Graph Clustering
with
Restricted Neighbourhood Search

by

Andrew Douglas King

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2004 by Andrew Douglas King

Abstract

Graph Clustering

with

Restricted Neighbourhood Search

Andrew Douglas King

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

Given a graph $G = (V, E)$, a *clustering* of G is a partitioning of V that induces a set of subgraphs $C_1, C_2, \ldots, C_k$. These subgraphs are known as *clusters*. A good clustering induces clusters that are both dense and sparsely inter-connected, and represents a natural grouping of the vertices into highly interrelated sets.

By assigning cost functions to the set of clusterings of a graph, we can apply local search techniques to the clustering problem in order to find a solution of low cost. We employ a specialized local search method to create the RNSC (Restricted Neighbourhood Search Clustering) algorithm, a new randomized algorithm that generates significantly lower-cost clusterings than previous approaches.

The object of this thesis is to outline the graph clustering problem and its applications, and to present the RNSC algorithm and its results in detail along with promising avenues of future research.

Dedication

To Ellis Ostovich. You were a dear, loving friend and a true intellectual.

Acknowledgements

I would like to thank Rudi Mathon, my supervisor, for his support and guidance throughout the project. I would also like to thank Derek Corneil, my second reader, for his valuable feedback and criticism. Further, I would like to thank my colleagues in the field of discrete math, particularly Gary MacGillivray, Frank Ruskey, and Nataša Pržulj, for supporting my research and fostering my interest in the subject.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cluster analysis, in its most general form, encompasses a fundamental cognitive problem. Given a set of objects, how can we recognize groups of similar objects while differentiating between those that are dissimilar? Identifying these groups, or *clusters*, has natural applications to any operation involving the organization of the underlying set of objects. Such applications include the study of biological cell structure, module organization in software engineering, and VLSI chip design.

Graph clustering is a relatively new area within cluster analysis. In the case of graph clustering, we want to find a decomposition of the vertex set into natural subsets that are highly intra-connected and sparsely inter-connected. By applying a cost function to the set of decompositions, we can apply local search methods in order to find suitable clusters. This thesis discusses applications of and approaches to graph clustering. Its chief concern is the new RNSC (Restricted Neighbourhood Search Clustering) algorithm, a graph clustering algorithm that yields superior results as measured by our performance criteria.

1.1 Graph Clustering: Definitions and Preliminaries

Let $G = (V, E)$ be a simple graph. A *k-clustering* of G is a partitioning of V into k sets $V_1, V_2, \ldots, V_k$. These partitions induce the subgraphs $C_1, C_2, \ldots, C_k$ of G, which we call *clusters*. In a good clustering, the clusters have high density, i.e. are nearly complete graphs, and there are few edges in E whose endpoints lie in different clusters.

Less formally, a clustering of G is a partitioning of the graph into induced near-cliques which are sparsely inter-connected. If G is a weighted graph, then we demand that in a good clustering, each C_i contains a high edge weight sum and the sum of the weights of edges in G between these subgraphs is low. The question of how to evaluate the "goodness" of the clusterings of a graph is rather complicated, unlike assigning costs to proper colourings of a graph, for example. This issue of performance criteria is discussed in greater depth in Chapter 2.

Notation and terminology in the cluster literature is inconsistent, so we will adopt our own for the purposes of this thesis. A glossary of terms is given in Appendix A.

When considering a graph G, the graph's vertex and edge sets will be V and E respectively. The set of clusterings of G, loosely equivalent to the set of partitionings of V, will be denoted $\mathcal{C}[G]$. Within this set, clusterings will generally be denoted with the symbol $\mathcal{C}$. Given a clustering, the clusters within said clustering will be denoted with the symbol C. For a given cluster C, which is a graph, the vertex and edge sets will be $V(C)$ and $E(C)$ respectively. For a vertex $v \in V$, the cluster containing v will be denoted C_v. We must now give some definitions.

