly changing the graph, but not affecting the minimum cuts (by changing the edge-weights very little).

Our algorithm is based on minimum-cut trees, which were defined in [45]. For $G(V, E)$, we define T_G, or simply T, to be its minimum-cut tree (or simply, min-cut tree). The min-cut tree is defined over V and has the property that we can find the minimum cut between two nodes s, t in G by inspecting the path that connects s and t in T. The edge of minimum capacity on that path corresponds to that minimum cut. The capacity of the edge is equal to the minimum cut value, and the removal of it yields two sets of nodes in T but also in G, which correspond to the two sides of the cut. For every undirected graph, there always exists a min-cut tree.

Thus, the min-cut tree provides an easy way to find the community S for any s with respect to some t. We simply need to find the edge of minimum capacity (and closest to s), on the path that connects s and t, and after removing it the community S will be the side of the cut where s lies in.

Expansion and conductance

Let $(S, \bar{S})$ be a cut in G. We define the expansion of this cut to be

$$\psi(S) = \frac{\sum_{i \in S, j \in \bar{S}} w_{ij}}{\min\{|S|, |\bar{S}|\}},$$

where $w_{i,j}$ is the weight of edge (i, j). The expansion of a (sub)graph is the minimum expansion over all the cuts of the (sub)graph. The quality of a clustering can be measured in terms of expansion: The expansion of a clustering of G is the minimum expansion over all its clusters. The bigger the expansion of the clustering the higher its quality.

Similarly to expansion, we can also define conductance. For the cut $(S, \bar{S})$ in G, conductance is defined as

$$\phi(S) = \frac{\sum_{i \in S, j \in \bar{S}} w_{ij}}{\min\{w(S), w(\bar{S})\}},$$

where $w(S) = w(S,V) = \sum_{i \in S} \sum_{j \in V} w_{ij}$. The conductance of a graph is the minimum conductance over all the cuts of the graph. For a clustering of G, let $C \subseteq V$ be a cluster and $(S, C \backslash S)$ a cluster within C, where $S \subseteq C$. The conductance of S in C is

$$\phi(S,C) = \frac{\sum_{i \in S, j \in C \backslash S} w_{ij}}{\min\{w(S), w(C \backslash S)\}}.$$

The conductance of a cluster $\phi(C)$ is the smallest conductance of a cut within the cluster. For a clustering, the conductance is the minimum conductance of its clusters.

Both expansion and conductance seem to give very good measures of quality for clusterings and are claimed in [59] to be generally better than simple minimum-cuts. We agree with this claim, as it was true for the majority of the data we have experimented with, including that presented in Subsection 4.4.5. The main difference between expansion and conductance is that expansion treats all nodes as equally important and conductance gives greater importance to nodes that have many similar neighbors, where 'similar' is given in terms of edge-weights: the bigger the edge-weight the more 'similar' the nodes.

But there are also difficulties in using either expansion or conductance for clustering applications. Both are computationally hard, usually raising problems that are NP-hard, since they have an immediate relation to the 'sparsest cut' problem. Hence, approximations must be employed. But there are also problems relevant to the graph-structure that worsen their performance, e.g. isolated nodes, very sparse regions, etc.

Kannan et.al. [59] have addressed many of these issues. (We have borrowed the above definitions of expansion and conductance from their work.) They propose a bicriteria optimization problem that requires:

a) the clusters to have some minimum conductance (or expansion) α, and

b) the sum of the edge-weights between clusters to not exceed some maximum fraction ϵ of the sum of the weights of all edges in G.

The algorithm that we are going to present has a similar bicriterion and is based on expansion only. In our clustering algorithm we don't differentiate among nodes according to their neighborhoods.

4.4.3 Artificial Sink and Cut-Clustering Algorithm

Addition of an artificial sink

In the first section we defined a community S, containing source s, with respect to sink t. But sometimes we wish to find a community in G to include a source s (or a set of source-nodes) when no sink-node is provided. In this case, either the definition of S has to be changed, so as to include a 'similarity' measure between its nodes that separates S from the rest of the graph, or an artificial t has to be defined. One idea in this direction is to add a new node, which we call an 'artificial sink', and connect it to all nodes of the graph. This idea has been used before in [25], and this was also the motivation for our work. There, the definition of a community is slightly different and the bounds not as well defined. In particular, a community is a set S, such that $\forall s \in S$, $\sum_{v \in V-S} w(s,v) \leq \sum_{v \in S} w(s,v)$, that is, all community members predominantly link to other community members. Our definition of a community covers this definition.

Also, in [25] the artificial sink is not connected to all nodes, but only to the outermost nodes of a community already found under certain connectivity criteria. And the weight of the edges that connect the artificial sink to the graph is constant, equal to the edges-weights of the rest of the graph, which are all set to be constant as well (i.e. equal to 1). In our case, we connect the artificial sink to all nodes of the graph and parameterize the weight of the adjacent edges to the artificial sink. The rest of the edge-weights don't have to be constant, but can carry any positive real values. More formally, let $G(V, E)$ be a weighted, undirected graph, as defined in Subsection 4.4.2. Let t be a new node in the graph and connect t with every node in V with an undirected edge of weight equal to some value α. Let $G'(V', E')$ be the new graph, where $V' = V \cup t$ and $E' = E \cup e(t, v), \forall v \in V$. $|V'| = n + 1$ and $|E'| = m + n$.

The outcome of a community S for some sink s with respect to t is directly related to the parametric value of α. This raises the following questions:

a) For some α, what is the quality of community S produced? That is, can we bound the inter- and intra-community cuts somehow?

b) What is a good choice of α in general?

Next we will study several properties of the artificial sink and the new graph G'. We will also propose a general clustering algorithm for G and answer the above questions for α and S in terms of that algorithm.

Quality of the community

Let G' be the graph including the artificial sink t as defined above. If s $(\neq t)$ is any source in G', and S is the community of s with respect to t, we will show in this section that we can lower bound the expansion of S, thus providing a strong quality measure for S. In particular, the expansion of S is lower bound by α, the parameter connecting t to all other nodes. That is, we will show that $\phi(S) \geq \alpha$.

Let T' be the min-cut tree of G' and let $X = V - S$. Then, $V' = S \cup t \cup X$. Figure 4.12 shows how T' looks. Call $v \in S$ the node that is adjacent to t in T'. Now let (P, Q) be any cut of S. $P \cup Q = S$, $P \cap Q = \emptyset$. We wish to lower bound $c(P, Q)$, the cut-value between P and Q. (Figure 4.13.)

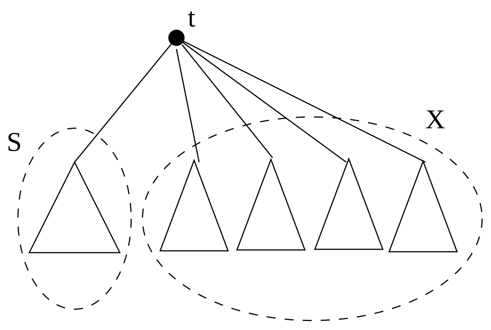

Figure 4.12: Communities of the min-cut tree

First, we need a couple of lemmata and the notion of 'contraction'. When contracting a subset A of V in G, we replace A by a single node. Loops get removed and parallel edges are merged into a single edge with weight equal to the sum of the weights of the parallel edges.

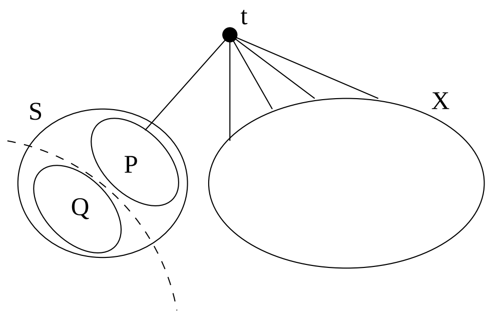

Figure 4.13: Cut within a community

Lemma 4.4.1 *Let T be the (unique) min-cut tree of an undirected graph G, and let A be a subtree of T. Let G' be the graph that results after contracting A in G, and let T' be the min-cut tree of G'. Let T'' be the tree that results after contracting A in T. Then T' and T'' are identical.*

Proof. The proof follows directly from [45] and the method by which a min-cut tree is computed. ■

Lemma 4.4.2 *Let T' be the min-cut tree of G' and let S, P, Q, X, and v be as defined above. Contract $t \cup X$ into x in G' and T'. Then $c(x, Q) \leq c(P, Q)$, where $c(x, Q)$ and $c(P, Q)$ correspond to the total edge-weight connecting x and Q, and x and P respectively.*

Proof. As defined above, v is the node of S that is adjacent to t in T, or equivalently, adjacent to x in T'. But by the definition of a min-cut tree, and the fact that v is in P it follows that the cut that separates x from $P \cup Q$ is smaller than the cut between P and $x \cup Q$. Or, $c(x, P \cup Q) \leq c(P, x \cup Q) \Rightarrow c(x, P) + c(x, Q) \leq c(P, x) + c(P, Q)$, and the lemma follows. ■

We now lower-bound the expansion of S:

