APPROXIMATION ALGORITHMS FOR A GRAPH-CUT PROBLEM WITH APPLICATIONS TO A CLUSTERING PROBLEM IN BIOINFORMATICS

SALIMUR RASHID CHOUDHURY
Bachelor of Science, Islamic University of Technology, 2004

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Salimur R. Choudhury, 2008

I dedicate this thesis to my ***parents*** *and* ***sister***.

Abstract

Clusters in protein interaction networks can potentially help identify functional relationships among proteins. We study the clustering problem by modeling it as graph cut problems. Given an edge weighted graph, the goal is to partition the graph into a prescribed number of subsets obeying some capacity constraints, so as to maximize the total weight of the edges that are within a subset. Identification of a dense subset might shed some light on the biological function of all the proteins in the subset.

We study integer programming formulations and exhibit large integrality gaps for various formulations. This is indicative of the difficulty in obtaining constant factor approximation algorithms using the primal-dual schema. We propose three approximation algorithms for the problem. We evaluate the algorithms on the database of interacting proteins and on randomly generated graphs. Our experiments show that the algorithms are fast and have good performance ratio in practice.

Acknowledgments

I express my deep acknowledgment and profound sense of gratitude to my supervisor Dr. Daya Gaur for his inspiring guidance, helpful suggestions and persistent encouragement as well as close and constant supervision throughout the period of my Masters degree.

I would also like to thank my M.Sc. supervisory committee members Dr. Hans-Joachim Wieden, Dr. Stephen Wismath and Dr. Shahadat Hossain for their guidance and suggestion. I would also like to thank my external examiner Dr. Abraham Punnen for his valuable suggestions and comments.

I am grateful to Dr. Daya Gaur and to the School of Graduate Studies for the financial assistantships.

I am very much thankful to my family and fellow graduate students Mohammad Tauhidul Islam, Sardar Haque and Sadid Hasan for the continuous encouragement that helped me to complete this thesis.

Contents

Approval/Signature Page ii

Dedication iii

Abstract iv

Acknowledgments v

Table of Contents vi

List of Tables ix

List of Figures x

Glossary xi

1 Introduction 1
- 1.1 Importance of Protein Protein Interaction 2
- 1.2 PPI network Representation 2
- 1.3 Clustering PPI network 3
- 1.4 Definitions 4
  - 1.4.1 Class P 4
  - 1.4.2 Class NP 4
  - 1.4.3 Polynomial time reductions 4
  - 1.4.4 NP-Completeness 5
  - 1.4.5 Approximation Algorithm 5
- 1.5 Organization of the thesis 6

2 Related Work 8
- 2.1 Definitions 8
- 2.2 Previous work 9
  - 2.2.1 Max k-cut 9
  - 2.2.2 Capacitated max k-cut 11
  - 2.2.3 Min k-cut 12
  - 2.2.4 Capacitated min k-cut 12

2.2.5 Multiway cut 13
2.2.6 Max k-uncut 13
2.2.7 Capacitated max k-uncut problem 13
2.2.8 Multiway uncut 13
2.3 Relationship between min k-cut and max k-uncut 14

3 Linear and Integer Linear Programming (ILP) 16
3.1 Linear Programming 16
3.2 Integer Linear Programming 17
3.3 Integer Linear Program for Capacitated Max k-uncut Problem 18
3.4 Another ILP formulation 20
3.5 Linear Programming Relaxation 21
3.6 Integrality Gap 21

4 Approximation Algorithms 25
4.1 Local Search and Recursive Greedy methods 25
4.1.1 Local Search 25
4.1.2 Recursive Greedy method 27
4.2 Local Search Algorithm 27
4.2.1 Definition of Capacitated Max k-uncut problem 27
4.2.2 The Swap Algorithm 27
4.2.3 Approximation Algorithm 28
4.3 Ejection Chain Algorithm 34
4.3.1 Ejection Algorithm 35
4.4 Recursive greedy algorithm 36
4.4.1 Greedy method for max 2-uncut problem 36
4.4.2 Recursive greedy method 37

5 Experiments and Results 41
5.1 Implementation 41
5.1.1 Swap Algorithm 42
5.1.2 Recursive Greedy method 43
5.1.3 Ejection Algorithm 43
5.2 Data Sets 44
5.3 Experimental results 44
5.3.1 Protein Interaction Database 44
5.3.2 Randomly Generated Graphs 54
5.4 Conclusions 62

6 Conclusion and Future Work 66
6.1 Conclusion 66
6.2 Future Research Work 67

Bibliography **68**

List of Tables

2.1 Some partitioning problems 10

5.1 Experimental results on database 1 (uniform sizes) 46
5.2 Experimental results on database 2 (uniform sizes) 48
5.3 Experimental results on database 1 (two unbalanced subsets) 50
5.4 Experimental results on database 2 (two unbalanced subsets) 52
5.5 Experimental results on database 1 (three unbalanced subsets) 54
5.6 Experimental results on database 2 (three unbalanced partitions) 56
5.7 Experiments on random dense graphs 58
5.8 Experiments on random sparse graphs with $p = \frac{5}{|V|}$ 60
5.9 Experiments on random small sparse graphs with $20k$ orderings with $p = \frac{5}{|V|}$ 63
5.10 Experiments on random small sparse graphs with $100k$ orderings with $p = \frac{5}{|V|}$ 64

List of Figures

1.1 Example of a PPI Network 3

2.1 Example Graph 11

5.1 Comparison of the performance ratio of the algorithms on database 1 (uniform sizes) 45
5.2 Comparison of the timing of the algorithms on database 1 (uniform sizes) . 47
5.3 Comparison of the performance ratio of the algorithms on database 2 (uniform sizes) 49
5.4 Comparison of the timing of the algorithms on database 2 (uniform sizes) . 49
5.5 Comparison of the performance ratio of the algorithms on unbalanced subsets of database 1 51
5.6 Comparison of the timing of the algorithms on unbalanced subsets of database 1 51
5.7 Comparison of the performance ratio of the algorithms on two unbalanced subsets of database 2 53
5.8 Comparison of the timing of the algorithms on two unbalanced subsets of database 2 53
5.9 Comparison of the performance ratio of the algorithms on three unbalanced subsets of database 1 55
5.10 Comparison of the timing of the algorithms on three unbalanced subsets of database 1 55
5.11 Comparison of the performance ratio of the algorithms on three unbalanced subsets of database 2 57
5.12 Comparison of the timing of the algorithms on three unbalanced subsets of database 2 57
5.13 Comparison of the performance ratio of the algorithms on random dense graphs 59
5.14 Comparison of the timing of the algorithms on random dense graphs 59
5.15 Comparison of the performance ratio of the algorithms on random sparse graphs with $p = \frac{5}{|V|}$ 61
5.16 Comparison of the timing of the algorithms on random sparse graphs with $p = \frac{5}{|V|}$ 61

5.17 Comparison of the performance between the swap and the ejection chain algorithms on small random sparse graphs with $p = \frac{5}{|V|}$ 65

Glossary

- **Graph**: A non-negative edge weighted undirected *graph* G consists of a set of vertices V and a set of edges E that connect pairs of vertices. Normally we denote the graph as G =(V, E). G may have non-negative edge weights denoted by $w(u,v)$, where $(u,v) \in E$ is an unordered pair of vertices.

- **Hypergraph**: A *hypergraph* $H = (V, E)$ is a generalization of a graph where V denotes the set of vertices and each edge of E is a non-empty subset of V. In a hypergraph, edges can connect any number of vertices (greater that 1).

- **Weight of a graph**: The *weight of a graph* (W) denotes the sum of all the weights of the edges $(u,v) \in E$, where $u \in V$ and $v \in V$; *i.e.* $\sum_{(u,v)\in E} w(u,v)$

- **Partition**: Let $V_i \subseteq V$ for all $i = 1,\ldots,k$ be a collection of subsets of V. This collection is called a partition if $\cup_{i=1}^{k} V_i = V$ and $V_i \cap V_j = \phi$ for all i,j. If V is the vertex set of a graph, then we refer it to as the partition of the graph.

- **Self edge**: Given a partition of V, we define an edge $(u,v) \in E$ as a *self edge* if $u \in V_i$ and $v \in V_i$, for some i that is both of the end points are in the same subset in the partition.

- **Cross edge**: Given a partition of V, we define an edge $(u,v) \in E$ as a *cross edge* if $u \in V_i$ and $v \in V_j$ where $i \neq j$, that is the end points of the edge are in different subsets in the partition.

- **A Matching between two subsets**: A *matching* M between two equal sized sets V_1 and V_2 is defined as a set of pair of vertices (u, v) where $u \in V_1$ and $v \in V_2$ such that $M = \{(u_1, v_1), (u_2, v_2), ...(u_q, v_q)\}$ where $u_1 \neq u_2 ... \neq u_q$ and $v_1 \neq v_2 ... \neq v_q$.

- **Perfect matching between two subsets**: A matching is perfect if $|M| = |V_1| = |V_2|$, *i.e.* all vertices of the subsets are in some pair.

- **Weight of a maching**: The weight w_M of a matching M between two subset of vertices V_1 and V_2 in a weighted graph G is defined as $\sum_{(u,v) \in M} w(u, v)$ where $u \in V_1$ and $v \in V_2$ and $w(u, v)$ is the weight of the edge (u, v).

- **Capacity of a subset**: The capacity s_i of a subset V_i is the maximum number of vertices it can contain.

Chapter 1

Introduction

Clustering plays a vital role in the analysis of data. It has been widely used for a long time in different areas like business analysis, data mining, image analysis. Nowadays it is also being used in bioinformatics to analyze gene structure, protein structure etc.

The goal of clustering is to group the elements into subsets based on similarity among the elements, *i.e.* elements within the same subset should be similar and the elements in the different subsets are dissimilar.

Often we can represent the data sets as weighted graphs, where vertices correspond to the elements to be clustered and the weights of the edges represent similarity between those entities [16]. We can then use graph based clustering algorithms to solve the problem. Graph clustering algorithms typically try to optimize some criteria like minimum sum, minimum diameter, k-median etc. [6].

Graph based clustering has been widely used to solve different types of clustering problems in bioinformatics. Here we mention a few applications of clustering in bioinformatics. Kawaji *et al.* [32] use graph based clustering to cluster protein sequences into families. King *et al.* [35] use cost based clustering on the Protein Protein Interaction networks to identify and predict protein complexes. A graph based clustering algorithm for analyzing gene expression is described in Ron *et al.* [40]. Xu *et al.* [45] use another graph based

method to cluster gene expression data. Hajirasouliha *et al*. [27] use graph based algorithms for optimal pooling of genome re-sequencing.

This thesis is about algorithms for clustering using graph cuts. We develop approximation algorithms for clustering and analyze them theoretically and experimentally. We also demonstrate the efficacy of the algorithms on graphs arising from protein protein interaction networks (PPI).

1.1 Importance of Protein Protein Interaction

The function of unknown proteins may be inferred on the basis of their interaction with a known protein with a known function. Mapping protein protein interactions provides insight into protein function and helps to model the functional pathways to clarify the molecular mechanisms of cellular processes [37]. We can study the protein protein interactions to understand how proteins function within the cell.

1.2 PPI network Representation

We can represent the PPI network using a simple graph. We can represent proteins as nodes and two proteins that interact are represented as adjacent nodes connected by an edge. Figure 1.1 is an example of a PPI network where the nodes represent the proteins and edges represent the interactions between the proteins. So if we can cluster the PPI network then we can find out the characteristics of an unknown protein from the functions of the other proteins that are in the same cluster.

Modeling PPI networks as graphs has been used by many applications, for instance predicting protein complexes within PPI networks [5].

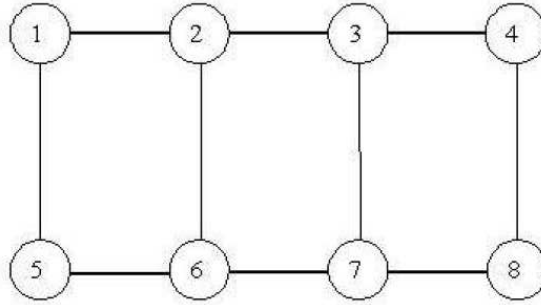

Figure 1.1: Example of a PPI Network

1.3 Clustering PPI network

Partitioning the graph into different subgraphs is the most common method for clustering a graph [37] that lead us to optimization problems like max-cut, min-cut, max-uncut etc. To cluster the PPI network we transform the protein network into a simple graph and then apply graph cut algorithms. The main goal of clustering a PPI network is to put the related proteins into the same cluster; that is we want to minimize the edges across the clusters or maximize the edges within the clusters. We can formulate this problem as a max k-uncut problem that maximizes the edges within the clusters, or as a min k-cut problem that minimizes the edges across the clusters.

In this thesis we design approximation algorithms for max k-uncut problem that can be used to cluster PPI networks with one additional constraint. The constraint is that we can specify the size of the clusters too. We call the max k-uncut problem with this constraint, the *capacitated max k-uncut* problem. We leave it to the experts to draw any biological relevant conclusions from the clustering obtained using our methods (see similar applications in [27]).

1.4 Definitions

Now we define some terms to be used in the following chapters. Please refer to the excellent book by Garey and Johnson [18] for further details.

1.4.1 Class *P*

P is a set of decision problems that can be solved on a deterministic Turing machine in polynomial time. The shortest path problem and breadth first search problem are in class P [18].

1.4.2 Class *NP*

NP is the set of decision problems that can be solved on a nondeterministic Turing machine in polynomial time.

Example: In the vertex cover problem we are given a graph $G = (V,\ E)$ and an integer k. We have to find a subset $V_s \subseteq V$ so that for every edge $(u, v) \in$ E we have either $u \in V_s$ or $v \in V_s$ and $|V_s| = k$. This problem is in the class NP because we can easily design a polynomial time verifier for this problem. The verifier first checks whether $|V_s| = k$ or not. Then for every edge $(u, v) \in E$ it checks whether $u \in V_s$ or $v \in V_s$ and it can do this in polynomial time.

1.4.3 Polynomial time reductions

If we can transform the instances of a problem Π_1 to the instances of another problem Π_2 such that satisfiable instances of Π_1 are mapped to satisfiable instances of Π_2 and vice versa in polynomial time then we call this a polynomial time reduction. Suppose a problem Π_1 is polynomial time reducible to another problem Π_2 then we denote it as $\Pi_1 \leq_p \Pi_2$.

1.4.4 *NP*-Completeness

A problem Π_1 is *NP-Complete* if the following holds:

1. $\Pi_1 \in NP$, and

2. $\Pi_2 \leq_p A$ for every $\Pi_2 \in NP$.

If a problem satisfies the second condition but not necessarily the first one then we call this problem, an *NP* hard problem [9]. It is generally believed that *NP*-complete problems do not lend themselves to efficient algorithms. Approximation algorithms are an elegant way of coping with the intrinsic hardness. We describe them next.

1.4.5 Approximation Algorithm

We know that an optimization problem can be a minimization or a maximization problem. Every optimization problem has three parts [18]:

a. A set of instances (D).

b. For each instance $I \in D$, a finite set of candidate solution $C(I)$.

c. A function f that assigns a positive rational number $f(I,\alpha)$ to each candidate solution $\alpha \in C(I)$ for all $I \in D$. This positive rational number is called the solution value for α.