Definition 1.1.1. *The* density *of an unweighted graph or cluster is the proportion of possible edges that are present.*

For example, a clique has density 1, an independent set has density 0, and the graph C_4 has density $4/6$. The density of the graph $G = (V, E)$ is $|E|/\binom{|V|}{2}$.

Definition 1.1.2. *Given a clustering $\mathcal{C}$ on a graph G, a* cross-edge *is an edge whose* *endpoints lie in different clusters. A* contained edge *is an edge whose endpoints lie in the same cluster.*

In this thesis we present the RNSC algorithm, a stochastic graph clustering algorithm which uses restricted neighbourhood search. We can view a clustering of a graph G as an approximation of a partitioning of the graph into cliques, which is equivalent to an approximation of a proper colouring in $\overline{G}$, the complement of G (in the sense that colour classes will be nearly-independent sets). We chiefly consider discrete (unweighted) graphs, although the weighted case will be mentioned from time to time. All graphs in this thesis are undirected.

1.2 Restricted Neighbourhood Search and the RNSC Algorithm

The main contribution of this thesis is the Restricted Neighbourhood Search Clustering (RNSC) Algorithm, which uses a largely unexplored search scheme called restricted neighbourhood search. Like the tabu search method, which is described in [27, 28], restricted neighbourhood search is a meta-heuristic that can be applied to various solution space search schemas. It is a form of local search algorithm that relies on problem-specific idiosyncrasies. In the case of the cost-based clustering problem, such idiosyncrasies allow us to maintain our data structures very efficiently, particularly in the case of the naive cost function, which is described in Section 2.1.2.

Various considerations must be made vis-à-vis search parameters in both theory and practice. In general, we consider both intensification (focusing the search on a particular area of the search space) and diversification (random shaking on a periodic basis) in an attempt to improve performance. We must also decide what our moves will be. That is, how we will traverse the search space. In the case where cluster sizes must be preserved, we might use a transposition of two vertices between their respective clusters. In RNSC, however, a move will simply consist of one vertex being moved from one cluster to another cluster. Algorithmic considerations are discussed thoroughly in Chapter 3.

1.3 Experimental Procedures and Performance

There are several problems with the current state of cluster algorithm benchmarking. This is discussed in some depth by van Dongen in [64], as are several methods of scoring clusterings of a graph. This scoring is to a large extent a subjective matter. Without a solid benchmark set and cost function, experimental performance is less meaningful. In his thesis, van Dongen presents both naive and scaled performance criteria for clusterings of both simple and weighted graphs. We consider these and other factors when judging the empirical performance of the RNSC algorithm. However, since RNSC uses the performance criteria themselves to find good clusterings, scores will certainly, in general, be better than they will be for more natural, organic clustering algorithms like MCL. Because the goodness of a clustering is to a large extent a subjective matter, it makes sense to consider several performance measures over a large set of test graphs.

Of course, we must consider a balance when judging the performance of the algorithm; we want to reach our final solution in a small number of iterations (moves, in our case), but we also want each move to be carried out very quickly. Because Restricted Neighbourhood Search is a stochastic meta-heuristic, we also consider the possibility of performing the algorithm multiple times on the same instance, with different random seeds or different initial conditions. Our experimental methods and results are discussed in Chapter 4.

1.4 Organization

The remainder of this thesis lies in four chapters, organized as follows:

In Chapter 2, we discuss in greater depth the graph clustering problem. In particular, we outline the many concerns that need to be addressed by the RNSC algorithm and discuss previous approaches to the problem, both practical and theoretical. Also addressed in this chapter is the subject of graph problems which are closely related and analogous to graph clustering.

In Chapter 3, we outline the RNSC algorithm itself, taking particular care to discuss the method of approach, the parameters of the algorithm, and their significance. The experimental results (as well as the experimental procedures) are addressed in Chapter 4. Finally, Chapter 5 contains our conclusions drawn from the experimental results, as well as a subjective assessment of the successes and failures of the algorithm. In this chapter, suggestions for possible future development and discussion of remaining work are presented. Appendix A contains a glossary of notation.

An application of RNSC to the problem of protein complex prediction in protein-protein interaction networks is presented in [41].