Lemma 4.4.3 *For S as defined above, the expansion of S in G is lower-bounded by α. That is, for any $P, Q \subset A$, s.t. $P \cup Q = A$ and $P \cap Q = \emptyset$,*

$$\frac{c(P, Q)}{\min\{|P|, |Q|\}} \geq \alpha.$$

Proof. By Lemma 4.4.1 we can contract $t \cup X$ in T and G and treat it as a single node, say y. Then y is connected in T to either a node of P, or a node of Q. W.l.o.g. say P. But then Lemma 4.4.2 can be applied, according to which $c_{T'}(y, Q) \leq c_{T'}(P, Q) \Rightarrow$. But $c_{T'}(y, Q)$ in contracted T is equal to $c_T(t \cup X, Q)$ in uncontracted T. So, $c_T(t \cup X, Q) \leq c_T(P, Q) \Rightarrow c_T(t, Q) + c_T(X, Q) \leq c_T(P, Q) \Rightarrow |Q|\alpha + c_T(X, Q) \leq c_T(P, Q) \Rightarrow |Q|\alpha \leq c_T(P, Q) \Rightarrow \alpha \leq \frac{c_T(P,Q)}{|Q|} \Rightarrow \alpha \leq \frac{c_T(P,Q)}{\min\{|P|,|Q|\}}$. (Note: We use the notation $c_{T'}$ for the cuts in contracted T and c_T in uncontracted T, to avoid confusion). ■

Cut-clustering algorithm

We can use the ideas of the previous section to develop a general clustering algorithm for graph G. Simply, add an artificial sink t with parameter α. Call the new graph G' and let T' be its min-cut tree. Remove t from T'. The remaining connected components (former communities of T') form the clusters. We call this algorithm the 'cut-clustering algorithm' (Figure 4.14).

```
CutCluster_Basic($G(V, E), \alpha$)
    Let $V' = V \cup t$
    For all nodes $v \in V$
        Connect $t$ to $v$ with edge of weight $\alpha$
    Let $G'(V', E')$ be the new graph after connecting $t$ to $V$
    Calculate the minimum-cut tree $T'$ of $G'$
    Remove $t$ from $T'$
    Return all connected components as the clusters of $G$
```

Figure 4.14: Basic cut-clustering algorithm

The value of α is a lower-bound on the expansion of every cluster formed by this procedure. This follows directly from Lemma 4.4.3. Thus, the algorithm guarantees high quality within each cluster. But for the cut-clustering algorithm we can also upper-bound the cut-values between clusters. In particular, we show the following lemma:

Lemma 4.4.4 *Let S be a cluster produced by the cut-clustering algorithm applied to graph $G(V, E)$. Let t be the artificial sink, G' the new graph, T' its min-cut tree, and let $X = V - S$. Then for S we can show that*

$$\frac{c(S, V-S)}{|V|} \leq \frac{c(S, V-S)}{|V-S|} \leq \alpha.$$

Proof. Because the min-cut tree puts X on the t side of the minimum cut between

S and t, we have that $c(S, X) + c(S, t) \leq c(S, t) + c(X, t) \Leftrightarrow c(S, x) \leq |X|\alpha$, or equivalently, $\frac{c(S,X)}{|X|} \leq \alpha \Rightarrow \frac{c(S,V-S)}{|V|} \leq \alpha$. ■

With regard to the bi-criteria posed in [59], which we mentioned in the previous section, we see that our algorithm satisfies the same lower bound for the expansion of the clusters. (In fact, because of the slack variables removed in Lemma 4.4.3, the connectedness of the clusters produced is often stronger than that indicated by the expansion measure.) Where the criteria differ is in the upper-bounds of the cuts between individual clusters. In [59], the total sum of edge-weights between clusters is upper-bounded by a fraction ϵ of the sum of all edges, where ϵ is a parameter to the input. Our bi-criterion upper-bounds the cut-value between clusters in terms of expansion-like measures.

If we wish to approximate an upper bound on the sum of the edge-weights between clusters, here are some approaches:

1) The inter-cluster cuts are also upper-bound in our algorithm, $c(S, X) \leq |X|\alpha$, thus summing up over all clusters we find that $W_I \leq n(k-1)\alpha$, where W_I is the sum of all inter-cluster cuts and k is the number of clusters produced. If $n(k-1)\alpha \leq \epsilon W$, where W is the total sum of all edge-weights, the upper bound criterion is satisfied as well.

2) If $n(k-1)\alpha > \epsilon W$, we can limit the number of clusters the algorithm returns, so not to exceed the upper bound. Of course, this way the algorithm fails to cluster the entire graph, but still produces heavily connected components within the graph. In practice, it is often the case that besides bigger clusters of the graph, many singletons get produced as well. This happens when the lower bound expansion criterion is too strong to allow for some nodes to be co-clustered with other nodes. In that case, we either relax the expansion criterion (i.e. decrease α) or focus only on a subset of the clusters produced, usually the biggest in size, or those with smallest inter-cluster

weight.

In general, our algorithm comes close to satisfying the above conditions besides imposing its own criteria that guarantee high quality of the clusters produced.

Choice of α

We have addressed earlier the question of a good choice for α. But how does the number and sizes of clusters change with respect to α?

It is easy to see that if α is very small, close to 0, the min-cut between t and any other node of G will produce a trivial cut isolating t from the rest of the nodes. The value of this cut is $n\alpha$, and thus by decreasing α enough we can always guarantee that t will be a single node on one side of the cut, and hence it will be an external node in the min-cut tree; thus, the clustering algorithm will produce only one cluster with respect to t, and this cluster will be the entire graph G.

On the other extreme, if α is very large it will cause T to be a star with t at its center. If W is the total sum of the weights of all edges in G, for any value of $\alpha > W$, and for all $v \in V$, the cut between t and v will produce two sets, $V - v$ and v. Thus, the clustering algorithm will produce n trivial clusters, all singletons.

For values of α between these two extremes the number of clusters will be between 1 and n, but the exact value depends on the structure of G and the distribution of the weights over the edges. What is important, though, is that the number of clusters increases monotonically with α. If we increase the value of α, the number of clusters can only increase as well, or stay the same. When implementing our algorithm we often need to apply a binary-search-like approach in order to determine the best value for α, or make use of the 'nesting property'.

The nesting property has been used by Gallo et.al. [32] in the context of parametric maximum flow algorithms. A parametric network is defined as a regular network G

with source s and sink t, only the edge-weights are linear functions of a parameter λ, as follows:

1) $w_\lambda(s,v)$ is a non-decreasing function of λ for all $v \neq t$.

2) $w_\lambda(v,t)$ is a non-increasing function of λ for all $v \neq s$.

3) $w_\lambda(v,w)$ is constant for all $v \neq s$, $w \neq t$.

A maximum flow or minimum cut in the parametric network G corresponds to a maximum flow or minimum cut in G for some particular value of λ.

This parametric network allows directed edges, thus applies directly to the undirected case as well. Also, it is immediate to see that it is a generalized version of our graph G' after the artificial sink has been added to G with parameter α. In fact, the weights are a linear function only for the edges adjacent to the artificial sink, and we can use both non-decreasing and non-increasing values for α, since t can be treated as the sink, but also as the source of G'.

The following lemma holds for our graph G':

Lemma 4.4.5 *For a source s in G' a given on-line sequence of parameter values $\alpha_1 < \alpha_2 < ... < \alpha_{max}$, yields a sequence $S_1, S_2, ...S_{max}$ of communities of s with respect to t, such that $S_1 \subseteq S_2 \subseteq ... \subseteq S_{max}$.*

Proof. This is a direct result of a similar lemma in [32]. ∎

In fact, in [32] it has been shown that for some s the total number of different communities S_i is no more than $n-2$ and they can all be computed in time proportional to a single max-flow computation, when a variation of the Goldberg-Tarjan preflow-push algorithm [42] is employed.

Thus, if we want to find a cluster of s in G of certain size or other characteristic we can simply use this methodology, find all clusters fast and then choose the one that fits best. Also, because the parametric preflow algorithm in [32] finds all clusters either in increasing or decreasing size, we can stop the algorithm as soon as a desired cluster has been found. We shall use the nesting property again, in Subsection 4.4.4, when introducing the recursive cut-clustering algorithm.

Heuristic for the cut-clustering algorithm

The running time of the Basic cut-clustering algorithm is equal to the time to calculate the minimum-cut tree, plus a small overhead for extracting the subtrees under t. But calculating the minimum-cut tree can be equivalent to computing $n-1$ maximum flows [45] in the worst case. We now present a heuristic that finds the clusters of G much faster, usually in time proportional to the number of final clusters produced.

Lemma 4.4.6 *Let $v_1, v_2 \in V$ and C_1, C_2 be their communities with respect to t in G'. Then either C_1 and C_2 are disjoint or the one is a subset of the other.*

Proof. This is a special case of a more general lemma in [45]. If C_1 and C_2 overlap without the one containing the other then either $C_1 \cap C_2$ or $C_1 - C_2$ is a smaller community for v_1, or symmetrically, either $C_1 \cap C_2$ or $C_2 - C_1$ is a smaller community for v_2. Thus C_1 and C_2 are disjoint or the one is a subset of the other. ■

We use the above lemma in order to reduce the number of minimum cut computations necessary. If the cut between some node v and t yields the community (and candidate cluster) C, then we don't use any of the nodes in C as sources to find following minimum cuts with t, since according to the previous lemma, they cannot produce better (i.e. larger) communities. Instead we mark them as part of community C, and later, if C becomes a part of a larger community C' we mark all nodes of C as part of C'. The heuristic relies on the choice of the next node for which we find its minimum cut with respect to t. The bigger the cluster produced, the smaller the number of minimum cut calculations. We sort all nodes by the sum of the weights of their adjacent edges, in decreasing order. Each time we compute the minimum cut between the next unmarked node and t. We have seen that in practice this reduces the number of max-flow computations almost to the number of communities (and final clusters) in G, speeding up the algorithm significantly.