If the problem is a maximization one then the value for an optimal solution for an instance $I \in D$ is denoted as $OPT(I)$. It is the value of $f(I,\alpha^*)$ of an optimal solution for I where $\alpha^* \in C(I)$. For all $\alpha \in$ C(I), $f(I,\alpha^*) >= f(I,\alpha)$.

A polynomial time algorithm A is an approximation algorithm for a particular optimization problem if given any instance $I \in D$, it finds a "good" candidate solution $\alpha \in C(I)$ of the problem. The value $f(I,\alpha)$ of the candidate solution α found by A when applied to I is denoted as $A(I)$.

For the NP-hard problems there are no known polynomial time algorithms. So our goal is to find an approximation algorithm A that runs in polynomial time and has the property that for all instances I, $A(I)$ is close to $OPT(I)$. The worst case performance of the approximation algorithm is defined as the performance ratio of the algorithm. For a maximization problem, a β approximation produces a solution with value $A(I) \geq \beta\ OPT(I)$ for all instance I in polynomial time. Note that $\beta \leq 1$ and the goal is to design approximation algorithms with β as close to 1 as possible.

1.5 Organization of the thesis

In chapter 2 we describe different optimization problems related to clustering like max k-cut, capacitated max k-cut, min k-cut, capacitated min k-cut, max k-uncut and finally capacitated max k-uncut. We also describe the related research work of these problems.

In chapter 3 we describe the integer linear programs for the capacitated max k-uncut problem. We also study the linear programming relaxations for the integer programs. We exhibit a large integrality gap for the linear programming relaxations for the capacitated max k-uncut.

In chapter 4 we introduce two local search algorithms and one recursive greedy method for solving the capacitated max k-uncut problem. We present the worst case analysis of the approximation ratio in this chapter.

We compare the algorithms (introduced in chapter 4) experimentally in chapter 5 on the graphs arising from PPI networks and on random graphs.

Finally, we conclude the thesis with future research directions in chapter 6.

Chapter 2

Related Work

In this chapter we present some optimization problems that are related to graph partitioning. We start with max k-cut problem and then we present all the other problems in Table 2.1. We notice that problems are different in terms of additional inputs and objective functions with additional constraints.

2.1 Definitions

Max k-cut:

Given an undirected graph $G = (V, E)$ and a positive integer k, with each edge of G having non-negative weight $w(u, v)$ on each edge in E. We need to partition the vertices into $k \geq 2$ subsets so as to maximize the sum of the weights of the cross edges. We call this problem, the *max-cut* problem if the partition size is two. The max k-cut problem is NP-complete [31] for $k = 2$.

Other graph cut problems are described in Table 2.1.

In this thesis we design approximation algorithms for the capacitated max k-uncut problem. In this version, capacities for each subset in the partition are also specified as a part of the input. If we consider edge weight $w(u, v) = 1$ for all the edges $(u, v) \in E$ then we call this

the *unit weighted version* of the problem. Consider the following example for capacitated max k-uncut problem.

In Figure 2.1 we are given a graph with 8 vertices, a positive integer $k = 2$ and the capacities of the two subsets are $s_1 = 4$ and $s_2 = 4$. Our goal is to partition the graph into 2 subsets V_1 and V_2 in such a way so that we maximize the total weight of the self edges whilst maintaining the capacity constraints. From the Figure we notice that one valid solution for this problem can be $V_1 = \{1,2,5,6\}$ and $V_2 = \{3,4,7,8\}$. The total weight of the self edges for this solution is $4(M+\varepsilon)$. The best possible solution is $V_1 = \{1,2,3,4\}$ and $V_2 = \{5,6,7,8\}$ and the total weight of the self edges is $6M$ (here we assume ε is very small compared to M).

Formally we define capacitated max k-uncut problem as follows:

Input: A weighted undirected graph $G = (V,E)$, an integer k and capacities $s_1,\ldots,s_k$, where $\sum_{i=1}^{k} s_i = |V|$.

Output: Partition the vertices into k subsets $V_1,....,V_k$, where the ith subset V_i contains at most s_i vertices and the total weight of the self edges is maximized.

2.2 Previous work

We now present some of the previous works related to the problems defined in Table 2.1 and the problem defined in section 2.1.

2.2.1 Max *k*-cut

Sahni *et al.* [41] give a $1/2$ approximation algorithm for the max cut problem. Goemans *et al.* [22] give a 0.87856 approximation algorithm for the max cut problem using semi definite programming. Goemans *et al.* [23] give a 0.83601 approximation algorithm for

Table 2.1: Some partitioning problems

Problem Name	**Additional inputs**	**Additional Constraint**	**Objective function**
Capacitated max k-cut	Capacities of k subsets, $s_i, \ldots, s_k$	Capacity constraint for each subset and $\sum_{i=1}^{k} s_i = \|V\|$	Maximize the weight of the cross edges
Min k-cut			Minimize the cross edges
Capacitated min k-cut	Capacities of k subsets, $s_i, \ldots, s_k$	Capacity constraint for each subset and $\sum_{i=1}^{k} s_i = \|V\|$	Minimize the weight of the cross edges
Multiway cut	A set of terminals, $t_1, .., t_k \in V$	Each subset contains exactly one terminal	Minimize the weight of the cross edges
Max k-uncut			Maximize the weight of the self edges
Capacitated max k-uncut	Capacities of k subsets, $s_i, \ldots, s_k$	Capacity constraint for each subset and $\sum_{i=1}^{k} s_i = \|V\|$	Maximize the weight of the self edges
Multiway uncut	A set of terminals, $t_1, .., t_k \in V$	Each subset contains exactly one terminal	Maximize the weight of the self edges

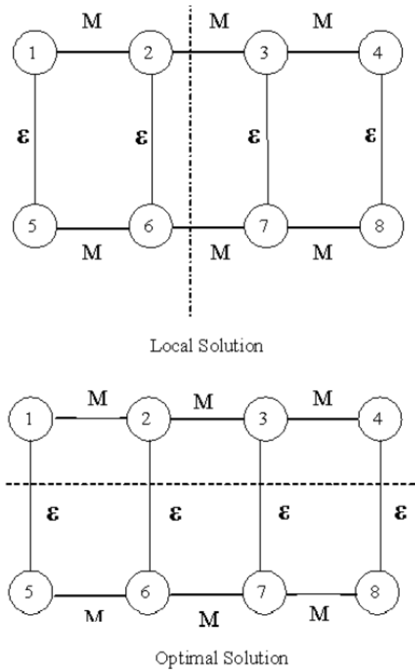

Figure 2.1: Example Graph

the max 3-cut problem (same bounds as in de Klerk *et al.* [13]). Frieze *et al.* [17] obtain a solution for the max k-cut problem with expected value no smaller than $\alpha_k \sim (2logk/k^2)$. Kann *et al.* [30] show that the best possible performance ratio that can be obtained by any algorithm for the max k-cut problem is $1-1/(34k)$ unless $P = NP$. Hajirasouliha *et al.* [27] give a simple local search approximation algorithm that guarantees a $1-1/k$ performance ratio. In this local search algorithm they pick any vertex from any subset and move it to another subset in the partition if it can improve the weight of the cross edges after moving the vertex and continue this step until there is no such vertex.

2.2.2 Capacitated max *k*-cut

Feige *et al.* [14] give an approximation algorithm for unequal capacities with a lower bound of $1/2+\varepsilon$ when $k = 2$, where ε is a universal constant. Andersson [4] describes an algorithm that obtains a $1-1/k+\Omega(1/k^3)$ performance guarantee for equal capacities. Ageev

et al. [2] consider a generalization of capacitated max k-cut and give a $1/2$ approximation algorithm for the max k-cut problem for general hypergraphs with fixed, possibly different, subset sizes. Ageev *et al.* [1] give a $1/2$ approximation algorithm for the capacitated max-2-cut problem. In both cases a randomized rounding technique known as pipage-rounding is used. Gaur *et al.* [20] give a local search algorithm for the uniform capacitated max k-cut problem and obtained a $1-1/k$ performance guarantee.

2.2.3 Min k-cut

We can solve the min cut problem using a standard network flow algorithm [18] in polynomial time. The problem of finding a min k-cut is polynomial time solvable for any fixed k though it is NP-hard if k is a part of the input [43].

There is a $2-2/k$ approximation algorithm due to Saran *et al.* [42] based on Gomory-Hu trees [25]. Boykov *et al.* [7] give a 2-approximation algorithm for the min k-cut problem using a local search based approach.

2.2.4 Capacitated min k-cut

The capacitated min k-cut problem is NP-complete [19], even for $k=2$. To the best of our knowledge there is no known approximation algorithm for the capacitated min k-cut problem.

We use the technique from Gaur *et al.* [20] to approximate the capacitated min k-cut problem. In a single iteration we pick any two vertices from different subsets in the partition and swap the vertices between the subsets if that decreases the weight of the cross edges, and repeat until no such pair exists.

But unfortunately for this algorithm there exists an infinite family of graphs (Figure 2.1) with a bad local optimum. For the example in Figure 2.1 if we start the initial random

partition (for capacitated min 2-cut) as in [Figure 2.1 (local solution)] then we can not perform any swapping as there is no pair of vertices for which we can increase the weight of the self edges by swapping the vertices. So this is a local optimum for this example and the value of the local optimum is $2M$. The optimal solution for this instance is 4ε [Figure 2.1 (optimal solution)] where ε is a small positive number. So the performance ratio of this approximation algorithm for min k-cut is arbitrarily bad.

2.2.5 Multiway cut

There is a $2-2/k$ approximation algorithm for the multiway cut problem due to Dalhous *et al.* [10]. They compute a minimum weight isolating cut called c_i for each $i = 1, \ldots, k$ and then discard the heaviest of these cuts to get the k cut.

2.2.6 Max k-uncut

There is no known approximation algorithm for the max k-uncut problem to the best of our knowledge.

2.2.7 Capacitated max k-uncut problem

There is no previously known approximation algorithm in the literature for this problem.

2.2.8 Multiway uncut

Langberg *et al.* [36] consider the multiway uncut problem and give a 0.8585 approximation algorithm. They use linear programming relaxation and randomized rounding to design the algorithm.

2.3 Relationship between min k-cut and max k-uncut

Theoretically, the min k-cut and max k-uncut problems are related. In the min k-cut problem, our task is to partition V into k subsets so that we can minimize the total weight of the cross edges. On the other hand, in max k-uncut problem we have to maximize the total weight of the self edges.

Given a partition, every edge of G is either a self edge or a cross edge.

If we denote the total weight of the edges as W, the total weight of the cross edges as C and the total weight of the self edges as S then the following holds:

$$S = W - C$$

So we can say that if we can minimize the total weight of the cross edges then we can maximize the total weight of the self edges. From an optimal solution for the min k-cut problem we can get the optimal solution for the max k-uncut problem and this holds true for the capacitated version too. However this relationship does not extend to the approximate solutions, *i.e* a β approximate solution for the min k-cut does not imply a β approximate solution for max k-uncut. We know that both the min k-cut and the max k-uncut problems are NP-Complete so we can only expect approximation algorithms for these problems but the approximation bound given by an approximation algorithm for the min k-cut problem might not give the same approximation bound for the max k-uncut problem. Suppose we have a $\frac{1}{2}$-approximation algorithm for the min k-cut problem and for a given graph the optimal solution is half of the total weight of the edges, that is the total weight of the cross edges in the optimal solution is half of the total weight of the edges. So the algorithm for this particular graph might return the total weight as the weight of the cross edges, as the algorithm gives a $\frac{1}{2}$-approximation. The solution of the max k-uncut problem will be 0 as

there will be no self edges, where as the optimal solution to the max k-unut has weight half the total weight.

In the next chapter we examine two integer programs for the max k-unut problem.

Chapter 3

Linear and Integer Linear Programming (ILP)

We describe linear programming and integer linear programming in sections 3.1 and 3.2 of this chapter. In sections 3.3 and 3.4 we describe two integer linear programs for the capacitated max k-uncut problem. In section 3.5 we describe the linear programming relaxations and study the integrality gap in section 3.6.

3.1 Linear Programming

Linear programming has been widely used to develop approximation algorithms for different optimization problems. We can formulate the optimization problem as an integer linear program and then solve the linear programming relaxations. Finally round the LP solution to obtain an integral solution.

A linear program is defined in terms of an objective function and a set of constraints. The *objective function* is a linear function of decision variables that are unknown and the set of constraints consists of linear equalities and inequalities. The standard form of a linear program is as follows [39]:

$$\text{minimize } \bar{c}\bar{x}$$

$$\text{subject to } A\bar{x} = \bar{b}$$

$$\bar{x} \geq 0$$

This is a *minimization problem*. We can also model a problem as a maximization problem.

The linear function $\bar{c}\bar{x}$ is called the *objective function* where $\bar{c}$ and $\bar{x}$ are vectors. A is a matrix of known coefficients and $\bar{b}$ is a vector. The *decision variables* are represented using vector $\bar{x} = (x_1, x_2, ..x_n)$. An assignment of values to the elements of vector $\bar{x}$ satisfying the constraints is called a *feasible solution*. A feasible solution with minimum objective function value is called an *optimal solution* for a minimization problem, and a feasible solution with maximum objective function value is the optimal solution for a maximization problem.

The simplex algorithm [11] is the most used algorithm to solve a linear program though it is not a polynomial time algorithm in the worst case. Two other polynomial time algorithms for linear programming are due to Khachiyan [34] and Karmakar [12].

3.2 Integer Linear Programming

In the linear programming the variables can take any real values. If we restrict the variables to be integers then we call it an *integer linear program*. The following is the general form of an integer linear program :

$$
\begin{aligned}
\text{minimize } & \bar{c}\bar{x} \\
\text{subject to } & A\bar{x} = \bar{b} \\
& \bar{x} \geq 0 \\
& x_i \in \mathcal{N}
\end{aligned}
$$

Note that variables in $\bar{x}$ are restricted to take integer values. Typically branch-and-bound and cutting plane algorithms are used [39] to solve an integer linear program.

Branch-and-bound is an algorithmic technique to find the optimal solution by keeping the best solution found so far and uses it to prune the search space. It typically enumerates implicitly all the possible candidate solutions for a problem.

Cutting plane algorithms can also be used to solve ILP. Normally in cutting plane algorithms we consider the linear programming relaxation of the problem. Linear programming relaxation (ILP without the integer constraints) might not return an integral solution. So if it does not return the integral solution we add a linear constraint that does not exclude any integer feasible points and we continue this step until we get an integral primal solution or an unbounded dual solution. This linear constraint is called a cutting plane or cut. Gomory [24] developed a method to generate such cuts. Several other methods for generating cuts are known. See the excellent text by Wolsey and Nemhauser [44] for a detailed discussion of these techniques.

3.3 Integer Linear Program for Capacitated Max k-uncut Problem

We develop two integer linear programs for the capacitated max k-uncut problem.

Let $G = (V, E)$ be an edge weighted, undirected graph. We are interested in partitioning V into k subsets $V_1, \ldots, V_k$ with associated capacities $s_1, \ldots, s_k$ and $\sum_{i=1}^{k} s_i = |V|$, so as to maximize the total weight of the self edges among the subsets in the the partition.