Chapter 2

Graph Clustering and Related Problems

2.1 The Graph Clustering Problem

2.1.1 Overview

Graph clustering is a natural problem to consider. However, in contrast to similar graph problems, for example clique finding and colouring, determining the goodness of a given clustering for a graph is not so concrete. Such a clustering is best put in context. For example, two clusterings of the same graph may both seem very good, and even equally natural, where one clustering contains many more clusters than the other. How do we determine which is better? Even more difficult, how do we determine which is better without *a priori* knowledge of the graph's context and meaning?

For a broad, general solution to the graph clustering problem, a fine balance of high- and low-level functionality is needed. As stated in [64], it seems unlikely that there might be a good general cluster algorithm that adopts a one-way approach, and the high dimensionality of the solution space would imply that exhaustive algorithms are all but useless. Rather, our approach should be to create a general algorithm (ours, the RNSC

n	0	1	2	3	4	5	6	7	8	9	10	11
B_n	1	1	2	5	15	52	203	877	4140	21147	115975	678570

Table 2.1: The Bell numbers B_n for $n < 12$.

algorithm, is stochastic) that works well in spite of the depth of the problem. In our case, this means that we need to adopt expressive performance criteria. This is necessary to a large extent even in those cases where we know beforehand the general class of solution that we are looking for.

We want an algorithm that will work well whether or not we know anything about the graph's context, but invariably, understanding of our data set will help us apply the algorithm effectively. For example, an application to protein complex prediction within protein-protein interaction networks is discussed in [41]. This application is a clear instance in which knowledge of not only the graph's properties but the problem itself helps us to form meaningful data with the clustering algorithm.

The number of k-clusterings of a graph on n vertices is the Stirling number of the second kind $S(n,k)$, so the total number of clusterings of such a graph is the sum over all possible k of these Stirling numbers. This is the *Bell number* B_n, and is given in closed form as

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k = \sum_{k=1}^{n} S(n,k) = \sum_{k=1}^{n} \left[\frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n \right] \tag{2.1}$$

as given in [58]. Table 2.1 gives the Bell number B_n for $n \leq 12$.

The RNSC algorithm is based on changing the clustering by iteratively moving one vertex from its cluster to another (possibly empty) cluster. It is therefore pertinent to consider the distance between two clusterings in the search space. Given two clusterings of a graph $G = (V,E)$ for which a vertex v is in the same cluster in each (without loss of generality), the distance between them can be no more than $|V| - 1$. This distance is achieved trivially by moving each vertex to its new cluster.

2.1.2 Performance Criteria

Restricted Neighbourhood Search is a local search meta-heuristic that works to minimize a cost function in the solution space. Thus we must consider what kind of cost function we want to use, as well as any other performance criteria which may guide the algorithm usefully.

It is difficult to provide an expressive cost function for graph clusterings. Beyond that, we run into the problem that the more expressive cost functions are more costly to evaluate and maintain over the algorithm's run. Stijn van Dongen gives two simple vertex-wise performance criteria for clusterings on unweighted graphs as the sum of a *coverage measure* taken on each vertex. The first is naive, and for a clustering $\mathcal{C}$ on a graph $G = (V, E)$ in which $|V| = n$, it is expressed as

$$\mathrm{Cov}(G, \mathcal{C}, v) = 1 - \frac{\#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)}{n - 1} \tag{2.2}$$

where $\#^1_{\mathrm{out}}(G, \mathcal{C}, v)$ is the number of cross-edges incident to v and $\#^0_{\mathrm{in}}(G, \mathcal{C}, v)$ is the number of vertices in C_v that are not adjacent to v (recall that C_v is the cluster containing v in $\mathcal{C}$). In a good clustering, we want both $\#^1_{\mathrm{out}}(G, \mathcal{C}, v)$ and $\#^0_{\mathrm{in}}(G, \mathcal{C}, v)$ to be small for all vertices.