4.4.4 Recursive Algorithm

We have referred to contraction on a set of nodes in Section 3.2. We can use successive contractions to develop a recursive cut-clustering algorithm. After having applied the basic cut-clustering algorithm to graph G, with some initial value of α, we can apply the same algorithm on the clusters, instead of the nodes. For this, we contract the clusters of G into single nodes. Contracting the clusters of the initial graph yields a new graph, for which we can apply the basic algorithm as before, with a new α-value, which has to be smaller, for otherwise no new clusters will be found. (This is a direct result of the nesting property of Lemma 4.4.5.) This results in clustering the clusters of the initial graph, and can serve as a method for creating a hierarchy of clusters. Each time the current clusters get contracted and the procedure repeated.

The quality of the clusters at each level of the hierarchy is the same as for the initial basic algorithm, depending each time on the value of α, only the expansion measure is now over the clusters instead over the nodes.

The recursion stops either when the clusters returned are of desired number and/or size, or when the cut-clustering algorithm fails to produce other clusters besides the extreme cases of a single cluster and all singletons.

In our experiments we used cluster contraction to produce such a hierarchy of clusters, and the outline of the algorithm is described in Figure 4.15.

Hierarchical clustering offers a better 'locality' view on the structure of the graph. A node of the graph can be part of a small, very dense cluster, but it can also be part of a more general cluster that contains a broader set of nodes. Hierarchical clustering will produce both of these clusters, at different levels, providing a better overview of the structure of the graph.

```
CutCluster_Recursive($G(V, E)$)
    Let $G^0 = G$
    For $(i = 0; ; i++)$
        Set new, smaller value $\alpha_i$ /* possibly parametric */
        Call CutCluster_Basic($G^i$, $\alpha_i$)
        If ((clusters returned are of desired number and size) or
            (clustering failed to create non-trivial clusters))
                break
        Contract clusters to produce $G^{i+1}$
    Return all clusters at all levels
```

Figure 4.15: Recursive cut-clustering algorithm

It is also a direct result of the nesting property (Lemma 4.4.5) that the clusters at higher levels will be supersets of clusters at lower levels:

Lemma 4.4.7 *Let $\alpha_1 < \alpha_2 < ... < \alpha_{max}$ be a sequence of parameter values that connect t to V in G'. Let $\alpha_0 \leq \alpha_1$ be small enough to yield a single cluster in G and $\alpha_{max+1} \geq \alpha_{max}$ be large enough to yield all singletons. Then all α_i values, for $0 \leq i \leq max$, yield clusters in G which are supersets of the clusters produced by each α_{i+1}, and all clusterings together form a hierarchical tree over the clusterings of G.*

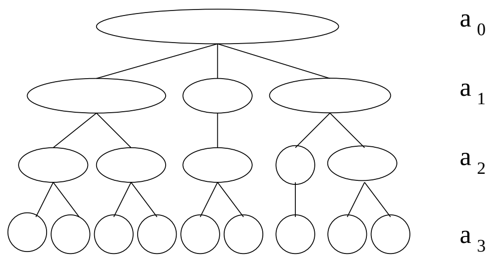

Figure 4.16: Hierarchical tree of clusters

4.4.5 Results

Citeseer data set

Citeseer [81], is a digital library for scientific literature. Scientific literature can be viewed as a graph, where the documents correspond to the nodes and the citations between documents to the edges that connect them.

Our experiment was performed on a large subset of Citeseer, with 132,210 documents and 461,170 citations. All edges were undirected with normalized weight over the number of outbound citations. That is, if document A cites 12 other documents, the node corresponding to document A gets connected via undirected edges to each of the 12 documents, and the weight of each edge will be 1/12. Weights for the inbound edges are determined by the nodes for which they are outpointing. In the end, rescaling might be necessary over all weights, if many of them are too small for practical use. Normalization is a fair method for assigning edge-weights, since some documents may point to many other documents and some only to few. This way, the sum of all out-pointing edges for all nodes is equal.

In the experiment we apply the recursive cut-clustering algorithm of Figure 4.15. We start with the largest possible value of α that gives non-trivial clusters (not all singletons), as described earlier in this section. Then we contract those clusters, producing the graph of next level. We cluster again with the largest value of α. These are clusters of the second level and correspond to clusters of clusters of nodes. We repeat the process until it is not possible to create any more clusters between a single one and all singletons.

From the clusters produced, we conclude that at lower levels the clusters contain nodes that correspond to documents with very specific content. For example, there are small clusters (usually less than 10 nodes) that focus on topics like 'LogP Model of Parallel Computation', 'Wavelets and Digital Image Compression', 'Low Power CMOS', 'Nonholonomic Motion Planning', 'Bayesian Interpolation', and thousands others. In order to evaluate the quality of the clusters, we ranked all documents within each cluster according to the total edge-weight by which they are connected to the rest of the documents in that cluster, that is, the sum of all edges between that node and the rest of the nodes in that cluster. For each cluster, the top-ranked document was the most heavily connected one within its cluster, and the bottom-ranked document was the one least connected to the other nodes. In Table 4.1 we present 3 documents for the clusters mentioned above. For each entry the first document is the top-ranked document of its cluster, the second document is of medium ranking, and the third document is the bottom-ranked document of its cluster. We can see that the clusters contain documents of high quality, closely related to each other.

LogP Model of Parallel Computation
LogP: Towards a Realistic Model of Parallel Computation
A Realistic Cost Model for the Communication Time in Parallel Programs
LogP Modelling of List Algorithms

Wavelets and Digital Image Compression
An Overview Of Wavelet Based Multiresolution Analyses
Wavelet Based Image Compression
Wavelets and Digital Image Compression Part I & II

Low Power CMOS
Low Power CMOS Digital Design
Power-Time Tradeoffs In Digital Filter Design And Implementation
Input Synchronization in Low Power CMOS Arithmetic Circuit Design

Nonholonomic Motion Planning
Nonholonomic Motion Planning: Steering Using Sinusoids
Nonholomobile Robot Manual
On Motion Planning of Nonholonomic Mobile Robots

Bayesian Interpolation
Bayesian Interpolation
Benchmarking Bayesian neural networks for time series forecasting
Bayesian linear regression

Table 4.1: Citeseer data - example titles are from the highest, median, and lowest ranking papers within a community, thus demonstrating that the communities are topically focused

For clusters of higher levels, we notice that they combine clusters of lower levels and singletons that haven't been clustered with other nodes yet. The clusters of these levels (approximately after 10 iterations) focus on more general topics like 'Concurrent Neural Networks', 'Software Engineering, Verification, Validation', 'DNA and Bio Computing', 'Encryption' etc.

At the highest levels, the clusters become quite general, with broad topics covering entire areas like 'Networks', 'Databases', 'Programming Languages', etc. At these levels, we can also draw conclusions about the distribution of the documents in the database. E.g. what percentage of papers is related to 'Algorithms' more than to the other clusters, or what are the sub-clusters of cluster 'Compilers', and so on.

Because of space-limitations and the huge number of papers in the database, it is difficult to include more detailed results of the citeseer data. Nevertheless, we can conclude that the recursive cut-clustering algorithm provides a good tool for hierarchical clustering of large graphs, and can be applied to a wide range of applications, as long as they can be described efficiently in the form of a graph.

Also, as has been pointed out in Subsection 4.4.3, because of the strict bounds that the algorithm imposes on the expansion for each cluster, it may happen that only a subgraph of G gets clustered into non-trivial clusters. This is also the case for the citeseer data, which is a very sparse graph (with an average node degree of only approx. 3.5). Thus, many of the initial documents never get clustered with other nodes (except at the highest level). Nevertheless, when this is the case, the clusters that get produced are still of very high quality and can serve as representative communities or seed sets for other, less strict clustering algorithms (we have used the clusters from the citeseer data in combination with the clustering algorithm of [25] and clustered the entire citeseer data into topic-related levels, exactly as described above).

We elaborate on the citeseer data in much more detail in Section 5.2.

4.4.6 Conclusions

We have shown that minimum-cut trees with artificial sinks provide a good mean for strong clustering (or for extracting heavily connected components) from both a theoretical perspective and in practice. Cut-clustering algorithms are relatively fast (especially with the use of the heuristic of Subsection 4.4.3), they are simple to implement and give robust results. The flexibility of choosing α, and thus determining the quality of the clusters produced, is a great advantage of the cut-clustering method. Also, if α is not given, we can find all breakpoints of α fast [32].

On the other hand, one limitation of our algorithms is the fact that they don't parameterize over the number and sizes of the clusters. The clusters are a natural result of the algorithm and cannot be set as desired (unless searched for by repeated calculations). Also, another problem might occur by the fact that clusters are not allowed to overlap. But sometimes it seems more natural to assign a node to multiple categories, something we have also noticed in our citeseer data-set. This is a more general limitation of all clustering algorithms that produce disjoint clusters.

Implementation-wise, maximum flow algorithms have been sped-up significantly over the past years [40], [15], but they are still computationally intense. Randomized or approximation algorithms could yield similar results but in less time, thus laying the basis for very fast cut-clustering techniques of high quality.