We introduce a $0/1$ variable x_{ui} for each vertex $u \in V$ and each subset V_i, which is set to 1 if u is in subset V_i. Let y_{uvi} be another $0/1$ variable for each edge $(u,v) \in E$, and for each subset V_i in the partition, which is set to 1 if both x_{ui} and x_{vi} are set to 1, that is both the end points of an edge are in subset V_i, otherwise it is set to 0. $w(u,v)$ denotes the weight of the edge (u,v) and the objective is to maximize $\sum_{i=1}^{k} \sum_{(u,v) \in E} w(u,v) y_{uvi}$. The ILP is as follows:

$$\text{maximize} \sum_{i=1}^{k} \sum_{(u,v) \in E} w(u,v) y_{uvi} \tag{3.1}$$

$$\text{subject to} \sum_{i=1}^{k} x_{ui} = 1; \text{ for every vertex } u \in V. \tag{3.2}$$

$$y_{uvi} \leq \frac{1}{2}(x_{ui} + x_{vi}); \text{ for } (u,v) \in E \text{ and } i \in [1..k]. \tag{3.3}$$

$$\sum_{u \in V} x_{ui} \leq s_i; \text{ for } i \in [1..k]. \tag{3.4}$$

$$x_{ui} \in \{0,1\}; \forall u \in V \text{ and } i \in [1..k]. \tag{3.5}$$

$$y_{uvi} \in \{0,1\}; \forall (u,v) \in E \text{ and } i \in [1..k]. \tag{3.6}$$

The first constraint (3.2) ensures that every vertex of the graph is in exactly one subset in the partition. The second constraint (3.3) enforces y_{uvi} to be 1 if vertices u and v are both in subset V_i (*i.e.* if (u,v) is a self edge) and 0 otherwise. The third constraint (3.4) is the capacity constraint.

3.4 Another ILP formulation

There is an IP formulation due to Calinescu *et al.* [8] for the multiway cut problem. We examine a similar formulation for our problem.

Let x_{ui} be a $0/1$ variable for each vertex $u \in V$ and for each subset V_i in the partition, which is set to 1 if the vertex $u \in V$ is in partition V_i. Another $0/1$ variable y_{uvi} is set to 1 if (u,v) is a cross edge with either $u \in V_i$ or $v \in V_i$ and set to 0 if it is a self edge. Therefore $\sum_{i=1}^{k} y_{uvi}$ returns 2 for every cross edge and 0 for every self edge of the partitions. d_{uv} is set to 1, if (u,v) is a self edge and set to 0, if it is a cross edge. The ILP is as follows :

$$\text{maximize} \sum_{(u,v)\in E} w(u,v) d_{uv} \tag{3.7}$$

$$\text{subject to} \sum_{i}^{k} x_{ui} = 1; \text{ for every vertex } u \in V. \tag{3.8}$$

$$y_{uvi} \geq x_{ui} - x_{vi}; \text{ for } (u,v) \in E \text{ and all } i \in [1..k]. \tag{3.9}$$

$$y_{uvi} \geq x_{vi} - x_{ui}; \text{ for } (u,v) \in E \text{ and all } i \in [1..k]. \tag{3.10}$$

$$d_{uv} = 1 - \frac{1}{2} \sum_{i=1}^{k} y_{uvi}; \text{ for } (u,v) \in E. \tag{3.11}$$

$$\sum_{u \in V} x_{ui} \leq s_i; \text{ for } i \in [1..k]. \tag{3.12}$$

$$x_{ui} \in \{0,1\}; \forall u \in V \text{ and } i \in [1..k]. \tag{3.13}$$

$$y_{uvi} \in \{0,1\}; \forall (u,v) \in E \text{ and } i \in [1..k]. \tag{3.14}$$

$$d_{uv} \in \{0,1\}; \forall (u,v) \in E \tag{3.15}$$

3.5 Linear Programming Relaxation

We call the integer linear program without the integrality constraints, the linear programming relaxation. So the linear program relaxation of an ILP is

$$\text{minimize } \bar{c}\bar{x}$$
$$\text{subject to } A\bar{x} = \bar{b}$$
$$x_i \geq 0$$

3.6 Integrality Gap

The *integrality gap* is the ratio between the optimal solution to the linear programming relaxation and the optimal solution to the integer linear program (for a maximization problem).

Theorem 3.1 : For an arbitrary graph, the linear programming relaxation of the IP in section 3.3 has the total number of edges $|E|$ as the optimal solution for the unit weighted case.

Proof: Consider a graph $G = (V, E)$ and we want to partition V into k subsets while maintaining the capacity constraints so as to maximize the number of self edges. Since $w(u, v) = 1;\ \forall\ (u, v) \in E$, the linear programming relaxation of the IP in section 3.3 as follows:

$$\text{maximize} \sum_{i=1}^{k} \sum_{(u,v)\in E} y_{uvi} \tag{3.16}$$

$$\text{subject to} \sum_{i=1}^{k} x_{ui} = 1; \text{ for every vertex } u \in V. \tag{3.17}$$

$$y_{uvi} \leq \frac{1}{2}(x_{ui} + x_{vi}); \text{ for } (u,v) \in E \text{ and } i \in [1..k]. \tag{3.18}$$

$$\sum_{u\in V} x_{ui} \leq s_i; \text{ for } i \in [1..k]. \tag{3.19}$$

$$x_{ui} \leq 1; \forall\, u \in V \text{ and } i \in [1..k]. \tag{3.20}$$

$$y_{uvi} \leq 1; \forall\, (u,v) \in E \text{ and } i \in [1..k]. \tag{3.21}$$

Given a partition, consider a cross edge (u,v) where $u \in V_i$ and $v \in V_j; j \neq i$. For this cross edge set $y_{uvi} = 1/2$ and $y_{uvj} = 1/2$ and for all other l, $y_{uvl} = 0$ where $i \neq j \neq l$. So for every cross edge $\sum_{i=1}^{k} y_{uvi} = 1$ and for every self edge (u,v) we get $y_{uvi} = 1$ where $u \in i$ and $v \in i$ and all other $j \neq i$, $y_{uvj} = 0$.

Therefore, for any arbitrary graph the objective function $\sum_{i=1}^{k} \sum_{(u,v)\in E} y_{uvi} = |E|$.

□

Theorem 3.2 : For an arbitrary graph, the linear programming relaxation of the IP in section 3.4 always returns the total number of edges $|E|$ as the optimal solution for the unit weighted case.

Proof: The linear programming relaxation of the IP in section 3.4 is as follows:

$$\text{maximize} \sum_{(u,v)\in E} d_{uv} \tag{3.22}$$

$$\text{subject to} \sum_{i=1}^{k} x_{ui} = 1; \text{ for every vertex } u \in V. \tag{3.23}$$

$$y_{uvi} \geq x_{ui} - x_{vi}; \text{ for } (u,v) \in E \text{ and } i \in [1..k]. \tag{3.24}$$

$$y_{uvi} \geq x_{vi} - x_{ui}; \text{ for } (u,v) \in E \text{ and } i \in [1..k]. \tag{3.25}$$

$$d_{uv} = 1 - \frac{1}{2}\sum_{i=1}^{k} y_{uvi}; \text{ for } (u,v) \in E. \tag{3.26}$$

$$\sum_{u\in V} x_{ui} \leq s_i; \text{ for } i \in [1..k]. \tag{3.27}$$

$$x_{ui} \leq 1; \forall\ u \in V \text{ and } i \in [1..k]. \tag{3.28}$$

$$y_{uvi} \leq 1; \forall\ (u,v) \in E \text{ and } i \in [1..k]. \tag{3.29}$$

$$d_{uv} \leq 1; \forall\ (u,v) \in E \tag{3.30}$$

Consider a partition of the vertices. In this relaxation, in subset V_i in the partition each vertex u is assigned equally and fractionally with value $\frac{s_i}{|V|}$. As vertices are assigned equally in each subsets so y_{uvi} is 0 for each edge (u,v) and for each subset V_i according to (3.24) and (3.25) of the program. For this reason d_{uv} is always 1 for any edge (u,v) according to (3.26). So $\sum_{(u,v)\in E} d_{uv}$ returns the total number of edges ($|E|$) as the value of the objective function for any arbitrary graph.

□

Theorem 3.3: The integrality gap of the linear programming relaxations of the integer programs of sections 3.3 and 3.4 is unbounded.

Proof: From Theorems 3.1 and 3.2 we know that both the linear programming relaxations of the IP in sections 3.3 and 3.4 return the total number of edges as the optimal solution for

any arbitrary unit weighted graph. So for a complete graph the integrality gap is unbounded as we now show.

Consider a complete graph and k subsets in the partition of equal capacity. The optimal solution to the integer linear program has value $\binom{\frac{|V|}{k}}{2}k$ where $|V|/k$ vertices are in each subset. In a complete graph the number of edges is $\binom{|V|}{2}$ which by the previous theorems is the optimal solution to the LP relaxation.

The integrality gap is

$$\frac{\binom{|V|}{2}}{\binom{\frac{|V|}{k}}{2} \cdot k} \tag{3.31}$$

That is

$$\frac{(|V|-1)k}{|V|-k} \tag{3.32}$$

So for $k = \frac{|V|}{2}$ the integrality gap is $|V|-1$.

□

The large integrality gap is indicative of the difficulty in obtaining a constant factor approximation algorithm using LP based approaches including the primal dual schema. Please refer to the excellent book by Vazirani [43] for the details of primal dual schema.

In the next chapter we discuss two local search algorithms and one recursive greedy algorithm for the capacitated max k-uncut problem.

Chapter 4

Approximation Algorithms

In the previous chapter we noted the difficulty in obtaining a constant factor approximation algorithm using linear programming. In this chapter we introduce some algorithms to approximate the capacitated max k-uncut problem. In section 4.1 we introduce the local search and the recursive greedy methods. In section 4.2 we introduce and analyze one simple local search algorithm based on swapping. We describe another local search algorithm based on an ejection chain in section 4.3. In section 4.4 we present one recursive greedy method to solve the capacitated max k-uncut problem and finally in section 4.5 we describe and analyze a recursive greedy method to solve the problem.

4.1 Local Search and Recursive Greedy methods

4.1.1 Local Search

Normally in a combinatorial optimization problem we have a set of elements S, called the *ground set* and our task is to arrange, group, order or select a subset of elements from S such that it optimizes the given function [26]. Some of the classical optimization problems include the traveling salesman problem, vertex cover problem and set cover problem.

Local search is a powerful technique to design approximation algorithms. It has been widely used for different optimization problems. Local search explores the space of all

possible solutions in a sequential manner until a locally optimal solution is found [28]. These types of algorithms start working from a candidate solution and move to a neighboring solution for a suitably defined neighbor in the search space. Normally every solution has more than one neighbor but the algorithm has to choose one neighbor to move to and this move is influenced by the information given about the solution in the neighborhood. The main idea of local search is: given a solution x from the set of candidate solutions for a combinatorial problem, local search tries to improve the value of the solution by making local changes to x. Local change might be adding elements from the ground set, deleting elements from x, changing the ordering of elements in x, or changing the way in which elements are grouped. If the solution improves after these changes then we get a new solution x'. We continue this step until no further improvement is possible.

We can put a bound on the number of iterations for the local search algorithm. Typically a local search algorithm terminates when it finds a locally optimal solution, that is when it cannot improve the value of the solution any more, or if it exceeds the time bound specified in the algorithm.

Local search algorithms have been successfully used for solving a large number of combinatorial problems like the traveling salesman, vertex cover, job scheduling etc. It has also been successfully used for different graph partitioning problems. Next we describe a application in graph cuts.

Kernighan *et al.* [33] describe a local search algorithm for uniform graph partitioning. In the uniform graph partitioning problem we are given an edge weighted graph $G = (V, E)$ and our task is to partition the vertices equally between two sets A and B such that the total weight of the cross edges is minimized. It is an important open problem to analyze the performance ratio of this algorithm theoretically. They showed empirically that the performance ratio of the algorithm is good.

4.1.2 Recursive Greedy method

The *greedy approach* is also a popular method to design approximation algorithms for optimization problems. The idea of the greedy approach is to build the solution incrementally. It selects the best partial solution in each iteration based on some simple criteria. If the partial solutions are computed by recursive calls then we call it recursive greedy [26]. Often a greedy approach does not give us the optimal solution but it can be used to get a good approximation bound.

Greedy methods have been used successfully in different problems like knapsack, job scheduling, tree vertex splitting [43].

4.2 Local Search Algorithm

4.2.1 Definition of Capacitated Max k-uncut problem

Given a non-negative edge weighted undirected graph $G = (V, E)$, an integer k and k capacities $s_1, \ldots, s_k$, where $\sum_{i=1}^{k} s_i = |V|$. Our goal is to partition the vertices into k subsets $V_1, \ldots., V_k$, where the ith subset V_i contains at most s_i vertices and the total weight of the self edges is maximized. Without loss of generality we assume that G is complete, missing edges in G can be considered as edges with weight 0.

4.2.2 The Swap Algorithm

Let $w(u, v)$ denote the weight of the edge $(u, v) \in E$ and $deg(u, V_i) = \sum_{(u,v) \in E, v \in V_i, v \neq u} w(u, v)$ denote the sum of the weights of the edges from a vertex u to the vertices in set V_i.

We start by partitioning the vertices into k sets, $V_1, \ldots, V_k$, arbitrarily assigning s_i vertices to set V_i, for all $i = 1, \ldots, k$.

In the algorithm we repeatedly determine a pair of vertices $u \in V_i$ and v $\in V_j$, $i \neq j$, for which

$$deg(u,V_i) + deg(v,V_j) < deg(u,V_j) + deg(v,V_i) - 2w(u,v) \quad (4.1)$$

If such a pair of vertices exists we reassign vertex u to V_j, and vertex v to V_i. We need to deduct $2w(u,v)$ from the right hand side of (4.1) because the edge between u and v before the swapping still remains a cross edge after swapping and it is counted twice, once for $deg(u,V_j)$ and a second time for $deg(v,V_i)$.

Upon termination of the algorithm, the following equation holds for all pairs $u \in V_i$ and $v \in V_j$ and all $i, j \in [1..k]$.

$$deg(u,V_i) + deg(v,V_j) \geq deg(u,V_j) + deg(v,V_i) - 2w(u,v), \text{ for all u} \in V_i \text{ and v} \in V_j \quad (4.2)$$

Please see section 5.1.1 for the runtime analysis of this algorithm.

4.2.3 Approximation Algorithm

In the following we analyze the worst case performance of the swap algorithm for all $k \geq 2$.

Theorem 4.1: The solution obtained using the swap algorithm has a value no smaller than $\frac{1}{d(k-1)+1}$ of the optimal solution value where k is the number of subsets in the partition and d is the ratio between the size of the largest and smallest subsets in the partition, assuming that the size of the smallest subset grows with the size of the graph.

Proof: Let us first consider the case of $(k = 2)$ two subsets V_1 and V_2 in the partition, each having the same sizes (capacities).

Upon termination of the algorithm the following condition holds:

$$deg(u,V_1)+deg(v,V_2) \geq deg(u,V_2)+deg(v,V_1)-2w(u,v), \text{ for all } u \in V_1 \text{ and } v \in V_2 \tag{4.3}$$

From the above equation, to get an upper bound on the total weight of all cross edges $(u,v) \in E, u \in V_1$ and $v \in V_2$, we consider a perfect matching (M) between the two partitions.

Summing (4.3) over all the edges in the perfect matching M we get

$$2S \geq 2C - 2W_M \tag{4.4}$$

Where S is the sum of the weights of the self edges, C is the sum of the weights of the cross edges and W_M is the minimum weight of perfect matching between V_1 and V_2.
We note that every self edge and cross edge is counted once for each of its end points (a total of twice).

The minimum weight perfect matching over all the matchings should be less than or equal to the average of all the perfect matchings. If the total number of vertices is $2n$ and each subset in the partition contains n vertices then the weight of the minimum perfect matching $W_M \leq C(n-1)!/n! \leq C/n$ where C is the weight of all the cross edges and $n!$ is the total number of perfect matchings over two subsets. We can now rewrite equation (4.4) as

$$S \geq C - \frac{C}{n} \tag{4.5}$$

That is

$$C \leq \frac{S}{1-\frac{1}{n}} \tag{4.6}$$

The optimal solution may contain all the edges as the self edges. So the performance ratio of the algorithm for two partitions each having the same number of vertices

$$p \geq \frac{S}{S+C} \tag{4.7}$$

$$\geq \frac{S}{S+\frac{S}{1-\frac{1}{n}}} \tag{4.8}$$

$$\geq \frac{1-\frac{1}{n}}{2-\frac{1}{n}} \tag{4.9}$$

If n is large enough then we can say that the performance ratio for two subsets in the partition of equal size is $\approx \frac{1}{2}$. Note that when $k = 2$ and the subset sizes are of the same size then the problem is NP-complete.

Now if the sizes of the subsets in the partition are not the same then we use the following procedure:

Procedure 4.1:

- Let $|V_2| > |V_1|$, without loss of generality assume that $|V_1|$ divides $|V_2|$ and let $\frac{|V_2|}{|V_1|} = d$ and let $|V_1| = n$.
- We mark all the vertices of V_2 as 0.
- We consider the first $|V_1|$ vertices of V_2 that are marked as 0 and sum up the inequality (4.3) for the minimum weight perfect matching that corresponds to these vertices and all the vertices of V_1 .

- We mark these vertices of V_1 that are considered in step 2 as 1.

- We continue step 2 until all the vertices of V_2 are marked 1.

After completing the above steps, *i.e*, summing up (4.4) over all the minimum perfect matchings we get

$$2dS_1 + 2S_2 \geq 2C - 2C/n \tag{4.10}$$

where we denote S_1 as the weight of the self edges of V_1 and S_2 as the weight of the self edges of V_2 and d is the ratio between the size of V_2 and V_1. Note that $d \geq 1$ and n is the total vertices of the smaller subset. Here we note that the self edges of the smaller subset are counted $2d$ times and the self edges in the larger partition are counted twice.

Suppose the minimum weight perfect matchings are $M_1, M_2, ...M_d$ and $C_1, C_2, ...C_d$ are the corresponding weights then we can say $\frac{2C}{n} = \frac{2C_1}{n} + ... + \frac{2C_d}{n}$.

Therefore we can write the above equation as

$$2d(S_1 + S_2) \geq 2C - 2C/n \tag{4.11}$$

As $2d(S_1 + S_2) \geq (2dS_1 + 2S_2)$

Let $S_1 + S_2 = S$ where S is the total weight of the self edges over both the subsets in the partition. So we can write

$$2dS \geq 2C(1 - 1/n) \tag{4.12}$$

We can write the performance ratio as

$$p \geq \frac{S}{S+C} \tag{4.13}$$

$$\geq \frac{S}{S+\frac{Sd}{1-\frac{1}{n}}} \tag{4.14}$$

$$\geq \frac{1-\frac{1}{n}}{d+1-\frac{1}{n}} \tag{4.15}$$

If n is large then we can say that

$$p \geq \frac{1}{d+1} \tag{4.16}$$

We now consider the problem for general k when $k \geq 2$.

We consider equation (4.10) for all possible pairs of subsets in the partition. If we sum up the equation (4.10) over all the subsets then the self edges are counted $(k-1)$ times and every cross edge is counted only twice. Therefore, if we assume $d = \frac{max\{|V_i|\}}{min\{|V_i|\}}$ and n is the number of vertices in the smallest subset in the partition.

$$(k-1)(2dS_i + 2S_j) \geq 2C - 2C/n \tag{4.17}$$

$$d(k-1)S \geq C(1-1/n) \tag{4.18}$$

$$S \geq \frac{C(1-1/n)}{d(k-1)} \tag{4.19}$$

S is the weight of the self edges returned by the algorithm and an optimal solution can

contain all the edges as the self edges.

So the performance ratio is

$$p \geq \frac{S}{S+C} \tag{4.20}$$

$$\geq \frac{S}{S+\frac{Sd(k-1)}{1-\frac{1}{n}}} \tag{4.21}$$

$$\geq \frac{1-\frac{1}{n}}{1-\frac{1}{n}+d(k-1)} \tag{4.22}$$

Now if n is large then the performance ratio is

$$\geq \frac{1}{d(k-1)+1} \tag{4.23}$$

□ [1]

Observation : The optimum solution of the capacitated max k-uncut problem for unit weighted version is the min $\{\ |E|, \sum_{i=1}^{k} \binom{s_i}{2}\}$ where $|E|$ denotes the total edges of the graph and s_i is the capacity of subset V_i.

Each subset V_i in the partition cannot contain more than $\binom{s_i}{2}$ edges if s_i is the capacity of the subset V_i. So the weight of the self edges in the graph is at most $\sum_{i=1}^{k} \binom{s_i}{2}$ edges. In the theorem 4.1 we use total edges $|E|$ as the optimal solution but for a dense graph $\sum_{i=1}^{k} \binom{s_i}{2}$ might be less than $|E|$. In such cases we can use $\sum_{i=1}^{k} \binom{s_i}{2}$ as the optimal solution and as $\sum_{i=1}^{k} \binom{S_i}{2} \leq |E|$ so we can get better performance ratio.

[1] I would like to thank Professor Ramesh Krishnamurti for extensive discussion on this proof.

4.3 Ejection Chain Algorithm

This algorithm has been inspired by the *ejection chain* method that has been used successfully for different optimization problems like traveling salesman, vehicle routing, crew scheduling etc [21]. Ejection chains generate complex compound moves. It generates a sequence of interrelated moves, that is, in every move it can change the states of one or more elements. We refer to the excellent chapter by Ahuja *et* in [3] for the details of the ejection chain method.

We perform a cyclic move of the vertices among the subsets in the partition if we can increase the total weight of the self edges of the vertices by this cyclic move.

This algorithm is similar to the algorithm due to Kernighan *et al.* [33] for the uniform min 2-cut problem.

Kernighan *et al.* [33] use swapping of elements between the two sets A and B. In their approach they initially randomly assign elements between two sets maintaining the uniformity constraint. In the first iteration we choose a pair of elements $a \in A$ and $b \in B$ such that if we swap these two vertices then we get the maximum increase in the weight of the self edges. Let the gain be g_1. Then we find another pair of vertices $a_1 \in A \setminus a_1$ and $b_1 \in A \setminus b_1$ that gives us the maximum gain considering that pair (a,b) is already swapped. In this way we consider all the pairs of vertices from the two partitions and calculate the gains. If the total number of vertices of the graph is $2n$ then we get $(a_1,b_1),.....,(a_n,b_n)$ pairs and a list of gains $g_1,....g_n$ for the corresponding pairs. Let $G(k) = \sum_{i=1}^{k} g_i$. We then consider $k \in [1..n]$ for which $G(k)$ is maximum and if the maximum is less than or equal to 0 then we stop the local search procedure, otherwise we swap the first k pairs of elements and start the procedure again.

The details of our ejection chain algorithm are described in the next section.

4.3.1 Ejection Algorithm

1. First we assume an order on the subsets in the partition. Suppose we have five subsets ranging from 1 to 5. We fix a random order of these subsets. For instance $2,3,1,4,5$.

2. We then find the maximum gain from some of the forward cycles given this order. We find the gain of a cycle by moving the vertices in the subsets cyclicly that are in that cycle. For example a cycle $C_i = (a,b,c)$ consists of three vertices and the order is $a \in p_3, b \in p_1, c \in p_2$. So we move vertex a of subset p_3 to p_1, vertex b from p_1 to p_2 and vertex c from p_2 to p_3, if we can improve the weight of the self edges overall. We pick the vertex from each subset which gives us the maximum gain in the weight of the self edges. That is if we have subset p_i and the next subset of the cycle is p_j then we pick $u \in p_i$ with $\max_u(deg(u,p_j) - deg(u,p_i))$ to be in the cycle. We consider cycles of length $2,3,4,..k$. A total of k cycles are considered for a given order of subsets in the partition.

3. We consider the cycle that returns the maximum gain that is, the cycle that give us the maximum increase in the weight of self edges and we shift the vertices among the subsets according to the order, if the gain is > 0.

4. Repeat step 2 until the maximum gain is ≤ 0.

Though we did not analyze the performance for this algorithm theoretically, we empirically study the algorithm for various sparse and dense graphs and the experimental results are discussed in chapter 5.

Note that the theoretical analysis of the performance ratio for a similar algorithm; due to Kernighan and Lin [33] for min cut is still an important open question.

4.4 Recursive greedy algorithm

4.4.1 Greedy method for max 2-uncut problem

First we consider the max 2-uncut problem and solve it using a greedy method. Let the two subsets in the partition be V_1 and V_2. Let $|V_1| = m$ and $|V_2| = n - m$. Now consider p solutions where $p = n/m$ (without loss of generality m divides n).

Theorem 4.2: There exists a $\frac{p-2}{p}$ approximation algorithm for the max 2-uncut problem where $p = \frac{|V_2|}{|V_1|}$ and $|V_2| \geq |V_1|$.

Proof: Consider a partition of $|V|$ with p subsets. Obtain a locally optimal solution using swap algorithm in section 4.2. We calculate the weight of the self edges considering V_i as a single subset and the rest $(V \setminus V_i)$ as the other subset in the partition. E_i denotes the weight of the self edges of the subset V_i and E_{ab} specifies the weight of the cross edges between V_a and V_b where $a \neq b$. So the maximum among these p solutions is at least the average of all the solutions. Consider V$_i$ and the rest that return the total weight of self edges is

$$= \sum_{i=1}^{p} E_i + \sum_{a,b\,:\,a<b\ \&\ a,b\ \neq i} E_{ab} \tag{4.24}$$

Next we use equation (4.24) to compute the average over p solutions. Every edge in E_{ab} where $a,\ b \neq i$ is counted twice as a cross edge once for V_a and once for V_b so the total number of self edges is at least

$$\geq p \sum_{i=1}^{p} E_i + (p-2) \sum_{a,b\,:\,a<b,\ b,a \neq i} E_{ab} \tag{4.25}$$

So the performance ratio is

$$\geq \frac{\sum_{i=1}^{p} E_i + \frac{(p-2)}{p} \sum_{a,b\,:\,a<b,\ b,a\neq i} E_{ab}}{\sum_{i=1}^{p} E_i + \sum_{a,b\,:\,a<b,\ b,a\neq i} E_{ab}} \tag{4.26}$$

Where $\sum_i E_i + \sum_{a,b\,:\,a<b\ \&\ a,b\ \neq i} E_{ab}$ is the total edges of the graph. So we can rewrite the equation as

$$\geq \frac{(p-2)}{p} \tag{4.27}$$

$\square$

This algorithm is effective for the case when the two subsets in the partition are highly unbalanced in size.

4.4.2 Recursive greedy method

Now we consider the general version of the capacitated max k-uncut problem where $k \geq 2$. We recursively solve using the following procedure with $k = 2$ (section 4.4.1) as a base case. We assume that $|V_1| \leq |V_2| \ldots \leq |V_k|$.

Recursive Greedy Algorithm

- We randomly assign vertices to p subsets in the partition and apply the swap algorithm described in section 4.2.2 to solve a max p-uncut problem where each subset has V_1 capacity and $p = \frac{|V|}{|V_1|}$.
- We take a subset V_i from p subsets in the partition and consider the rest of the subsets as a single subset that maximizes the weight of the self edges between these two subsets. The subsets are V_i and $V \setminus V_i$, so if E_i is the weight of the self edges of V_i

and $E^{'}$ is the total weight of the self edges of the other subsets $V_{j\neq i}$ then we pick that partition that maximizes E_i+α $E^{'}$ where α is the performance ratio for solving the rest of the partition $(V \setminus V_i)$.

- We recursively solve a max $(k-1)$-uncut problem over subsets V_j, $j \neq i$. Base case for this problem is $k = 2$ so we solve $k = 2$ problem by using the greedy method described section 4.4.1 or any of the algorithms described in sections 4.2 and 4.3. In the subproblem we have $k-1$ subsets and the value of p has changed (in step 1).

Analysis of the recursive greedy algorithm:

Theorem 4.3: In general the performance is $\prod_{i=1}^{k-1} \frac{(p_i-2)}{p_i}$ where $p_i = \lfloor \frac{|V \setminus \{V_1 \cup V_2 \cup \ldots \cup V_{i-1}\}|}{|V_i|} \rfloor$ and $|V_1| \leq |V_2| \ldots \leq |V_k|$.

Proof: Consider any i^{th} solution given by V_i and $V \setminus V_i$. Note that we solve the sub problem on the set $V \setminus V_i$ recursively. So the weight of the self edges in this solution is

$$E_i + \alpha(E - E_i - \sum_{i,j\,:\,i<j,\; j,i\neq i} E_{ij}) \tag{4.28}$$

Where E_i is the weight of the self edges of V_i and E_{ij} is the weight of the cross edges from V_i to edges in $V_{j\neq i}$.

Note that in the sub problem we are guaranteed to get α times of the total weight of all the edges in the subproblem as the weight of the self edges. So after summing the above equation over all i we get

$$\sum_{i=1}^{p} E_i + \alpha(p \cdot E - \sum_{i=1}^{p} E_i - \sum_{i,j\,:\,i<j,\; j,i\neq i} E_{ij}) \tag{4.29}$$

where $p = \frac{|V|}{min\{|V_i|\}}$

Let $\sum_{i=1}^{p} E_i = X$ and $\sum_{i,j \neq i} E_{ij} = 2Y$ where Y is the weight of all the cross edges across V_i and $V \setminus V_i$.

$$X + \alpha[p(X+Y) - X - 2Y] \tag{4.30}$$

As the maximum weight of self edges over all the possible p solutions is at least the average. So the maximum number of self edges is

$$\geq \frac{X + \alpha[p(X+Y) - X - 2Y]}{p} \tag{4.31}$$

we can rewrite this as

$$\geq \frac{X(1 + \alpha p - \alpha)}{p} + \frac{Y\alpha(p-2)}{p} \tag{4.32}$$

The optimal solution can contain almost all the edges, that is the weight is atmost $X+Y$ and if we consider $\frac{1+\alpha p-\alpha}{p} = a$ and $\frac{\alpha(p-2)}{p} = b$ then we can write the solution provided by this algorithm as $aX + bY$

so the performance ratio is

$$\frac{aX + bY}{X+Y} \tag{4.33}$$

$$\geq \min(a,b) \tag{4.34}$$

That is

$$pr \geq \min[\frac{1}{p} + \alpha(1 - \frac{1}{p}), \alpha(1 - \frac{2}{p})] \geq \alpha(1 - \frac{2}{p}) \tag{4.35}$$

In general the bound is $\prod_{i=1}^{k-1} \frac{(p_i - 2)}{p_i}$.