Chapter 5

Case Studies and Experimental Results

This chapter reports on experimental studies we performed on large real-world data-sets. Section 5.1 describes the clustering methods used in our experiments. In order to quantify the quality of the produced clusterings, we refer to the theoretical bounds given by the algorithms used, but also use information theoretical measures on the content of the clustered documents. This provides additional insight about the strengths and weaknesses of the algorithms.

There are two experiments we report on. The first, presented in Section 5.2, is on the citeseer ([81]) data-set, an extensive collection of documents of scientific literature. The second, in Section 5.3, is on a set of web-pages from the Open Directory Project, or dmoz [82].

The chapter concludes with a discussion about the strengths and weaknesses of our algorithms, potential improvements on the maximum flow algorithms used, and final remarks.

5.1 Methodology

5.1.1 Cluster Extraction

In the previous sections we presented and analyzed algorithms for link-based graph clustering. But not all algorithms produce clusters of same quality, or are easy to implement. The cut clustering algorithm can produce clusters of high quality, but it may fail to cluster the entire data. Instead, it often finds strong connected communities within the graph. The community clustering algorithm produces esoteric communities, which in general are quite good, but not as strong as those produced by the cut-clustering algorithm. Also, finding the best esoteric communities is not always easy, since there can be multiple solutions, and the problem itself is NP-hard. Finally, agglomerate, link-based algorithms are very simple to implement, but run the danger of producing poor clusters because of the local optimization they do. In our attempt to cluster large real-world data-sets, we realized that the best clusterings were produced by a combination of the above algorithms. We start off with strong clustering criteria, which we progressively weaken, in order to get more general clusters or hierarchical clusterings.

Progressive link-structure method

More specifically, our method is to initially apply the cut clustering algorithm for several values of α. We set certain constraints on the number and sizes of the clusters, and find the appropriate value for α that satisfies those. To find a clustering that satisfies both the number and sizes of the clusters may not always be feasible. In that case we can either relax the bounds, use a different algorithm or say that the clustering failed. From our experience, and as mentioned before, in most of the cases we experimented with, the cut clustering algorithm does not produce a partition of the graph, but rather extracts communities that cover portions of the graph, usually uniformly distributed.

At this point, we relax the clustering constraints and form esoteric communities, based on the initial communities found by the cut clustering algorithm. If $C = \{C_1, C_2, ..., C_k\}$ is such an initial clustering, then for all nodes $v \notin \bigcup_i C_i$ we assign v to a cluster C_i if v is connected to C_i with edges that have weight at least half of v's total adjacent weight. We repeat until no remaining node can be included in some of the clusters. The new, expanded clusters contain more nodes of the initial graph, and in order for us to accept C as a valid clustering of G, we require that the clusters cover almost the entire graph. If that is not the case, we can either weaken the criterion that expands the communities, or report on the current clusters only, or say that the algorithm failed. Weakening the criterion can be done in several ways, e.g. we can assign v to the cluster it is heaviest connected to (instead of requiring it to form an esoteric community). In our experiments that was never the case, and all expanded clusters formed esoteric communities.

Finally, at the third step, and only if the current number of clusters is too large, we merge them into bigger clusters, based on the links that connect them. In this case, we assign a *connectedness* measure between clusters. Let C_i, C_j be two clusters with sizes $|C_i|$ and $|C_j|$, respectively. Also, let $W_{i,j}$ be the sum of the weights of all edges between the two clusters. We define the *connectedness* measure μ for C_i and C_j to be equal to

$$\mu(C_i, C_j) = \frac{W_{ij}}{|C_i||C_j|} \tag{5.1}$$

which corresponds to the average edge-weight normalized by the sizes of the clusters. The merging algorithm considers among all pairs of clusters, the one with highest value of μ. Then, these two clusters get merged and the process repeated for the next highest-ranking pair. Notice that merging C_i with C_j deletes these two clusters and replaces them by a new cluster, which has size equal to the sum of their sizes and different edges connecting it to the remaining clusters. We can stop the merging at any number of clusters, but typical values range between 10 and 100, depending on the number of nodes and edges in the initial graph, and the current value of μ. In most cases, esp. for the citeseer data, we didn't need to make use of this step, since the cut clustering algorithm, together with the cluster expansion, produced good enough results.

Figure 5.1 shows the three steps just described.

```
PROGRESSIVEALGORITHM($G(V, E)$)
    1) Apply cut-clustering algorithm to find initial clusters
    2) Expand clusters from Step 1 into esoteric communities
    3) Merge clusters from Step 2 according to the edge-weights between clusters
```

Figure 5.1: Progressive link-structure method

5.1.2 Information theoretical measures

Except of quantifying the quality of the clusters by terms of expansion or minimum cut criteria, we can also draw conclusions from the content of the documents clustered. Both the citeseer and the dmoz data-sets contain text documents from which we can extract features that characterize individual documents or clusters of documents. To do this, we use information theoretical measures, applied to combinations of words extracted from the documents. We describe these techniques next.

Entropy Loss based Feature Extraction

Let D be a set of $|D|$ text documents. Divide D into two sets P (positive documents) and N (negative documents), according to some arbitrary partitioning criterion. $|P|+$

$|N| = |D|$.

Define a *feature* to be any consecutive 1-, 2-, and 3-word combination from the documents of D. We wish to extract those features that characterize the positive documents P. The basic idea is to assign scores to all the features, so that if a feature appears many times in the positive set and only few times in the negative set, it gets a higher score. The scores are defined in terms of expected entropy loss.

More specifically, we compute the entropy for each feature f independently. Let C be the event indicating whether a document is positive. The prior entropy of the class distribution is

$$e = -Pr(C)\lg(Pr(C)) - Pr(\overline{C})\lg(Pr(\overline{C})) \tag{5.2}$$

The posterior entropy of the class when the feature is present is

$$e_f = -Pr(C|f)\lg(Pr(C|f)) - Pr(\overline{C}|f)\lg(Pr(\overline{C}|f)) \tag{5.3}$$

Likewise, the posterior entropy of the class when the feature is absent is

$$e_{\overline{f}} = -Pr(C|\overline{f})\lg(Pr(C|\overline{f})) - Pr(\overline{C}|\overline{f})\lg(Pr(\overline{C}|\overline{f})) \tag{5.4}$$

Thus, the expected posterior entropy is

$$Pr(f)\lg(Pr(f)) + Pr(\overline{f})\lg(Pr(\overline{f})) \tag{5.5}$$

and the expected entropy loss is

$$e - (Pr(f)\lg(Pr(f)) + Pr(\overline{f})\lg(Pr(\overline{f}))) \tag{5.6}$$

For more details see [37], from which we adopted the above technique.

In order to extract the top features that characterize P, we extract all the features from P that satisfy minimal thresholds (which are parametric to the input) and rank them according to their expected entropy loss. The top features provide a good measure for the textual coherence of the documents in the cluster, and that also often translates into a good quality measure of the cluster, in general. A cluster that has many top-ranking features of low score is likely to be too general, and thus of low quality. Ideally, a cluster of high quality has several high-ranking features (maybe 10-15, depending on the size of the cluster) that are focused around some specific subject, but also several other lower-ranking features that correspond to subcategories within the cluster and demonstrate its internal structure.

The expected entropy loss method has been analyzed and used successfully in [37] and [38] for the naming of web-communities. The exact implementation issues are beyond this work, and we use this method only as an additional quality measure for our clusters, since it provides a simple, illustrative way to get a good idea about the clusters formed and the textual features they contain.

Cluster Naming

An extension to the feature extraction is that of naming the clusters. By using statistical measures ([36]) we were able to predict likely names for each cluster (SELF), as well as PARENT and CHILD features. SELF features are those that best characterize the cluster itself. PARENT features are those that are more general and would be good for characterizing a concept of which this cluster would be a subset. CHILD features are those that are common to the cluster, but do not fully describe it; they correspond to sub-concepts within the cluster.

Again, we don't focus any further on the exact way these names are extracted, or the accuracy of this particular method, but use it as yet another way to measure the quality of the clusterings, keeping in mind that eventual inaccurate and erroneous namings should be expected. But the fact that wrong namings will only lower the quality measure, together with the actual, surprisingly good results we got, are a strong indication about the potential of this method, as well as about the quality of our clusterings.

5.1.3 Experimental Setup

For our computations, we used two different computers. The first, *kyle*, is a Dell XPS850r, with a Pentium III processor at 850Mhz, and 512MB RAM. The second, *gaea*, is a Dell Precision Workstation 730, with four Itanium processors at 800Mhz each, and 64GB RAM.

Both machines run Redhat Linux 7.1, and all codes were written in C, and compiled with 'gcc' and optimization option -O4.

Most of the experiments were done on *kyle*. We used *gaea* for the larger data-sets, some of which required several gigabytes of RAM.

5.2 Citeseer data

5.2.1 Description of the Citeseer data

Citeseer [81] is a digital library for scientific literature. It currently contains several hundreds of thousands of documents and is growing rapidly. Our experiments were performed on a snapshot of citeseer, containing 1,345,911 document titles and 3,050,745 citations between them. The number of actual documents in the database was 144,812 and the citations between these documents only were 471,537. For these documents we also had available the title and abstract, which we used to extract content-based features.

We applied our progressive clustering algorithm on the graph, where every node corresponded to one of the 144,812 documents, and the edges corresponded to the citations between them. Before running the algorithm, we separated the graph into connected components, since we can work on each of them individually. The largest such connected component consisted of 132,210, or 91.298% of the initial nodes, and 461,170 edges. This was by far the most interesting connected component, and it is the only one we report results on here.