$\square$

In the next chapter we discuss the experimental results of the algorithms.

Chapter 5

Experiments and Results

In this chapter we present the experimental evaluations of the algorithms for the unit weighted version of the capacitated max k-uncut problem. In section 5.1 we discuss the implementation details of the algorithms, in section 5.2 we briefly describe the data sets and finally in section 5.3 we present the experimental results.

5.1 Implementation

We use Python 2.5 for implementing the three algorithms described in chapter 4. All experiments presented in this chapter were conducted on a 2.7 GHz Pentium 4, 64 bit processor with 1 GB RAM in the Windows XP environment.

The basic data structures that we use are lists and lists of lists to implement the algorithms.

- We maintain a list called *adjacent* to store the adjacency list of a vertex. *adjacent[i]* contains the list of vertices that are adjacent to the vertex i.
- We use a list of lists called *graph* to store the subsets in the partition. $graph[k]$ stores the list of vertices that are in subset V_k.
- We use another list of lists called *neighbour*. $neighbour[i][k]$ denotes the list of vertices that are in subset V_k and adjacent to vertex i.

The basic functions that are used by the algorithms are:

- *degree*(u, V_i): It is used to compute the total weight of the edges from a vertex u on to subset V_i. It takes a vertex u and a subset V_i as the arguments and returns the weight of the edges from u incident on the vertices in V_i. It uses *neighbour*$[u][V_i]$ to calculate the weights.

- *update neighbour* (*adjacent[u]*,V_i, V_j): If we swap the vertices then we update the neighbour list of those vertices that are adjacent to the swapped vertices. If one of the swap vertices is u and has been moved from V_i to V_j then this function is called to update the neighbour list of those vertices that are adjacent to u. This greatly improves the running time of the degree function.

5.1.1 Swap Algorithm

In the *swap algorithm* we first randomly assign vertices among $V_1, V_2, \ldots, V_k$ subsets. For each pair of vertices (u, v) where $u \in V_i$ and $v \in V_j$ and $i \neq j$ we use the swap step (described in page 27). In each local step we use *degree*(u, V_i) to calculate the degree and after each swap we call *update neighbour* (*adjacent[u]*,V_i, V_j) function.

Analysis:

For the general case with positive integral weights the runtime analysis is as follows: We denote w_e as the weight of an edge $e \in E$ and $\sum_{e \in E} w_e = W$. The *swap algorithm* can start with a total weight of the self edges as 0 and can iterate for every pair of vertices of every pair of subsets in the partition, and in each iteration it will improve the weight of the sum of the self edge by at least 1, so the running time of the algorithm is $O(k^2 \cdot n^2 \cdot W)$ where k is the number of subsets and n is the number of vertices in the graph.

5.1.2 Recursive Greedy method

Analysis:

Recursive greedy method recursively calls *swap algorithm* for $k-1$ times where k is the total number of partitions. Recall from the anaysis of recursive greedy, $|V_1| \leq |V_2| ... \leq |V_k|$. In the *recursive greedy method* we initially get p solutions for V_1 where $p = \frac{|V|}{|V_1|}$. In this step the time taken is $O(p^2 \cdot n^2 \cdot W)$. Furthermore there are k recursive calls so the total time taken is almost $O(k \cdot p^2 \cdot n^2 \cdot W)$ This analysis applies to the general case with positive integral edge weight.

The program designed for the *swap algorithm* and *recursive greedy method* contains almost 1100 lines of code.

5.1.3 Ejection Algorithm

In the *ejection algorithm* we assume a random order of the partitions and calculate the degree of the vertices that are in a cycle in that order. We use $degree(u, V_i)$ to calculate the degree. We then swap the vertices of that cycle that gives us the maximum gain (if positive) and call the $update$ *neighbour* (*adjacent[u]*,V_i, V_j) function to update the neighbour lists of the vertices that are adjacent to the swapped vertices.

Analysis:

All the k cycles for a fixed ordering of subsets can be discovered in $O(n^2)$ time therefore the running time is $O(n^2 W)$. If we choose c random ordering of the subsets then the running time is $O(cn^2 W)$.

Remarks: The algorithms run in pseudo polynomial time in the general case and for the unit weighted case run in polynomial time. For the general case we can get an $(1-\varepsilon)$ approximate solution in time that is a polynomial in the input size and $1/\varepsilon$ (see [38]).

The implementation of the *ejection algorithm* contains 700 lines of code, with code reuse from the above two algorithms.

5.2 Data Sets

We evaluated the algorithms on the following data sets.

- *Protein interaction database*: The protein protein interaction database contains the data about the protein protein interaction.

  Each row of the databases contains the information about the pair of proteins (with *protein id*) that interacts. We consider every protein as a vertex and put an edge between two proteins if they interact. We set the weight of the edge to 1. All the non-edges are considered as edges with weight 0.

- *Random Graph*: We construct some random sparse and dense graphs for experiments. The procedure to construct the random graphs is described in section 5.3.2.

5.3 Experimental results

5.3.1 Protein Interaction Database

We ran the algorithms described in chapter 4 on the two protein protein interaction databases from [29]. First consider $k = 2,..,20$ subsets of equal size. Initially we randomly partition the graph into k subsets and run the *swap*, *ejection* and *greedy* algorithms on these graphs and do this for 30 random start points. The first database consists of 1476 vertices and

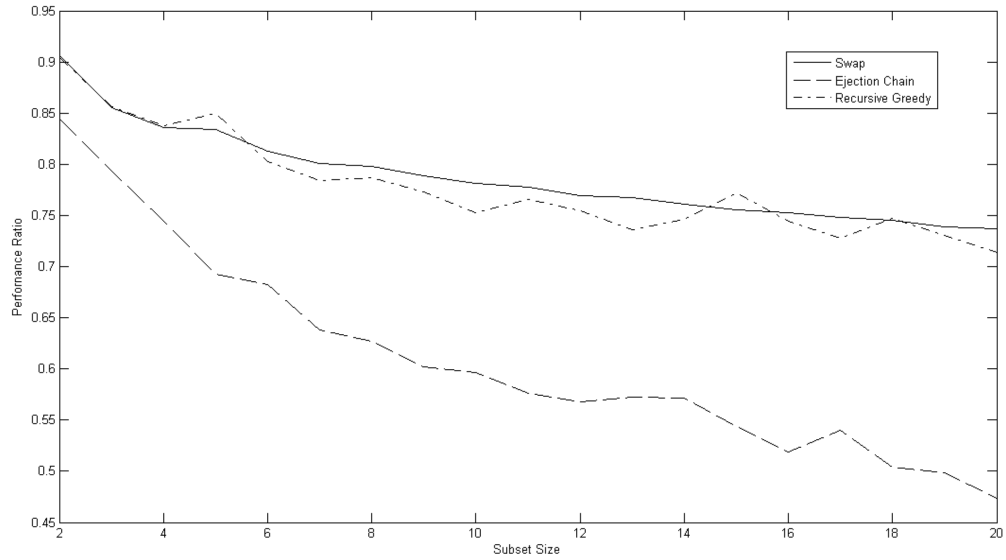

Figure 5.1: Comparison of the performance ratio of the algorithms on database 1 (uniform sizes)

3026 edges and the second database contains 2633 vertices and 3967 edges. The number of subsets in the partition (k), average performance over 30 runs and the average time the algorithms take to obtain the solution is illustrated in tables 5.1 and 5.2. The optimal solution is the minimum of $|E|$ and $\sum_{i=1}^{k} \binom{s_i}{2}$ where E is the total number of edges of the graph and s_i is the size of V_i.

The performance ratio and the time taken by the algorithms on the graphs arising from the protein interaction database is described in tables 5.1 and 5.2.

Figure 5.1 and Figure 5.3 depict partition size vs. the performance ratio of the algorithms graphically.

In figures 5.2 and 5.4 we compare the time taken by the algorithms to solve the problem. The experimental results on both databases of protein protein interaction show that the performance of the swap algorithm and the recursive greedy method is almost the same.

Table 5.1: Experimental results on database 1 (uniform sizes)

k	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
2	0.90	42.26	0.84	96.55	0.90	78.81
3	0.85	63.90	0.79	271.03	0.85	176.26
4	0.83	67.31	0.74	417.14	0.83	228.93
5	0.83	81.29	0.69	219.71	0.84	278.42
6	0.81	81.79	0.68	278.35	0.80	386.73
7	0.80	90.55	0.63	380.84	0.78	389.53
8	0.79	112.20	0.62	483.98	0.78	421.43
9	0.78	84.29	0.60	478.35	0.77	480.04
10	0.78	81.17	0.59	586.27	0.75	538.03
11	0.77	90.30	0.57	702.16	0.76	567.19
12	0.76	78.16	0.56	955.53	0.75	689.17
13	0.76	85.99	0.57	1003.95	0.73	709.19
14	0.76	86.20	0.57	1215.43	0.74	783.55
15	0.75	87.36	0.54	884.068	0.77	870.78
16	0.75	88.01	0.51	862.821	0.74	962.48
17	0.74	89.98	0.53	1073.03	0.72	1001.64
18	0.74	85.93	0.50	748.00	0.74	998.76
19	0.73	86.92	0.49	784.70	0.73	1032.91
20	0.73	89.30	0.47	787.47	0.71	1023.64

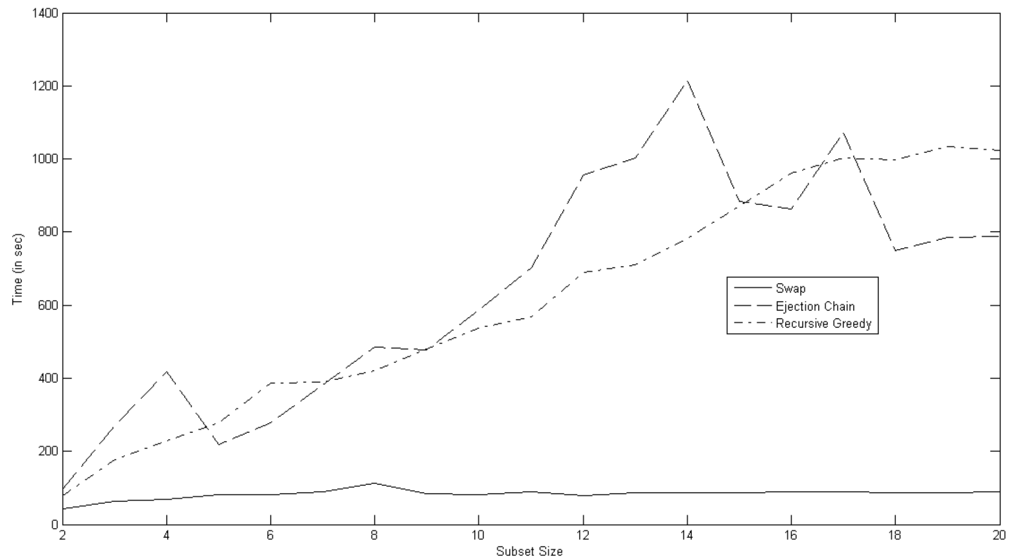

Figure 5.2: Comparison of the timing of the algorithms on database 1 (uniform sizes)

The *recursive greedy method* works the same as the *swap algorithm* but it uses the *swap algorithm* $k-1$ times so it takes much more time than the *swap algorithm*. On the other hand the *ejection algorithm* takes much more time than the *swap algorithm* and the *recursive greedy method* as it considers longer cycles than the other two algorithms in a single step of iteration and it has worse performance ratio than the other two algorithms because it works with a single fixed random ordering of the subsets in the partition. If we can consider all the possible orderings of the subsets then we can improve the performance ratio for the algorithm but it takes much more time. These results with more orderings are reported for sparse graphs later (Figures 5.9 and 5.10).

Now we consider the unbalanced subsets in the partition. In the first database we consider two subsets; the size of the first subset is 50 and the second subset contains the rest of the vertices of the graph. We then run the algorithms on these subsets. We do this similarly for the cases where the size of the first subsets are from $100, 150, ..., 700$ and the second

Table 5.2: Experimental results on database 2 (uniform sizes)

k	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
2	0.85	109.70	0.84	602.55	0.85	228.97
3	0.79	177.73	0.72	939.42	0.77	467.31
4	0.75	202.50	0.63	1615.80	0.73	637.51
5	0.72	205.98	0.58	2100.87	0.71	786.28
6	0.71	360.52	0.56	2715.37	0.69	966.98
7	0.69	353.85	0.54	2902.63	0.68	1130.26
8	0.67	375.87	0.52	1267.31	0.68	1299.98
9	0.67	394.38	0.51	1420.39	0.66	1360.07
10	0.65	326.32	0.48	2283.90	0.65	1593.14
11	0.65	221.73	0.47	1638.57	0.63	1934.36
12	0.64	230.38	0.47	1818.85	0.62	2080.15
13	0.63	223.38	0.47	1975.25	0.62	2129.36
14	0.63	231.90	0.43	2255.33	0.62	2211.85
15	0.63	299.26	0.44	2046.45	0.62	2295.36
16	0.62	425.79	0.42	2329.97	0.60	2480.59
17	0.62	342.24	0.44	3072.12	0.59	2660.97
18	0.61	342.06	0.41	2737.05	0.60	2915.62
19	0.61	407.41	0.43	3099.41	0.60	3237.95
20	0.60	249.45	0.42	3022.26	0.60	3479.29

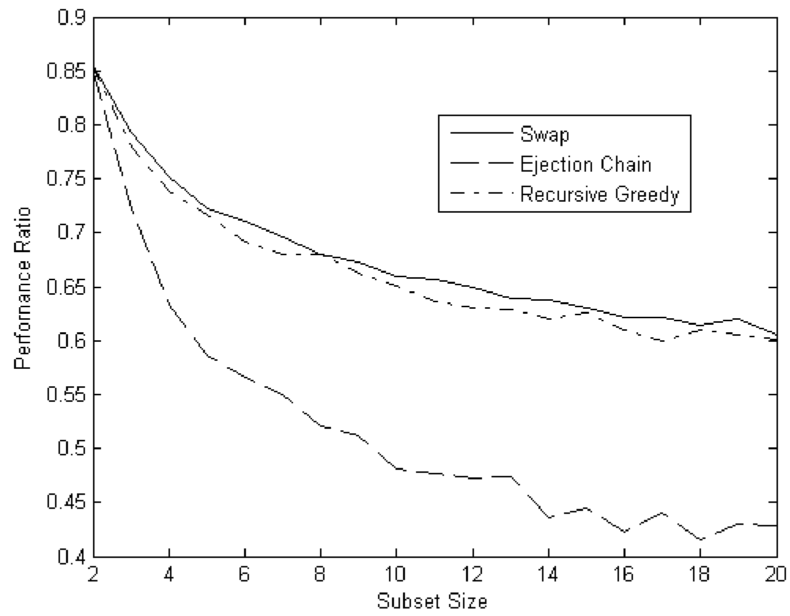

Figure 5.3: Comparison of the performance ratio of the algorithms on database 2 (uniform sizes)