Another preprocessing step required, before applying the clustering algorithm, was that of making the graph undirected. Citations between two documents are directed, and so were the edges of the graph. We have seen in Section 4.1 how to transform directed edges into undirected ones by normalizing their weight. Note that initially all edge-weights are equal to 1.

Finally, when studying the results of the clustering algorithm, it is important to keep in mind special characteristics of the citeseer data:

1) The graph is very sparse. The average node-degree is less than 7.

2) Citations point almost always backwards in time. Thus the initial directed graph is acyclic, with more recent papers pointing to older ones. It has been argued in other studies (e.g. [70]) that because of this, older documents might be favored and a penalty parameter, depending on how old a paper is, should be introduced. We chose to avoid such a parameter, and handle all documents the same way. (In [70] the evolution of clusters was being studied and time was an important factor.)

3) Noise. The documents in citeseer are processed automatically, and often contain parse errors, wrong field entries, multiple copies of the same document, and other mistakes. This may drop the quality of the results, although we noticed that in

general our algorithm was very stable, e.g. by clustering together different copies of the same paper, etc.

5.2.2 Initial cluster extraction

In order to cluster the citeseer data, we initially applied the cut clustering algorithm for several values of α. Among those, we noticed that as α increases, the smallest value not yielding a trivial, single cluster, produced exactly two clusters: one of size 88, and one of size 132,122. Since the larger cluster again covers almost the entire data, we increased α to the immediately next value that produced more than two clusters. That value was $\alpha = 10$ (actually, the range $[10, 20]$ of values resulted in the same clustering), and the number of clusters now jumped into the thousands.

We applied the heuristic of Subsection 4.4.3 and extracted the first n clusters, where $n \in \{5, 10, 20, 50, 100, 500, 1000\}$. Table 5.1 shows the sizes of these clusters. The first column of the table refers to the number of clusters, and the second column shows the total number of nodes in the clusters. The third column shows the total number of nodes as a percentage over all nodes in the graph.

So, e.g. the top 100 clusters contained 3,517 nodes (35.17 nodes on average), or covered 2.660% of the citeseer documents.

#clusters	init. #nodes	percent.	exp. #nodes	percent.
5	466	0.352%	132,210	100%
10	1,008	0.762%	132,210	100%
20	1,609	1.217%	132,203	99.995%
50	2,539	1.920%	132,199	99.992%
100	3,517	2.660%	132,196	99.989%
500	7,969	6.028%	132,202	99.994%
1000	11,498	8.697%	132,199	99.992%

Table 5.1: Sizes of initial and expanded clusters

Measuring the quality of the clusters by applying Lemma 4.4.3, we see that the expansion of all clusters produced is at least $\alpha = 10$. A more practical measure is that of the expected entropy loss and the percentage of positive occurrences of the features extracted from the title and the abstract of the documents.

Table 5.2 refers to the features extracted from the top 100 and 1000 clusters, and we see that for 1000 clusters the top feature appears in 68.5% of the documents of each cluster. The top three features appear in 62.9% and the top ten features in 53.5% of the documents. So, even though the clusters were formed based on link-structure only, they have high percentages of the features characterizing them. For other numbers of clusters the percentages are similar.

#top feat.	aver. % pos. 100 clust.	aver. % pos. 1000 clust.
1	66.8	68.5
2	64.7	65.5
3	61.4	62.9
5	57.4	59.2
10	51.6	53.5
20	44.8	46.9
30	40.8	43.1

Table 5.2: Top features after initial iteration

5.2.3 Cluster expansion

After calculating the initial clusters, we expanded them as described earlier, thus creating esoteric communities. It is very interesting that this cluster expansion method resulted in assigning at least 132,196 nodes, that is all but 14 nodes, or 99.989% of the entire citeseer data, to some cluster! And this was true for all different cases of numbers of clusters! (See columns 4 and 5 of Table 5.1.)

Thus, we conclude that the initial clusters are evenly spread over the citeseer graph, in all cases, capturing its entire distribution of heavily connected components.

#clusters	min	max	mean	aver.	st.dev.
5	17,666	36,722	22,738	26,442	7,898
10	8,945	16,569	13,255	13,221	2,513
20	3,282	10,655	6,324	6,610	1,917
50	695	5,991	2,433	2,644	1,275
100	282	4,633	1,147	1,322	750
500	39	2,191	214	264	201
1000	10	1,377	102	132	111

Table 5.3: Statistics for expanded cluster sizes

This, together with the low degree of the nodes (6.976 on average) are the two main reasons for the resulting partitioning. It is also interesting that even though many of the initial clusters were very small (that is, singletons or 2-node clusters), esp. for 100, 500, and 1000 clusters, the expanded clusters (or esoteric communities) had sizes at least 282, 39, and 10 respectively. Also, the largest communities never grow too large, and the standard deviation from the average size is small as well. Statistics for the expanded cluster sizes are shown in Table 5.3. So, for example, for 20 clusters the smallest cluster contained 3,282 nodes, the largest 10,655, and the mean 6,324 nodes. The average size was 6,610 nodes with standard deviation 1,917.

#top feat.	aver. % pos. 100 clus.	aver. % pos. 1000 clus.
1	48.9	49.7
2	44.1	45.0
3	40.7	41.9
5	36.3	37.7
10	30.2	31.5
20	24.0	25.2
30	21.1	21.9

Table 5.4: Top features after cluster expansion

Concentrating again on the positive percentage of the top features (in the case of 1000 clusters), we can see from Table 5.4 that the top feature occurs in 49.7% of the documents within each cluster. The top three and top ten features occur in 41.9%

and 31.5% of the documents respectively. Again, even though the clustering has been entirely based on the link-structure the features demonstrate high concentration.

Also, Tables 5.5 and 5.6 show the top 5 features for 5 clusters and 20 clusters, respectively. (Tables are sorted by cluster size.) From these features we can see how concentrated the clusters are content-wise. 5 clusters are probably too few for clustering the entire data-set, but the features do provide a good high-level idea about the very general classes of the citeseer data. 20 clusters seem to give a better classification; the clusters are now very focused, though still general enough. As the number of clusters increases, they become even more focused on very specific subjects. Table 5.7 shows the top 3 features for the largest and smallest among 1000 clusters.

Size	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
36,722	image	numerical	method	d	dimensional
32,575	protocol	network	users	learning	access
22,737	logic	language	calculus	semantics	reasoning
22,509	specification	formal	software	verification	development
17,666	parallel	memory	performance	processor	parallelism

Table 5.5: Top 5 features for 5 clusters

One minor issue is that of the (up to 14) remaining nodes that were never clustered. Even though they also often formed esoteric communities themselves, we prefered to remove them from our data-set and worked with the remaining nodes. This seemed the most fair approach and had virtually no impact on the results.

5.2.4 Cluster merging

At this point, in the cases of 5-100 clusters, the algorithm stopped. But for more than 100 clusters we continued with the third step of the progressive link-based method. Next, we describe the results for 1000 clusters.

Size	Features 1-5
10,655	security, secure, polynomial, protocols, n
9,259	numerical, parallel, equations, matrix, solution of
8,543	database, query, object oriented, object, relational
8,314	specification, formal, software, verification, development
8,079	image, vision, motion, camera, scene
7,982	memory, shared memory, parallel, shared, processor
7,685	speech, word, corpus, neural, speech recognition
7,460	robot, agent, planning, autonomous, environment
7,320	parallel, memory, compiler, parallelism, fortran
6,328	service, multimedia, qos, of service, quality of service
6,320	wavelet, regression, estimation, bayesian, function
6,039	genetic, evolutionary, genetic algorithm, fitness, ga
5,969	rewriting, logic, calculus, languages, functional
5,600	logic, reasoning, logic programming, semantics, knowledge
5,597	network, traffic, packet, tcp, internet
4,734	proof, calculus, theorem, proving, type
4,858	learning, neural, neural network, training, machine learning
4,214	software, code, program, language, compiler
3,964	markov, markov chain, random, chain, stochastic
3,282	learning, training, neural, classification, recognition

Table 5.6: Top 5 features for 20 clusters

After 1000 esoteric communities covered the citeseer data, we combined them iteratively into fewer, and thus larger, clusters. By applying the merging algorithm, we noticed that the most interesting clusters were formed when they were between 20 and 200. While more than 200, the clusters were still pretty small and often separate clusters focused on very similar topics and should actually be merged together. When less than 20, especially when less than 10, the clusters tended to become too general.