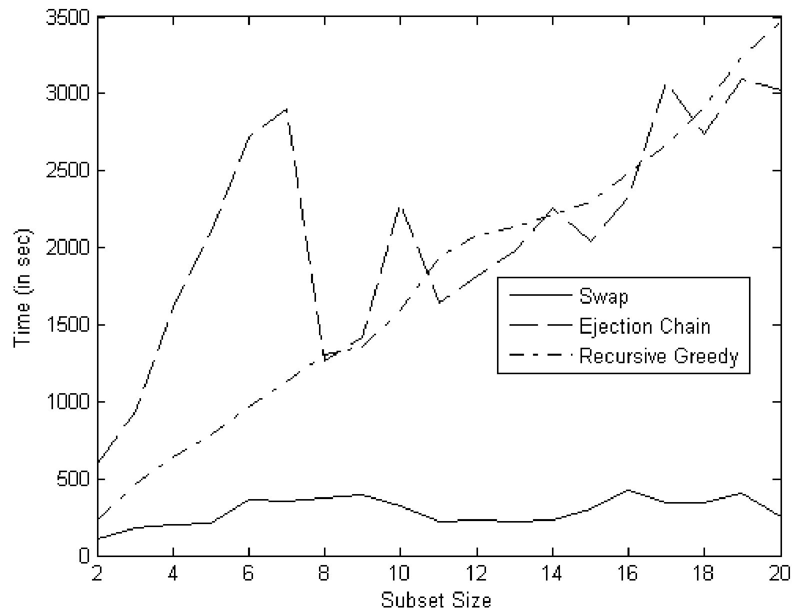

Figure 5.4: Comparison of the timing of the algorithms on database 2 (uniform sizes)

Table 5.3: Experimental results on database 1 (two unbalanced subsets)

Size of the first subset	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
50	0.99	3.71	0.99	20.37	0.96	73.86
100	0.99	7.12	0.98	23.35	0.95	88.66
150	0.98	10.44	0.98	35.34	0.92	68.57
200	0.98	3.55	0.97	45.2	0.91	64.92
250	0.97	16.32	0.96	56.27	0.91	66.75
300	0.97	23.76	0.95	73.89	0.91	65.14
350	0.97	32.54	0.95	86.51	0.90	62.25
400	0.96	29.97	0.91	70.36	0.89	58.90
450	0.95	33.25	0.92	111.58	0.89	58.32
500	0.93	35.90	0.92	122.73	0.90	55.04
550	0.92	37.47	0.87	124.39	0.89	51.78
600	0.91	46.89	0.90	137.08	0.89	48.70
650	0.90	53.44	0.91	145.7	0.89	45.84
700	0.90	52.31	0.89	163.65	0.90	39.54

subsets contains the remaining vertices of the graphs.

Table 5.3 shows the performance ratio and the time taken by the algorithms on these instances.

From these experiments we can observe that the performance ratio of the algorithms are more or less the same. Here the *ejection algorithm* shows performance close to the other two algorithms because there is only one order for the two partitions, so it considers all the cycles of the two partitions.

Similarly we do some experiments on the second database and the result of the experiments is described in Table 5.4

We now consider three unbalanced subsets, where we fix the size of the first subset and change the size of the second subset. For both databases the size of the first subset is 100.

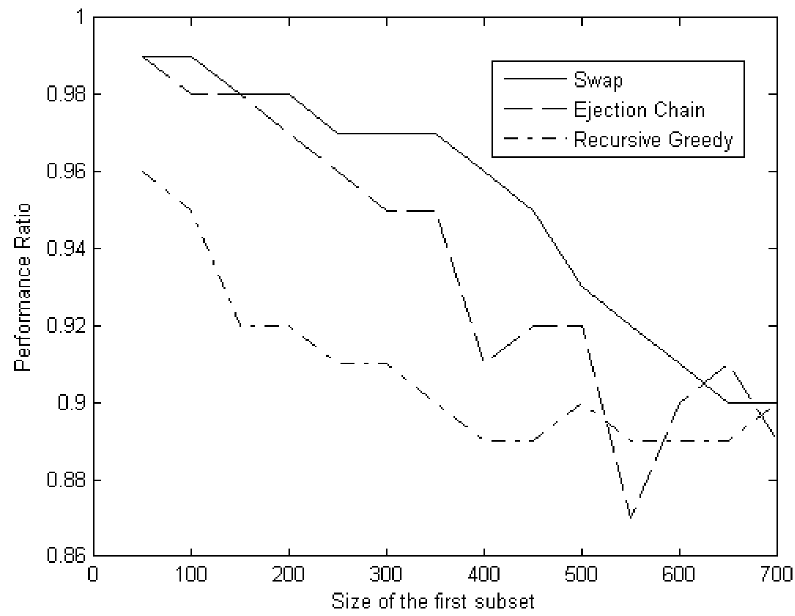

Figure 5.5: Comparison of the performance ratio of the algorithms on unbalanced subsets of database 1

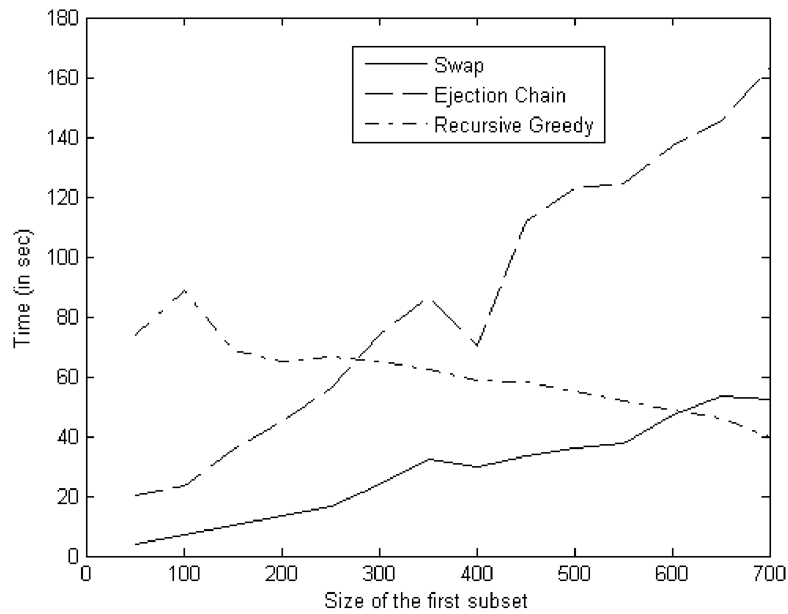

Figure 5.6: Comparison of the timing of the algorithms on unbalanced subsets of database 1

Table 5.4: Experimental results on database 2 (two unbalanced subsets)

Size of the first subset	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
50	0.99	7.23	0.99	56.33	0.94	200.86
100	0.98	12.46	0.99	78.12	0.95	197.66
150	0.97	17.98	0.98	115.36	0.92	194.26
200	0.96	24.49	0.97	144.18	0.91	185.66
250	0.96	31.74	0.96	186.57	0.91	176.32
300	0.95	37.11	0.95	228.1	0.90	183.49
350	0.94	40.57	0.94	285.21	0.89	191.64
400	0.93	58.69	0.94	348.19	0.87	201.38
450	0.92	56.13	0.92	341.28	0.85	176.33
500	0.92	74.87	0.92	350.77	0.85	169.41
550	0.91	103.29	0.90	362.88	0.84	173.57
600	0.91	109.35	0.89	285.45	0.83	185.37
650	0.90	108.42	0.90	533.03	0.81	180.55
700	0.90	100.21	0.89	563.62	0.83	187.67
750	0.89	96.411	0.87	468.2	0.81	176.75
800	0.89	116.04	0.87	535.93	0.81	175.56
850	0.88	115.41	0.86	616.24	0.82	168.14
900	0.88	117.10	0.87	657.46	0.82	161.85
950	0.87	124.90	0.85	710.47	0.81	167.84
1000	0.87	136.39	0.86	903.54	0.81	210.31
1050	0.86	128.94	0.85	777.28	0.81	300.28
1100	0.86	148.18	0.86	981.68	0.82	212.78
1150	0.86	145.64	0.85	885.35	0.83	136.09
1200	0.85	161.39	0.84	1094.96	0.83	139.22
1250	0.86	163.58	0.84	400.83	0.84	113.94
1300	0.85	136.56	0.84	945.95	0.83	100.41

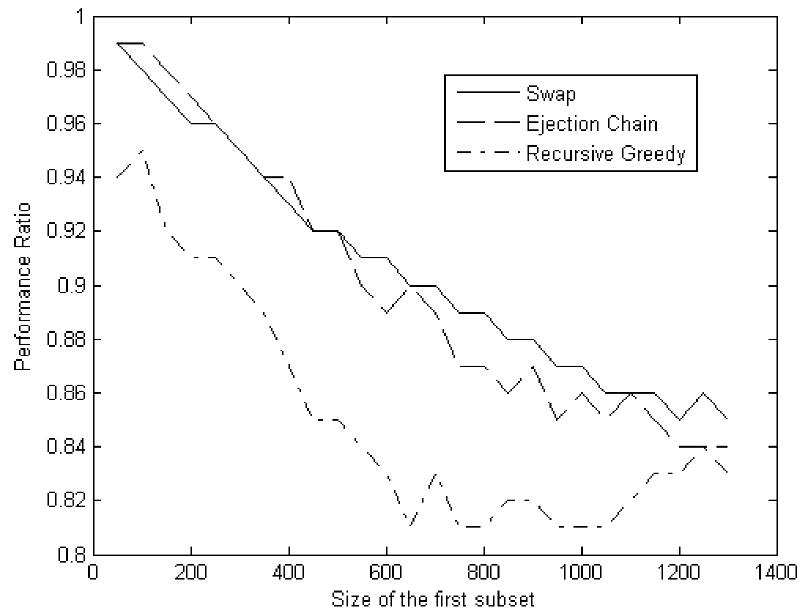

Figure 5.7: Comparison of the performance ratio of the algorithms on two unbalanced subsets of database 2

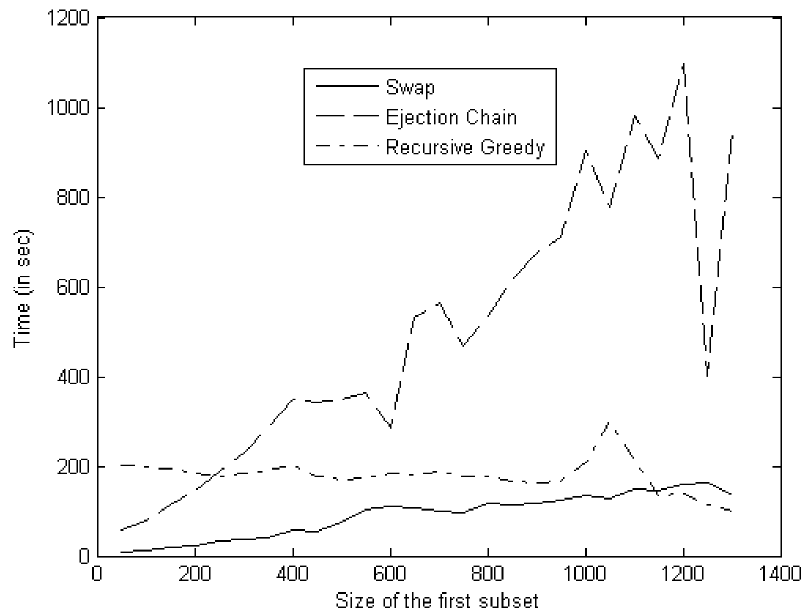

Figure 5.8: Comparison of the timing of the algorithms on two unbalanced subsets of database 2

Table 5.5: Experimental results on database 1 (three unbalanced subsets)

Size of the second subset	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
100	0.98	18.44	0.95	57.54	0.90	151.5
200	0.97	35.73	0.93	79.41	0.88	125.4
300	0.96	49.69	0.90	114.51	0.87	125.5
400	0.94	58.16	0.86	135.16	0.86	122.7
500	0.93	64.68	0.75	152.59	0.86	121.3
600	0.90	54.80	0.88	184.32	0.86	107.7
700	0.88	54.24	0.88	193.79	0.87	98.75
800	0.90	51.25	0.88	195.76	0.89	99.34
900	0.92	46.45	0.77	157.84	0.91	94.19
1000	0.94	40.97	0.89	114.15	0.92	105.26
1100	0.95	29.24	0.93	90.24	0.93	101.93
1200	0.96	22.89	0.95	74.81	0.94	101.73
1300	0.98	13.34	0.95	42.92	0.95	100.71

The results of the experiments are described in tables 5.5 and 5.6.

From the tables 5.5 and 5.6 we find that the performance ratio of the algorithms are almost same for the different three subsets in the partition but the *ejection algorithm* and the *recursive greedy method* take much more time than the *swap algorithm*.

5.3.2 Randomly Generated Graphs

We also generate some random graphs and run experiments on those graphs. We generate a random graph using the following steps:

1. We specified the total number of vertices ($|V|$) of the graph.

2. We make $|V|/5$ subsets in the partition each subset contains 5 vertices.

3. pr is the probability of an edge being present.