We have extracted the names for each of the clusters at certain levels according to the methodology described in Section 5.1.2. Table 5.8 displays the top two SELF, PARENT, and CHILD names for the 10 largest clusters, and Table 5.9 displays the top names for the 10 smallest clusters. At this point the total number of clusters

Size	Feature 1	Feature 2	Feature 3
1377	security	cryptographic	secure
905	rewriting	term rewriting	rewrite
846	recurrent	neural	connectionist
828	parallel	hpf	fortran
807	regression	nonparametric	estimation
789	genetic	genetic programming	evolutionary
752	formal	specification	vdm
732	mpi	parallel	message passing
670	image	motion	optical flow
586	neural	neural networks	learning
510	robots	robot	autonomous
500	markov	markov chain	stochastic
479	synthesis	embedded	dsp
478	graph	vertices	drawing
474	mixture	of experts	neural
463	dna	adleman	of dna
458	nonlinear	stability	lyapunov
458	numerical	iterative	equations
457	type theory	proof	theorem
28	coding	wavelet	coder
27	unix	explicit dynamic linking	system v
26	service scheduling	service scheduling scheme	service
26	propositional	satis ability sat	ability sat
26	code	run time	proof carrying code
25	in files can	structuring schema	in files
25	wireless	atm	wireless atm
25	shared memory	shared	memory
25	cache	memory	referencing patterns
25	protocol	protocols	performance
22	logic programs	the well founded	logic
21	knowledge	knowledge representation	number restrictions
20	multicast	reliable	reliable multicast
19	distributed memory	parallel	fortran
19	mining	association	association rules
17	timed	clocks	timed systems
14	discovery	knowledge discovery	tasa telecommunication
10	require differ. views	in multidimensional	processing olap systems

Table 5.7: Top 3 features for largest and smallest of 1000 clusters

had been reduced to 100. Note that the SELF features are almost always also the top-ranking features, and that very similar entries in multiple columns are the result of insufficiencies of the naming algorithm.

size/ index	SELF 1 SELF 2	PARENT 1 PARENT 2	CHILD 1 CHILD 2
2852 988	security public key	protocols protocol	cryptographic attacks
2824 989	quality of service multimedia applications	network control	multicast end to end
2452 905	neural networks genetic algorithms	learning problems	genetic programming artificial neural
2448 981	server file system	network control	disks caching
2385 994	data parallel high performance	memory program	high performance fortran hpf
2264 941	machine learning learning algorithms	algorithms methods	case based reasoning decision trees
2146 976	mobile robot robot	environment control	navigation sensors
2114 996	high performance message passing	parallel communication	mpi distributed memory
2071 987	optimization problems polynomial time	linear g	semidefinite programming semidefinite
1957 954	camera vision	image method	uncalibrated calibration

Table 5.8: Names for the largest 10 clusters

5.2.5 Feature overlap

Besides the progressive link-based algorithm, we also experimented with a different method that merges the expanded clusters from the second step, not by heaviest links between clusters, but by largest feature overlap.

More specifically, the algorithm in this method is the same as that of the previous subsection for the first two steps, but in the third step it is not link-based

size/ index	SELF 1 SELF 2	PARENT 1 PARENT 2	CHILD 1 CHILD 2
617 803	probabilistic belief	logic reasoning	belief revision possibilistic
605 950	parallel programming data parallel	programs implementation	skeletons higher order
593 983	reinforcement learning learning algorithms	algorithms control	value function reinforcement
584 425	matrices singular value decomposition	algorithms method	subspace eigenvalue
579 800	ad hoc network wireless	algorithms simulation	routing protocol shortest path
519 949	model checking finite state	verification logic	infinite state ctl
507 902	computer graphics global illumination	method algorithms	radiosity hierarchical radiosity
459 998	real time systems qualitative	logic models	qualitative simulation temporal logic
361 612	hypermedia hypermedia systems	support work	open hypermedia hypertext
356 852	linear time algorithm linear time	graph algorithms	treewidth classes of graphs

Table 5.9: Names for the smallest 10 clusters

anymore. Instead, it compares the features between all pairs of the 1000 clusters. The comparison is done using a TFIDF on the feature weights.

There are a number of TFIDF algorithms which differ in their selection of feature weighting method and similarity measure [73]. The basic idea is to represent each document d as a vector $\vec{d} = (d^{(1)}, ..., d^{(|F|)})$ in a vector space, so that documents with similar content have similar vectors. Each dimension of the vector space represents a feature, that is a combination of words, selected as described earlier. We extract the top 50 features (ranked by expected entropy loss) for each document, which means $|F| = 50$ in our case.

The values of the vector elements $d^{(i)}$ are calculated as a combination of the statistics $TF(f, d)$ and $DF(f)$. The *term frequency* $TF(f, d)$ is the number of times feature f occurs in document d. The *document frequency* $DF(f)$ is the number of documents in which the feature f occurs at least once. The *inverse document frequency* $IDF(f)$ can be calculated from the document frequency:

$$IDF(f) = \log(\frac{|D|}{DF(f)}) \tag{5.7}$$

$|D|$ is the total number of documents. The inverse document frequency of a feature is low if it occurs in many documents and is highest if the feature occurs in only one. The value $d^{(i)}$ of feature f_i for document d is then calculated as the product

$$d^{(i)} = TF(f_i, d) \cdot IDF(f_i) \tag{5.8}$$

$d^{(i)}$ is called the weight of feature f_i in document d.

We used the TFIDF model to rank all pairs of clusters, so that the pair with highest feature similarity gets the highest rank. Feature similarity is defined as follows. Let C_k and C_l be two of the 1000 initial clusters. Let f be a common feature among the top 50 features in both clusters. We adjust the definition of the weight of feature f to apply to clusters instead of documents only:

$$d_k = TF(f, C_k) \cdot IDF(f) \tag{5.9}$$

The weight d_k of feature f in cluster C_k is based on the number of documents in C_k that contain feature f. The product $d_k \cdot d_l$ gives a similarity measure between C_k and C_l with respect to feature f. We sum up the TFIDF products over all common features in C_k and C_l and the result will be the similarity score between them.

Once we have found the TFIDF scores between all pairs of clusters, we merge the two clusters with the highest score, and recalculate all TFIDF's. (In practice we only need to find the TFIDF values between the newly formed cluster and the remaining clusters; the other values stay the same.)

Comparing the results for this method with that of Subsection 5.1.1, we conclude that when the total number of clusters is large (close to 1000) or very small (fewer than 20) the clusters produced are of higher quality. This is because for a large number of clusters, those that get merged are subject-wise extremely close. For a very small number of clusters, few features dominate each set, forcing clusters with similar main content to get merged together.

In contrast, for an average number of clusters (100-500), the progressive link-structure method is more effective, since it combines clusters that feature-wise may not be of very high quality, but link-wise are closely related to each other.

5.3 Web communities

5.3.1 Description of Data and Preprocessing

The Open Directory Project [82], or *dmoz*, is a human edited directory of the Web. Currently, it contains 3,218,307 sites in 369,906 categories, and 46,374 editors who synopsize and categorize new entries.

Our experiment was conducted as follows. We started off with the homepage of the Open directory, `http://www.dmoz.com`, and crawled all web-pages up to two links away, in a breadth-first approach. This produced 2,312,358 webpages.

In order to cluster these webpages, we represent them as nodes of a graph, where the edges correspond to hyperlinks between them, but we don't include links between web-pages of the same domain, because this biases the graph. We will elaborate more on links within the same domain at the end of this section. Removing isolated nodes, we get a graph of 1,269,838 nodes and 1,673,380 edges. At this point we apply another preprocessing step: We remove all nodes of single degree. Nodes of single degree can only be part of the cluster their adjacent neighbor belongs to, or form a singleton community by themselves. Cutting off single degree nodes, reduces the graph substantially and decreases the running time of the algorithms. Once the clustering has been performed, we can easily re-attach the single nodes to those clusters where they don't violate the clustering criteria. It turns out that 89.6% of all nodes in the graph have single degree, and removing them results in 131,965 nodes and 553,693 edges. The largest connected component of this graph covers 124,963 nodes and has 540,423 edges. This is the graph we are going to report on.

Similarly to the citeseer data, we again apply the progressive link-based method, starting off with the cut-clustering algorithm, expanding clusters into esoteric communities, and merging communities based on the link structure, when necessary. Of course, before applying any of these algorithms, we normalize the edges by outbound links, thus making the graph undirected.

5.3.2 Initial cluster extraction

Applying the cut-clustering algorithm, for various values of α, we see that the smallest value that produces more than one clusters is 2. But for $\alpha \in [2,6]$, and number of clusters ranging between 3 and 46, there is always one dominant cluster that contains at least 122,908 nodes, or 98.356% of the graph. Increasing α to 7 breaks this one large cluster, and produces thousands of smaller clusters, very similar in nature to the citeseer data. There are many clusters that contain a few hundred nodes. The biggest cluster contains about 9000 nodes. Looking at the web-pages in this cluster, we see that they were from the web-directory *Yahoo*. For larger values of α, about 1000, the Yahoo cluster breaks also, and the number of clusters now increases into the tens of thousands. For $\alpha = 10000$, all clusters are singletons. From the above description, we conclude that the most interesting cases are when α is between 7 and 1000, and these are the cases we will study in more detail.

5.3.3 Cluster expansion

After we find the initial clusters for several values of α, we expand these clusters, as in the citeseer case. We notice that this time the expanded clusters grow approximately by 30% on average and don't cover the entire data-set as before. For example, for $\alpha = 10$, the top 789 initial clusters cover 38,350 nodes, or 30.689% and expanded they cover 51,550 nodes, or 41.252% of the entire graph. Similar numbers apply for the other values in that range.

But, even though the number of clusters is now bigger, and they are harder to expand, their quality is again very high. In fact, features have a much higher positive percentage than for the citeseer data, averaging over 80% for the top 3-5 features for the range of α we study. Looking at the actual documents within the clusters, we see what the reason for this is. There are mainly two different types of clusters: The first type are clusters that contain documents from the same domain, or from similar categories. We excluded links within the same domain, but the Open Directory contains thousand of subcategories that cover extremely diverse and unrelated topics. This forces the clustering algorithm to create many small, but highly focused clusters. Web-pages from the same domain get merged together because of their adjacent web-pages from other domains with similar subject.

The second type are clusters that are broad subcategories of dmoz. These, in fact, are the most common among the largest clusters. Characteristically, such clusters can be seen in Table 5.10, which shows the top three features for the 12 clusters containing at least 200 nodes.

Size	Feature 1	Feature 2	Feature 3
8456	in yahoo	yahoo inc	copy yahoo inc
826	about altavista	altavista company	altavista reg
512	software search on	software	software top
354	and economy search	economy top regional	and economy about
334	health about dmoz	health search on	health top
331	employment about dmoz	employment search on	employment top
318	and tourism search	tourism search	tourism top
316	personal pages search	pages search on	personal pages
261	and environment search	and environment top	environment top regional
255	and entertain search	and entertainment about	and entertainment top
254	and culture top	and culture about	and culture search
236	and sports search	sports top regional	and sport top

Table 5.10: Top 3 features for largest dmoz clusters

5.3.4 Cluster merging

Since the number of clusters was very large, we applied the third step of the progressive link-based method, and merged together clusters that were heavily linked. The top features produced this way are again very similar to those of Table 5.10, only the expected entropy loss drops significantly very quickly. When the clusters become less than approx. 100, the features are not good enough to distinguish them, and the algorithm stopped.

Overall, the dmoz data-set was harder to cluster, mainly because of the very broad topics it covered. The citeseer data was larger, but the documents were more related to each other, since they were all from scientific literature, with main focus on computer science. Also, the features extracted from the citeseer clusters were of higher quality than those from the web-pages, for high-level clusterings. Another reason for the relatively poor naming of the feature extraction method, applied to web-pages, has been pointed out in [38]. There, we make the following claim, which we verify: the features extracted from the text of a web-page might be a poor measure for describing its content. Much better features can be extracted from the extended anchortext of the pages that point to the documents in the clusters. Unfortunately, we didn't have this information available for our data-set.

Still, we believe that the clusters extracted from the dmoz data-set can be of high quality (as we have seen in Table 5.10 for small clusters), but they are harder to cluster into more general groups, and even harder to name by the feature-extraction algorithm based on expected entropy loss.

5.4 Conclusions

5.4.1 Running times

Besides the quality of the clustering, another important factor that determines the applicability of an algorithm is speed. Most of the agglomerate methods, the cluster expansion, the feature extraction, and all the pre- and post-processing can be done in linear or sublinear time. But algorithms that contain maximum flow techniques at their core are superlinear by nature.

Currently, the fastest maximum flow algorithm ([41]) has a running time of $O(\min(n^{2/3}, m^{1/2})m \log(\frac{n^2}{m}) \log U)$. The fastest maximum flow implementation is based on Goldberg's push-relabel method, and we extracted it from a previous experimental study on minimum cuts ([15]). We used this same algorithm in both the experimental study of Section 3.2 and the clustering algorithms of Section 4. The theoretical bound of this algorithm is $O(n^2 m^{1/2})$, which for sparse graphs (with bounded average degree) translates to $O(n^{5/2})$. Heuristics, like the global update, speed up significantly the actual running time. For the data-sets in our experiments we calculated the complexity of the algorithm to less than quadratic, about $O(n^{1.3})$ to $O(n^{1.5})$. In practice, this translates into few (less than 30) seconds for graphs with 100,000-200,000 nodes and average degree of 3-7. Of course, the exact running time also depends on the actual structure of the graph, the implementation, the machine used, etc. With our experimental setup, and the given maximum flow implementations, for each data-set the clustering algorithm required between a few minutes (for graphs with few thousand nodes and edges) and 2-3 days for the largest instances, with all heuristics employed.

As said earlier, often the biggest bottleneck is the maximum flow routine. Since it was the currently fastest known implementation, we considered approximation algorithms that might be faster. Most interesting among those was an algorithm by Gabow ([30], and also [31]) that allows for subsequent iterations that yield flows ever closer to the maximum. This algorithm restricts edge-weights to depend on a parameter P, which is always a power of 2. Initially P is equal to the largest power of 2 that is smaller than at least one edge-weight of the graph. All edges get capacity 1 if their weight is larger than P, and 0 if it's less. A maximum flow algorithm is performed on the (now unit-capacity) network, and subsequently, P is set to half its initial value, and again 0 and 1 values are assigned to edges in the residual graph. After $\lg P \geq \lg w_{max} + 1$ iterations, where w_{max} is the maximum initial edge-weight, the algorithm converges to the exact maximum flow value.

The idea in the approximation variation is to stop the iterations when P is beyond some threshold. Ideally, this should reduce the number of necessary iterations, and if the threshold is not too low, the approximate maximum flow value should be close enough to the actual value. ([30] also shows how close this approximation will be, depending on the number of iterations.) Unfortunately, in practice this idea didn't perform too well. The additional overhead for keeping track of the virtual edge-weights, was about the same as the time saved from the approximation. Also, Gabow's algorithm is based on augmenting paths, whereas Goldberg's algorithm is based on local pushes and relabels. We were not able to adjust the one to the other and still preserve the advantages of the approximation algorithm.

We don't elaborate any further into this idea, but it would certainly be interesting to see faster implementations in practice, even in the form of approximation and randomized algorithms.

5.4.2 Internal links

In addition to the dmoz data-set, we also tried another data-set of web-pages, crawled from the internet. But this time we didn't exclude links within the same domain. The initial seed set was also just a random set of nodes, and the final graph we worked on contained 202,132 nodes and 1,072,327 links. Applying the progressive link-based method for different values of α, and number of clusters, we concluded that the clusters were of extremely low quality, because of the bias of links within the same domain. This links were dominating in many cases, and didn't allow for more general clusters to be built.

Also, when we attempted to extract features from these clusters, they often had extremely high occurrences in the positive set, but only because they were phrases common within that community, like "click here", "prev", "next", "copyright", etc. We concluded that any link-based clustering algorithm for web-pages, must exclude, or at least handle in a special way, interconnecting links. The dmoz data-set would most likely have similar resulted, hadn't these links been removed prior to any clustering.

Bibliography

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. *Automatic subspace clustering of high dimensional data for data mining application.* In Proc. ACM SIGMOD Conference on Management of Data, pp. 94–104, 1998.

[2] S. Aksoy and R. M. Haralick. *Graph theoretic clustering for image grouping and retrieval.* In CVP, pp. 63–68, 1999.

[3] M. S. Aldenderfer and R. K. Blashfield. *Cluster Analysis.* Sage, Beverly Hills, 1984.

[4] M. R. Andenberg. *Cluster Analysis for Applications.* Academic Press, Inc., New York, 1973.

[5] R. J. Anderson and J. C. Setubal. *Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study.* In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.

[6] D. L. Applegate and W. J. Cook. Personal communication. Rice University, 1997.

[7] A. Benczúr. *Counterexamples for directed and node capacitated cut-trees.* SIAM J. Comput. 24:3 (1995), pp. 505-510.

[8] M. Bern and D. Eppstein. *Approximation Algorithms for Geometric Problems.* In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1996.

[9] E. Bolten, A. Schliep, S. Schneckener, D. Schomburg, and R. Schrader. *Clustering Protein Sequences - Structure Prediction by transitive homology.* Technical Report, No. zaik2000-383, Zentrum für Angewandte Informatik Köln, 2000.

[10] F. J. Brandenburg and A. Sen. *Graph Clustering II: Trees of Cliques with Size Bounds.* to appear, 1999.

[11] S. Brin and L. Page. *Anatomy of a Large-Scale Hypertextual Web Search Engine*, Proc. 7th International World Wide Web Conference, 1998.

[12] S. Chakrabarti, B. Dom, P. Raghavan, S. Rajagopalan, D. Gibson, and J. Kleinberg. Automatic resource compilation by analyzing hyperlink structure and associated text. In Proc. 7th World Wide Web Conference, Brisbane, Australia, 1998.

[13] M. Charikar, C. Chekuri, T. Feder and R. Motwani. *Incremental clustering and dynamic information retrieval* In Proc. of 29th STOC, 1997.

[14] M. Charikar, S. Guha, D. Shmoys and E. Tardos. *A constant-factor approximation for the k-median problem.* In Proc. of 31st STOC, pp. 1–10, 1999.

[15] C. S. Chekuri, A. V. Goldberg D. R. Karger, M. S. Levine, and C. Stein. *Experimental Study of Minimum Cut Algorithms.* In Proc. 8th ACM-SIAM Symposium on Discrete Algorithms, pages 324–333, 1997.

[16] B. V. Cherkassky and A. V. Goldberg. *On Implementing Push-Relabel Method for the Maximum Flow Problem.* Algorithmica, 19:390–410, 1997.

[17] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. *Scatter/gather: a cluster-based approach to browsing large document collections.* In Proceedings of the 15th Annual International ACM/SIGIR Conference, pages 318–329, Copenhagen, Denmark, June 1992.

[18] D. Defays. *An efficient algorithm for a complete link method.* The Computer Journal, 18:364–366, 1977.

[19] U. Derigs and W. Meier. *Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation.* ZOR — Methods and Models of Operations Research, 33:383–403, 1989.

[20] S. van Dongen *Performance criteria for graph clustering and Markov cluster experiments.* Technical report, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, 2000.

[21] P. Drineas, A. Frieze, R. Kannan, S. Vempala and V. Vinay. *Clustering in large graphs and matrices.* In Proc. of 10th SODA, 1999.

[22] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. *Knowledge discovery in large spatial databases: Focusing techniques for ecient class identication.* In Proc. of 4th Int'l Symposium on Large Spatial Databases, 1995.