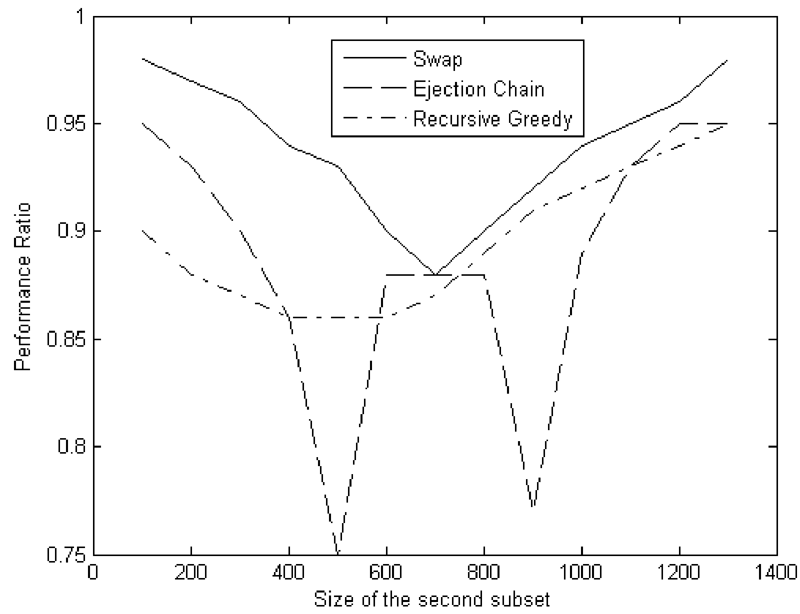

Figure 5.9: Comparison of the performance ratio of the algorithms on three unbalanced subsets of database 1

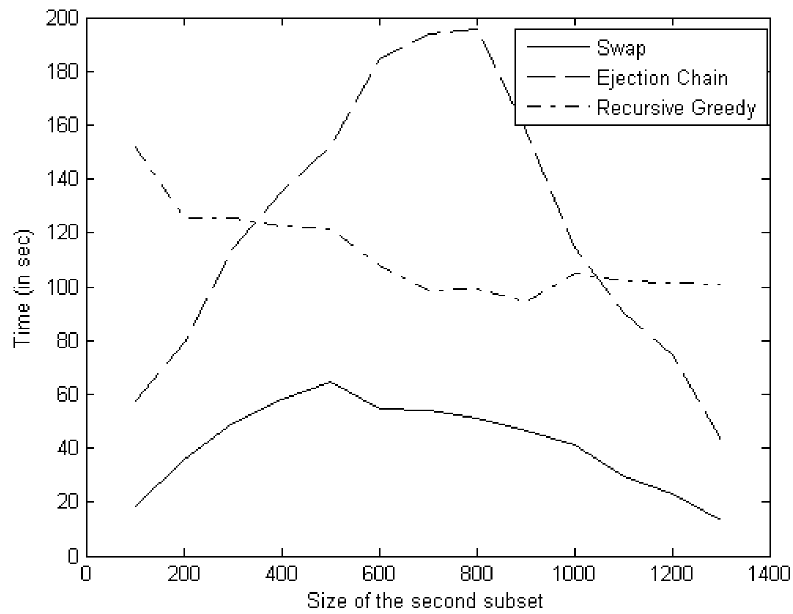

Figure 5.10: Comparison of the timing of the algorithms on three unbalanced subsets of database 1

Table 5.6: Experimental results on database 2 (three unbalanced partitions)

Size of the second subset	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
100	0.96	43.14	0.97	177.57	0.92	922.38
200	0.94	73.84	0.96	204.47	0.89	1041.43
300	0.92	90.49	0.94	315.04	0.84	1076.45
400	0.90	106.35	0.92	407.06	0.79	895.85
500	0.89	118.92	0.89	369.02	0.80	773.42
600	0.87	133.68	0.89	467.71	0.82	1032.39
700	0.88	146.02	0.85	667.41	0.78	917.57
800	0.87	155.72	0.84	723.0	0.80	727.77
900	0.85	162.89	0.85	678.41	0.77	842.39
1000	0.84	173.24	0.85	791.11	0.74	831.91
1100	0.84	177.23	0.76	749.86	0.78	608.84
1200	0.83	184.02	0.86	766.89	0.77	597.7
1300	0.82	181.08	0.76	743.94	0.81	662.59
1400	0.80	177.54	0.84	980.23	0.78	577.65
1500	0.81	169.95	0.82	812.81	0.81	719.07
1600	0.82	164.68	0.83	629.38	0.83	580.59
1700	0.83	157.89	0.85	531.86	0.82	717.83
1800	0.85	147.32	0.85	521.76	0.85	715.41
1900	0.87	134.7	0.89	428.32	0.87	535.82
2000	0.88	86.78	0.90	358.43	0.88	691.2
2100	0.90	109.3	0.92	440.43	0.90	268.23
2200	0.93	68.55	0.92	319.24	0.91	324.53
2300	0.94	57.29	0.94	267.09	0.89	258.5
2400	0.95	62.92	0.96	166.80	0.89	239.17
2500	0.97	34.91	0.98	126.31	0.96	293.12

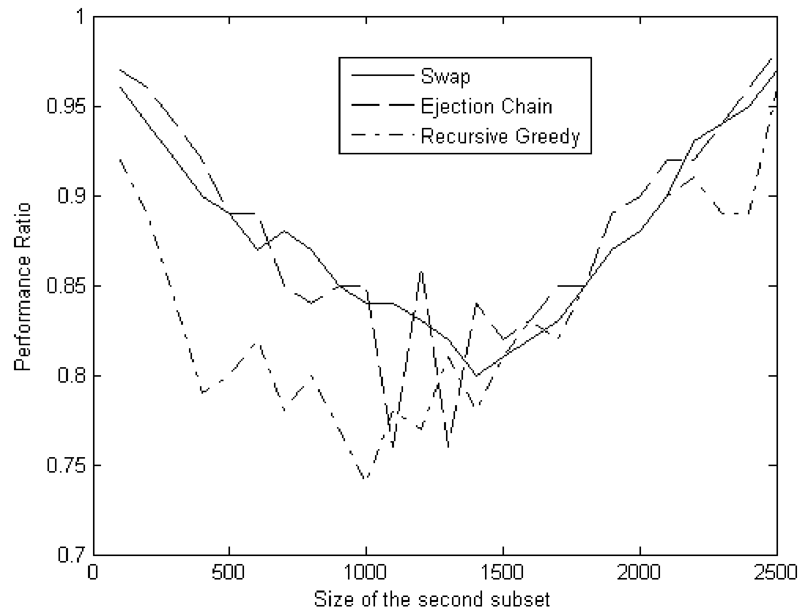

Figure 5.11: Comparison of the performance ratio of the algorithms on three unbalanced subsets of database 2

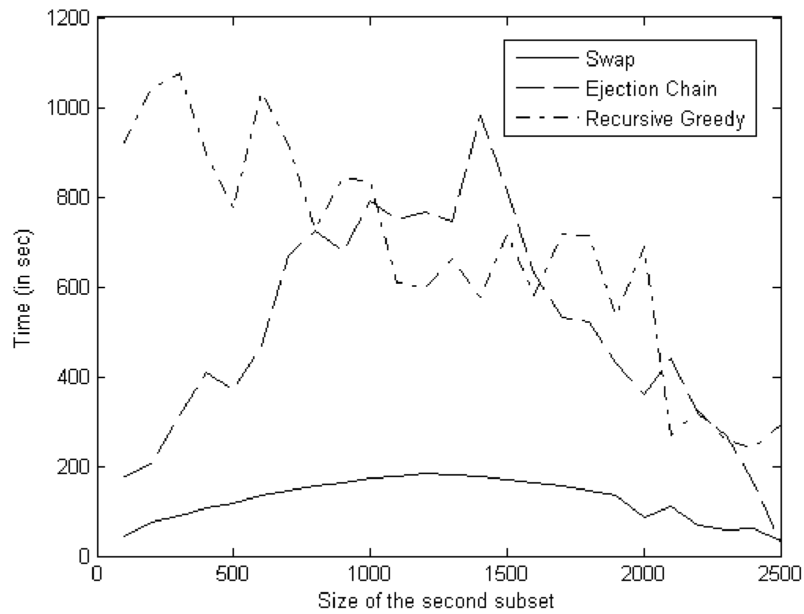

Figure 5.12: Comparison of the timing of the algorithms on three unbalanced subsets of database 2

Table 5.7: Experiments on random dense graphs

Total Edges	Total Vertices	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
596	50	0.92	0.11	0.81	0.12	0.91	0.46
2447	100	0.97	0.46	0.85	1.43	0.95	3.31
5628	150	0.99	1.29	0.85	5.05	0.97	12.08
9984	200	0.99	2.34	0.86	16.77	0.97	28.87
15589	250	0.99	3.31	0.85	37.62	0.98	55.02
22516	300	0.99	4.49	0.86	76.99	0.98	98.24
30553	350	0.99	6.06	0.86	147.80	0.99	167.17
40060	400	0.99	10.28	0.85	216.23	0.98	255.14
50649	450	0.99	10.40	0.86	419.12	0.99	367.06
62726	500	0.99	12.70	0.86	546.46	0.99	531.08

4. For every pair of vertices we generate a random number r, between 0 to 1. If r is less than or equal to pr then we put an unit weighted edge between these two vertices.

The average performance ratio of the three algorithms for these random graphs with the average time taken to get the optimal solution is described in Table 5.7. Here we consider $pr = 0.5$ for which the graph is dense.

Figures 5.5 and 5.6 illustrate the comparison of the performance and the time of the algorithms.

It is no surprise that performance ratios of the algorithms are quite good in this experiment because as the graph is dense so $\sum_{i=1}^{k} \binom{s_i}{2}$ is much less than the total number of edges. Therefore we decide to conduct some experiments on sparse graphs with the probability of being an edge is $\frac{5}{|V|}$.

The results of the experiments on the sparse graphs are described in Table 5.8.

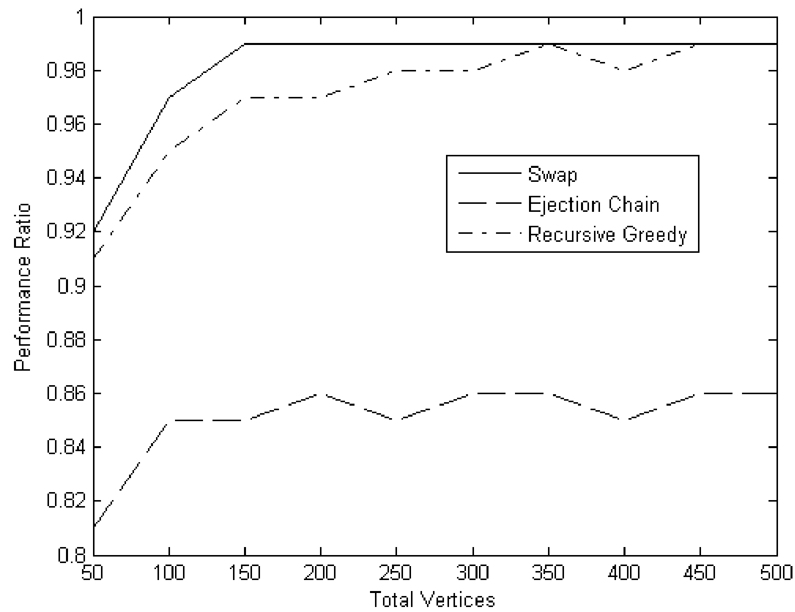

Figure 5.13: Comparison of the performance ratio of the algorithms on random dense graphs

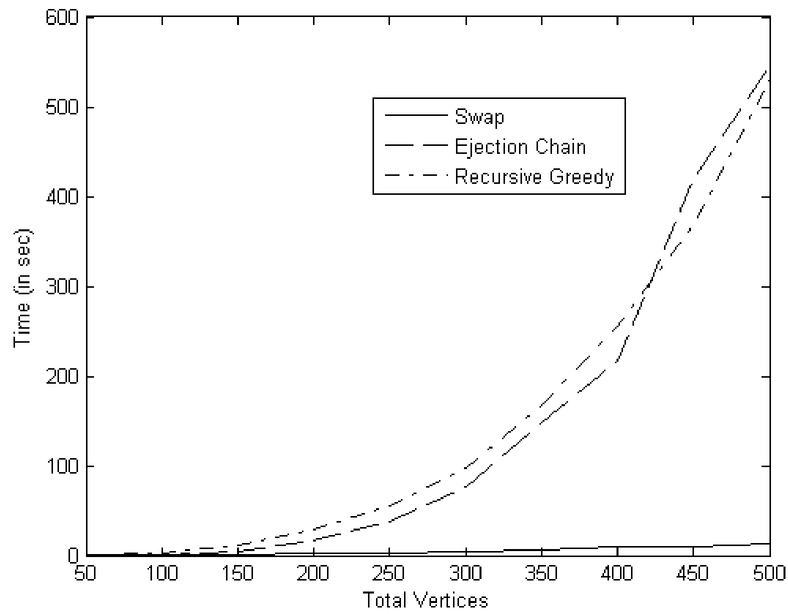

Figure 5.14: Comparison of the timing of the algorithms on random dense graphs

Table 5.8: Experiments on random sparse graphs with $p = \frac{5}{|V|}$

Total Edges	Total Vertices	Swap algorithm (perfo -rmance ratio)	Swap algorithm (time in sec)	Ejection algorithm (perfo -rmance ratio)	Ejection algorithm (time in sec)	R.G algorithm (Perfor -mance ratio)	R.G algo -rithm (time in sec)
311	50	0.72	0.24	0.56	0.24	0.69	0.6
608	100	0.59	0.86	0.39	2.05	0.56	5.0
932	150	0.56	1.87	0.33	9.58	0.53	17.54
1214	200	0.51	3.19	0.28	17.89	0.50	41.0
1453	250	0.51	4.95	0.26	49.51	0.48	78.28
1769	300	0.46	6.94	0.23	86.51	0.47	138.5
1938	350	0.45	9.17	0.23	184.41	0.46	205.37
2465	400	0.48	12.24	0.23	279.74	0.46	306.07
2741	450	0.45	15.22	0.19	449.41	0.45	444.79
3058	500	0.44	18.81	0.20	680.34	0.44	624.01
3339	550	0.44	22.66	0.20	1233.33	0.45	821.11
3619	600	0.44	27.28	0.18	1426.58	0.43	1052.28
3836	650	0.44	31.45	0.18	1555.5	0.43	1388.79
4143	700	0.42	36.64	0.18	1698.73	0.42	1713.38
4589	750	0.44	42.23	0.18	1741.9	0.44	2143.71
4789	800	0.43	47.72	0.17	1777.85	0.43	2520.79
5010	850	0.40	52.33	0.16	1920.20	0.42	3218.28
5400	900	0.42	60.04	0.16	2112.37	0.42	3744.60
5717	950	0.41	66.82	0.14	2132.68	0.42	4111.27
5934	1000	0.40	72.56	0.14	2217.00	0.41	4629.61

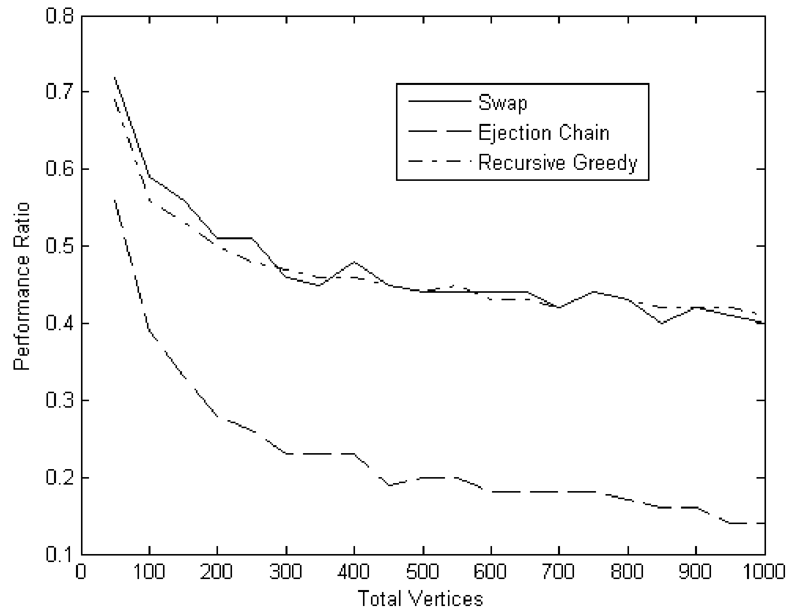

Figure 5.15: Comparison of the performance ratio of the algorithms on random sparse graphs with $p = \frac{5}{|V|}$

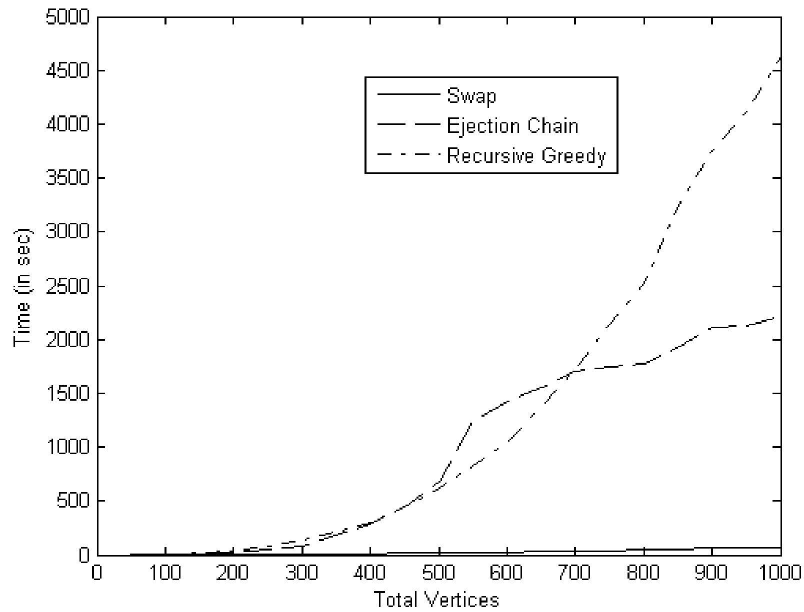

Figure 5.16: Comparison of the timing of the algorithms on random sparse graphs with $p = \frac{5}{|V|}$

The experiments on the sparse graphs show that the *swap algorithm* and *recursive greedy algorithms* have comparable performance ratio but the *recursive greedy method* takes much more time than the *swap algorithm* as it has to call the *swap algorithm* once for each partition. The performance of the *ejection algorithm* is worse than the other two but we can increase the performance ratio of the *ejection algorithm* by considering more orderings.