[23] B. Everitt. *Cluster analysis.* Halsted Press, New York, 1980.

[24] D. Fasulo. *An Analysis of Recent Work on Clustering Algorithms* Technical Report No. UW-CSE-01-03-02, Univ. of Washington, 1999.

[25] G. W. Flake, S. Lawrence and C. L. Giles. *Efficient identification of web communities.* In Proceedings of the Sixth International Conference on Knowledge

Discovery and Data Mining (ACM SIGKDD-2000), Boston, MA, 2000. ACM Press.

[26] G. W. Flake, K. Tsioutsiouliklis and S. Lawrence. *Minimum Cut Clustering*, The Learning Workshop, Snowbird, Utah, April 10-13 2001.

[27] G. W. Flake, R. E. Tarjan and K. Tsioutsiouliklis. *Graph Clustering Techniques based on Minimum Cut Trees* Technical Report 2002-006, NEC, Princeton, NJ, 2002.

[28] L. Fleischer. *Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic run time* Journal of Algorithms, 33:51–72, 1999.

[29] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[30] H. N. Gabow. *Scaling algorithms for network problems.* J. Comput. System Sci. 31 (1985), no. 2, 148–168.

[31] H. Gabow, R. E. Tarjan. *Faster scaling algorithms for network problems.* SIAM J. Comput. 18 (1989), no. 5, 1013-1036.

[32] G. Gallo, M. D. Grigoriadis and R. E. Tarjan. *A Fast Parametric Maximum Flow Algorithm and Applications.* SIAM Journal of Computing, Vol. 18, No. 1 (1989) 30-55.

[33] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[34] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. *Partition-based clustering in object bases: From theory to practice.* In Proc. of the Intl. Conf. on Foundations

of Data Organization and Algorithms (FODO), volume 730 of Lecture Notes in Computer Science (LNCS), pages 301–316, Chicago, IL, October 1993. Springer-Verlag.

[35] D. Gibson, J. Kleinberg, and P. Raghavan. *Inferring Web communities from link topology.* In Proceedings of the Ninth ACM Conference on Hypertext, Structural Queries, pages 225-234, 1998.

[36] E. Glover. Personal communication. NEC Research Institute, 2002.

[37] E. J. Glover, G. W. Flake, S. Lawrence, W. P. Birmingham, A. Kruger, C. Lee Giles, D. M. Pennock. *Improving Category Specific Web Search by Learning Query Modifications*, Symposium on Applications and the Internet, SAINT 2001, San Diego, California, January 8–12, 2001.

[38] E. Glover, K. Tsioutsiouliklis, S. Lawrence, D. Pennock, G.W. Flake. *Using Web Structure for Classifying and Describing Web Pages*, 11th International World Wide Web Conference, Honolulu, Hawaii, 2002.

[39] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[40] A. V. Goldberg. *Recent developments in maximum flow algorithms.* Technical Report No. 98-045, NEC Research Institute, Inc., Apr. 1998.

[41] A. V. Goldberg and S. Rao. *Beyond the Flow Decomposition Barrier.* J. Assoc. Comput. Mach., 45:753–782, 1998.

[42] A. V. Goldberg and R. E. Tarjan. *A New Approach to the Maximum Flow Problem.* J. Assoc. Comput. Mach., 35:921–940, 1988.

[43] A. V. Goldberg and K. Tsioutsiouliklis. *Cut tree algorthms.* Proc. 10th ACM-SIAM Symposium on Discrete Algorithms, pp. 376–385, 1999.

[44] A. V. Goldberg and K. Tsioutsiouliklis. *Cut tree algorthms: an experimental study.* J. Algorithms. 38:51–83, 1 (2001).

[45] R. E. Gomory and T. C. Hu. *Multi-terminal network flows.* J. SIAM, 9:551–570, 1961.

[46] T. F. Gonzalez. *Clustering to minimize the maximum intercluster distance.* Theoretical Computer Science, 38:293–306, 1985.

[47] M. Groetschel. Personal communication. ZIB Berlin, 1997.

[48] D. Gusfield. *Very Simple Methods for All Pairs Network Flow Analysis.* SIAM Journal on Computing, 19:143–155, 1990.

[49] J. Hao and J. B. Orlin. *A Faster Algorithm for Finding the Minimum Cut in a Directed Graph.* J. Algorithms, 17:424–446, 1994.

[50] F. Harary. *Graph Theory.* Addison-Wesley, Mass., 1969.

[51] E. Hartuv and R. Shamir. *A clustering algorithm based on graph connectivity.* Information processing letters, pp. 175–181, 2000.

[52] D. Hochbaum. *Various Notions of Approximations: Good, Better, Best and More.* In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1996.

[53] T. Hoya. *Graph Theoretic Techniques for Pruning Data and Their Applications.* IEEE Transaction on Signal Processing, pp. 2574-2579, Vol. 46, No. 9, Sept. 1998.

[54] T. C. Hu. *Combinatorial Algorithms.* Addison-Wesley, Reading, MA, 1982.

[55] Y. -W. Huang, N. Jing, and E. A. Rundensteiner. *Effective Graph Clustering for Path Queries in Digital Map Databases.* Proc. Fifth Int'l Conf. Information and Knowledge Management, pp. 215–222, 1996.

[56] P. Indyk. *A sublinear time approximation scheme for clustering in metric spaces.* In Proc. of 40th FOCS, pp. 154–159, 1999.

[57] A. K. Jain and R. C. Dubes. *Algorithms and Clustering Data* Prentice-Hall, NJ, 1988.

[58] K. Jain and V. Vazirani. *Primal-dual approximation algorithms for metric facility location and k-median problems* In Proc. of 40th FOCS, pp. 2–13, 1999.

[59] R. Kannan, S. Vempala, and A. Vetta. *On Clusterings - Good, Bad and Spectral.* In IEEE Symposium on Foundations of Computer Science, pages 367-377, 2000.

[60] R. M. Karp. *Reducibility among combinatorial problems.* In R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 85-103.

[61] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley & Sons, Inc., 1990.

[62] J. Kleinberg. *Authoritative sources in a hyperlinked environment.* Proc. 9th ACM-SIAM Symposium on Discrete Algorithms, 1998.

[63] M. Juenger, G. Rinaldi, and S. Thienel. *Practical performance of efficient minimum cut algorithms.* In Proc. 1st Workshop on Algorithm Engineering, Venice, Italy, 1997. Available via URL `http://www.dsi.unive.it/~wae97/proceedings/`.

[64] M. S. Levine. *Experimental Study of Minimum Cut Algorithms.* Technical Report MIT-LCS-TR-719, MIT Lab for Computer Science, 1997.

[65] J. MacQueen. *Some methods for classification and analysis of multivariate observations.* Proc. 5th Berkeley Symposium, 1:281–297, 1967.

[66] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. *Using automatic clustering to produce high-level system organizations of source code.* In Proceedings of the 6th Intl. Workshop on Program Comprehension, June 1998.

[67] H. Nagamochi, T. Ono, and T. Ibaraki. *Implementing an Efficient Minimum Capacity Cut Algorithm.* Math. Prog., 67:297–324, 1994.

[68] Q. C. Nguyen and V. Venkateswaran. *Implementations of Goldberg-Tarjan Maximum Flow Algorithm.* In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.

[69] M. Padberg and G. Rinaldi. *An Efficient Algorithm for the Minimum Capacity Cut Problem.* Math. Prog., 47:19–36, 1990.

[70] A. Popescul, G. W. Flake, S. Lawrence, L. Ungar, and C. L. Giles. *Clustering and identifying temporal trends in document databases.* In Proc. IEEE Advances in Digital Libraries 2000, 2000.

[71] S. Rizzi. *A Genetic Approach to Hierarchical Clustering of Euclidean Graphs.* Proceedings International Conference on Pattern Recognition (ICPR'98), Brisbane, Australia, pp. 1543-1545, 1998.

[72] R. Sablowski and A. Frick. *Automatic Graph Clustering.* Proceedings of Graph Drawing'96, Berkeley, California, September,1996.

[73] G. Salton and C. Buckley. *Term Weighting Approaches in Automatic Text Retrieval*, Technical Report 87-881, Dept. of Comp. Science, Cornell University, 1987.

[74] H. Saran and V. V. Vazirani. *Finding k-cuts within twice the optimal.* In IEEE, editor, Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, pages 743–751, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.

[75] R. Sibson. *Slink: an optimally efficient algorithm for a complete link method.* The Computer Journal, 16:30–34, 1973.

[76] E. M. Voorhees. *Implementing agglomerative hierarchic clustering algorithms for use in document retrieval.* Information Processing & Management, 22(6):465–476, 1986.

[77] J. H. Ward Jr. *Hierarchical grouping to optimize an objective function.* Journal of the American Statistical Association, 58(301): 235–244, 1963.

[78] Z. Wu and R. Leahy. *An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation.* IEEE Transactions on Pattern Analysis and Machine Intelligence, November 1993.

[79] C.-H. Yeh and B. Parhami. *Unified formulation of a wide class of scalable interconnection networks based on recursive graphs.* Proc. 11th Int'l Conf. Sys. Engineering, 1996.

[80] AltaVista - The Search Company. `http://www.altavista.com`.

[81] ResearchIndex, NEC Research Institute. `http://www.citeseer.com`.

[82] Open Directory Project. `http://www.dmoz.com`.

[83] Google. `http://www.google.com`.