We then consider more orderings of the subset in the partition for the *ejection algorithm* to solve the problem on some small sparse graphs. In these experiments we consider $20k$ and $100k$ orderings of the partitions where k is the total number of partitions and we get a better performance ratio for the algorithm which is close to the *swap algorithm* but it takes much more time than the previous as it considers more orderings.

The results of the experiments in comparison to the *swap algorithm* are illustrated in Table 5.9

5.4 Conclusions

In this chapter we experimentally study the performance ratio of the three algorithms described in chapter 4 on two protein protein interaction databases and on some random dense and sparse graphs.

From the results we find that all the algorithms show almost the similar performance ratio for the balanced and the unbalanced subsets for the protein protein interaction databases but the *recursive greedy method* and the *ejection algorithm* take more time than the *swap algorithm*. The *ejection algorithm* shows worse performance than the other two algorithms in the sparse graph if we consider only one ordering of the subsets in the partition but we can improve considaribly the performance ratio of the algorithm by considering more orderings.

Table 5.9: Experiments on random small sparse graphs with $20k$ orderings with $p = \frac{5}{|V|}$

Total Edges	Total Vertices	Swap algorithm (performance ratio)	Swap algorithm (time in sec)	Ejection algorithm (performance ratio)	Ejection algorithm (time in sec)
106	20	0.9	0.04	0.88	0.1
169	30	0.8	0.06	0.87	0.36
232	40	0.83	0.16	0.80	0.91
278	50	0.69	0.15	0.72	1.87
352	60	0.66	0.21	0.68	3.62
405	70	0.64	0.42	0.64	6.13
511	80	0.64	0.55	0.67	10.86
515	90	0.57	0.46	0.62	14.81
604	100	0.65	0.84	0.63	24.11
672	110	0.56	1.01	0.60	42.56
716	120	0.56	1.17	0.62	52.39
795	130	0.57	1.54	0.59	67.45
843	140	0.56	2.07	0.56	82.49
937	150	0.55	1.83	0.59	111.7
897	160	0.52	2.0	0.55	134.4
995	170	0.51	2.44	0.54	173.16
1082	180	0.53	2.53	0.54	214.07
1091	190	0.54	2.86	0.52	259.77
1164	200	0.50	3.14	0.54	319.09
1225	210	0.49	3.39	0.51	376.31
1303	220	0.51	4.87	0.52	437.75
1414	230	0.52	4.12	0.52	539.19
1394	240	0.51	5.84	0.50	645.59
1489	250	0.50	4.71	0.51	740.58
1576	260	0.50	6.04	0.50	973.99
1706	270	0.49	5.41	0.53	1021.54
1644	280	0.50	5.92	0.51	1151.16
1710	290	0.49	6.36	0.52	1349.82
1875	300	0.50	6.75	0.50	1487.68

Table 5.10: Experiments on random small sparse graphs with $100k$ orderings with $p = \frac{5}{|V|}$

Total Edges	Total Vertices	Swap algorithm (performance ratio)	Swap algorithm (time in sec)	Ejection algorithm (performance ratio)	Ejection algorithm (time in sec)
78	4	0.73	0.03	0.78	0.49
129	6	0.68	0.04	0.73	1.91
186	8	0.69	0.07	0.71	4.65
247	10	0.64	0.10	0.71	10.00
286	12	0.58	0.10	0.63	19.57
321	14	0.60	0.20	0.64	35.29
408	16	0.55	0.25	0.62	56.35
442	18	0.56	0.22	0.59	86.50
501	20	0.53	0.38	0.62	129.75
537	22	0.55	0.45	0.60	182.82
563	24	0.51	0.36	0.55	264.36
645	26	0.53	0.61	0.58	344.24
682	28	0.49	0.69	0.55	442.29
741	30	0.54	0.81	0.58	590.42
792	32	0.48	0.61	0.55	746.34
850	34	0.51	1.02	0.54	935.22
840	36	0.50	1.13	0.55	1070.72
962	38	0.50	1.14	0.54	1293.38
978	40	0.47	1.56	0.53	1555.94

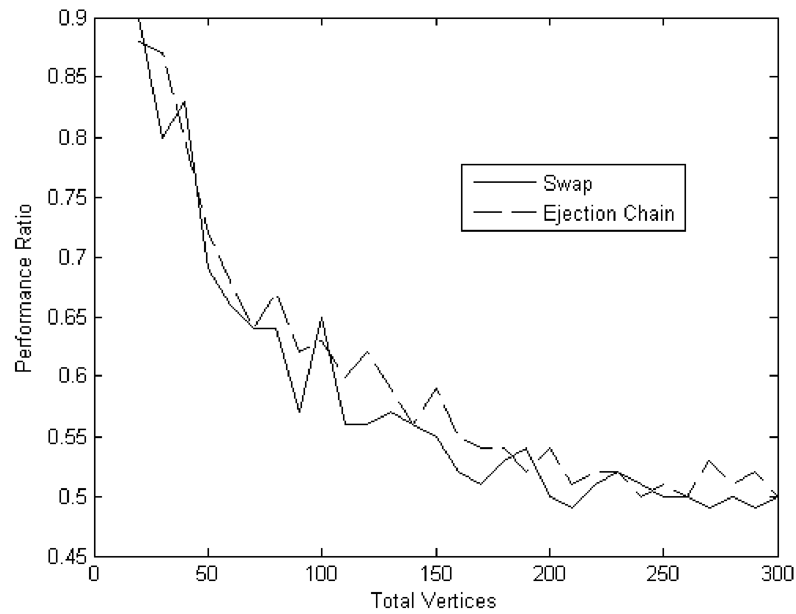

Figure 5.17: Comparison of the performance between the swap and the ejection chain algorithms on small random sparse graphs with $p = \frac{5}{|V|}$

In the next chapter we conclude with future research directions.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis we give the first set of approximation algorithms for the capacitated max k-uncut problem. We apply the algorithms to a clustering problem in bioinformatics. We experiment on graphs arising from protein protein interaction networks, however we do not draw any biological relevant conlusions from our experiments.

We consider two integer linear programs for the capacitated max k-uncut problem. We show that the integrality gap of the relaxations of these integer programs is not bounded.

We develop one local search based, one ejection chain based algorithm and one recursive greedy method to solve the problem. We analyze the local search based algorithm and recursive greedy method.

We empirically show that the local search based algorithm and recursive greedy method give us almost the same performance ratio but the ejection chain algorithm does not give us a good performance ratio for a fixed ordering. We also show that if we increase the number of orderings then the ejection method gives us a considaribly good performance but it takes more time than the other two algorithms.

6.2 Future Research Work

In the future we plan to consider the lagrangian relaxations of the linear programs described in chapter 3 in the hope of obtaining better upper bound on the optimal integral solutions.

Lagrangian relaxation has been used successfully in different combinatorial problems like traveling salesman, scheduling, set covering [15].

We can also solve integer linear programs using cutting plane algorithms, for instance by using *gomory cuts*. We applied the gomory cut technique to the LP of section 3.3. We took an odd cycle of length five and added all the gomory cuts to obtain an integral solution. We notice that 300 cuts were added to the LP. It is interesting to figure out a subset of the cuts to be added using which we can reduce the integrality gap and compute a better approximation.

The recursive greedy algorithm that is discussed in chapter 4 takes much time to solve the problem. We can also try to minimize the running time of this algorithm. It would be interesting to examine how to speed up the computation of ejection chain as well.

Bibliography

[1] A. Ageev, R. Hassin, and M. Sviridenko. A 0.5-approximation algorithm for max dicut with given sizes of parts. *SIAM Journal on Discrete Mathematics*, 14(2):246–255, 2001.

[2] A. Ageev and M. I. Sviridenko. An approximation algorithm for hypergraph max k-cut with given sizes of parts. In *Lecture Notes in Computer Science (Proceedings of ESA '00)*, volume 1879, pages 32–41, 2000.

[3] Ravindra K. Ahuja, Ozlem Ergun, James B. Orlin, and Abraham P. Punnen. *In Handbook of Approximation Algorithms and Metaheuristics edited by F. T. Gonzalez*. Chapman and Hall/CRC, 2007.

[4] G. Andersson. An approximation algorithm for max p-section. In *Lecture Notes in Computer Science (Proceedings of STACS'99)*, volume 1563, pages 237–247, 1999.

[5] G. Bader and C. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4,2, 2003.

[6] M. Bern and D. Eppstein. Approximation algorithms for geometric problems. In *Approximation Algorithms for NP-hard Problems edited by D. Hochbaum*, pages 296–345, 1996.

[7] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11):1222–1239, 2001.

[8] G Calinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *30th annual ACM symposium on theory of computing*, volume 48-52, pages 551–570, 1998.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Prentice Hall, 1998.

[10] E. Dalhous, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, pages 864–894, 1994.

[11] G. Dantzig. *Linear programming and extensions*. Princeton University Press, 1963.

[12] G. Bernard Dantzig and M. Narain Thapa. *Linear Programming*. Springer, 1997.

[13] E. de Klerk, D. V. Pasechnik, and J. P. Warners. On approximate graph colouring and max k-cut algorithms based on the θ function. In *J. Comb. Optim.*, volume 8(3), pages 267–294, 2004.

[14] U. Feige and M. Langberg. Approximation algorithms for maximization problems arising in graph partitioning. *J. Algorithms*, 41:174–211, 2001.

[15] M. L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science J*, 27,1:1–18, 1981.

[16] G. W. Flake, R. E. Tarjan, and K. Tsioutsioulikis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4):385–408, 2004.

[17] A. Frieze and M. Jerrum. Improved approximation algorithms for max *k*-cut and max bisection. *Algorithmica*, 18(1):67–81, 1997.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the theory of NP-Completeness*. Pearson Addison Wesley, 1979.

[19] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theor. Comput. Sci*, 1:237–267, 1976.

[20] D. R. Gaur, R. Krishnamurti, and R. Kohli. The capacitated max k -cut problem. *Mathematical Programming*, 115:65–72, 2008.

[21] F. Glover and C. Rego. Ejection chain and filter-and-fan methods in combinatorial optimization. *Springer*, 4:263–296(34), 2006.

[22] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. In *J. Assoc. Comput. Mach*, volume 42, pages 1115–1145, 1995.

[23] M. X. Goemans and D. P. Williamson. Approximation algorithms for max 3-cut and other problems via complex semidefinite programming. In *J. Comput. Sys. Sci.*, volume 68(2), pages 442–470, 2004.

[24] R. E. Gomory. Outline of an algorithm for integer solution to linear programs. In *Bulletin Amer. Math. Soc.*, volume 64 no. 5, pages 275–278, 1958.

[25] R. E. Gomory and T. C. Hu. Multiterminal network flows. *Journal of the SIAM*, 9:551–570, 1961.

[26] F. T. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.

[27] I. Hajirasouliha, F. Hormozdiari, S. C. Sahinalp, and I. Birol. Optimal pooling for genome re-sequencing with ultra-high-throughput short-read technologies. *Bioinformatics*, 24(13):i32–i40, 2008.

[28] Ellis Horowitz, Sartaj Sahani, and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms*. W. H. Freeman and Company, 1998.

[29] http://dip.doe mbi.ucla.edu.

[30] V. Kann, S. Khanna, J. Lagergren, and A. Panconesi. On the hardness of approximating max k-cut and its dual. *Chicago J. Theor.Comput. Sci.*, 2:1–18, 1997.

[31] R. M. Karp. On the complexity of combinatorial algorithms. *Networks*, 5:45–68, 1975.

[32] H. Kawaji, Y. Yamaguchi, H. Matsuda, and A. Hashimoto. A graph-based clustering method for a large set of sequences using a graph partitioning algorithm. In *Genome Informatics*, volume 12, pages 93–102, 2001.

[33] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. In *BSTJ*, volume 49 (2), pages 291–307, 1970.

[34] L. G. Khachiyan. A polynomial algorithm in linear programming. In *Soviet Mathematics Doklady*, volume 20, pages 191–194, 1979.

[35] A. D. King, N. Przulj, and I. Jurisica. Protein complex prediction via cost-based clustering. *Bioinformatics*, 20(17):3013–3020, 2004.

[36] M. Langberg, Y Rabani, and C. Swamy. Approximation algorithms for graph homomorphism problems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 176–187, 2006.

[37] C Lin, Y. Cho, W. Hwang, P. Pei, and A. Zhang. *Knowledge Discovery in Bioinformatics*. John Wiley and Sons, Inc., 2007.

[38] James B. Orlin, Abraham P. Punnen, and Andreas S. Schulz. Approximate local search in combinatorial optimization. *SIAM J. Comput.*, 33(5):1201–1214, 2004.

[39] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization; Algorithms and Complexity*. Prentice Hall, 1982.

[40] S. Ron and R. Sharan. A clustering algorithm for gene expression analysis. In *Eighth International Conference on Intelligent System for Molecular Biology*, 2000.

[41] S. Sahni and T. Gonzalez. P-complete approximation problems. *J. ACM*, pages 555–565, 1976.

[42] H. Saran and V. V. Vazirani. Finding k-cuts within twice the optimal. *SIAM Journal on Computing*, pages 24:101–108, 1995.

[43] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[44] Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. John Wiley and Sons, Inc., 1999.

[45] Y. Xu, V. Olman, and D. Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.