We can equivalently (modulo a linear mapping) express the cost of a clustering $\mathcal{C}$ on a graph G as a function $C_n(G, \mathcal{C})$:

$$C_n(G, \mathcal{C}) = \frac{1}{2} \sum_{v \in V} \big(\#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)\big). \tag{2.3}$$

Definition 2.1.1. *Consider a clustering $\mathcal{C}$ on a graph $G = (V, E)$. The graph* suggested *by $\mathcal{C}$ is the graph whose vertex set is V, in which two vertices are adjacent if and only if they are in the same cluster.*

Our formulation of the naive cost function is easy to describe. It is the size of the symmetric difference of the edge sets of G and the graph suggested by $\mathcal{C}$. That is, the number of edges in which the two graphs differ.

This cost function, however, is lacking. For sparse graphs, this cost function will be unreasonably low for clusterings without large clusters. For example, in a graph with an edge probability of < 0.5, a single cluster will always have a higher cost than n singleton clusters (see Figure 2.3 and Table 2.2). The naive cost function also fails to consider changes to small neighbourhoods as being more significant than changes to large neighbourhoods. In spite of this criterion's shortcomings, it allows the RNSC algorithm to maintain its data structures very efficiently, and is therefore of great use to us, as we will see in Chapter 3.

A more expressive cost function is derived from van Dongen's scaled coverage measure, which is defined as

$$\mathrm{Cov}(G, \mathcal{C}, v) = 1 - \frac{\#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)}{|N(v) \cup C_v|} \tag{2.4}$$

where $N(v)$ is the open neighbourhood of v. Again, we will manipulate this cost function to better suit our needs. The formula we will use for $C_s(G, \mathcal{C})$, the scaled cost function, is

$$C_s(G, \mathcal{C}) = \frac{n-1}{3} \sum_{v \in V} \frac{1}{|N(v) \cup C_v|} \left(\#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)\right). \tag{2.5}$$

Although this cost function is more expressive than the naive cost function, it leads to the RNSC algorithm's data structures being more complicated and time-consuming to maintain in practice. The $(n-1)/3$ factor in Equation 2.5 is used to preserve the bound of $n-1$ in the change of cost of a single move that is present in the naive cost function. It is easy to see that our naive cost function can change by an achievable maximum of $n-1$ in a single move. We will prove the same for the scaled cost function.

Theorem 1. *Moving a vertex from one cluster to another in a clustering $\mathcal{C}$ for a graph $G = (V, E)$ changes the scaled cost by less than $n-1$, where $n = |V|$.*

Proof. We need only prove the upper (positive change) bound, since reversing a move will reverse the change in cost. Suppose we make a move, moving vertex v from cluster C_i to

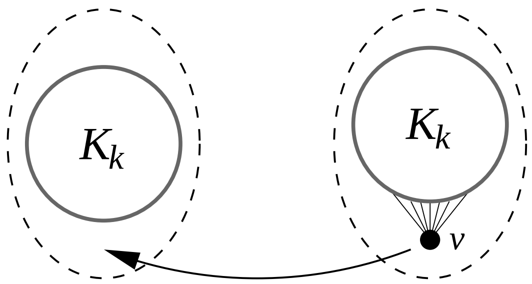

Figure 2.1: An example of a move that incurs the maximum possible change in scaled and naive cost. The naive cost increases by $n-1$ and the scaled cost increases by $k(\frac{4k}{6k+3}+\frac{4k}{3k+3})$, where $k=(n-1)/2$.

cluster C_j. For every $u \in V$, let $\alpha_u = \#^1_{\text{out}}(G,\mathcal{C},u)+\#^0_{\text{in}}(G,\mathcal{C},u)$ and let $\beta_u = |N(u) \cup C_u|$. We can partition the vertices for which the individual cost changes as follows:

$$
\begin{aligned}
&u \in S_1 = N(v) \cap C_i : \alpha_u := \alpha_u + 1 && && \frac{\alpha_u}{\beta_u} := \frac{\alpha_u}{\beta_u} + \frac{1}{\beta_u}\\
&u \in S_2 = N(v) \cap C_j : \alpha_u := \alpha_u - 1 && && \frac{\alpha_u}{\beta_u} := \frac{\alpha_u}{\beta_u} - \frac{1}{\beta_u}\\
&u \in S_3 = \overline{N(v)} \cap C_i : \alpha_u := \alpha_u - 1 && \beta_u := \beta_u - 1 && \frac{\alpha_u}{\beta_u} := \frac{\alpha_u}{\beta_u} - \frac{\beta_u - \alpha_u}{\beta_u^2 - \beta_u}\\
&u \in S_4 = \overline{N(v)} \cap C_j : \alpha_u := \alpha_u + 1 && \beta_u := \beta_u + 1 && \frac{\alpha_u}{\beta_u} := \frac{\alpha_u}{\beta_u} + \frac{\beta_u - \alpha_u}{\beta_u^2 + \beta_u}
\end{aligned}
\tag{2.6}
$$

The value of α_v, of course, is bounded above by $\beta_v - 1$.

Suppose the change in cost is maximum. Then of course S_2 and S_3 are empty. For $u \in S_1$, $\beta_u \geq |C_i|$ (note that for this analysis, $v \in C_i$). For $u \in S_4$, $\beta_u \geq |C_j|$. So the change in cost due to vertices in C_i (excluding v) is bounded by $\frac{(n-1)(|C_i|-1)}{3|C_i|}$. Similarly the change in cost due to vertices in S_4 is bounded by $\frac{(n-1)|C_j|}{3(|C_j|+1)}$. The change in cost due to v is bounded by $\frac{(n-1)}{3}$; our total change in cost is bounded by the sum of these three bounds, which is less than $n-1$. □

Theorem 2. *The bound given in Theorem 1 is achievable asymptotically.*

Proof. It is easy to show that this bound of $n-1$ can be approached asymptotically. Consider a graph consisting of a copy of K_k and K_{k+1}, which are in turn the clusters.

The cost of this clustering is 0. Now consider moving a vertex v from the $k+1$-clique into the same cluster as the k-clique. The new cost due to v will be $\frac{(2k)(2k)}{3(2k+1)}$. Each other vertex will carry a cost of $\frac{(2k)(1)}{3(k+1)}$ for a total contribution of $\frac{(2k)(2k)}{3(k+1)}$. This means that the new cost will be $\frac{4k^2}{6k+3} + \frac{4k^2}{3k+3}$, which approaches $2k$ as k approaches infinity. Recall that $2k = n - 1$. □

See Figure 2.1 for an example of this change in cost. In the next chapter, we will discuss how these two cost functions are implemented.

In [45], Mancoridis *et al.* present the notion of *modularization quality*, which is a cost function that, unlike our naive and scaled cost functions, is evaluated cluster-wise rather than vertex-wise. This is based on *intra-connectivity* and *inter-connectivity*. The intra-connectivity of a cluster is the cluster's density (which, for a cluster of size n with m arcs, is m/n^2 since they are considering directed graphs). The inter-connectivity between two different clusters is similarly defined as the proportion of possible edges between the two clusters that actually exist, i.e. $m/(n_1 n_2)$ for two clusters of sizes n_1 and n_2 with m arcs between them.

Denoting the intra-connectivity of cluster C_i by A_i and the inter-connectivity of clusters C_i and C_j by $E_{i,j}$, Mancoridis *et al.* give the modularization quality of a clustering containing k clusterings as

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\binom{k}{2}}\sum_{i,j=1}^{k} E_{i,j} & \text{if} \quad k > 1 \\ A_1 & \text{if} \quad k = 1 \end{cases} \tag{2.7}$$

That is, the average inter-connectivity over each pair of clusters, subtracted from the average intra-connectivity of the clusters. This cost function falls between -1 and 1, and can easily be applied to simple undirected graphs by replacing the notion of density in a directed graph with density in an undirected graph. This is applied to an example later in this section.

For reasonably large, sparse graphs, the modularization quality function heavily favours very dense clusters, far more than the naive cost function. Unless the graph

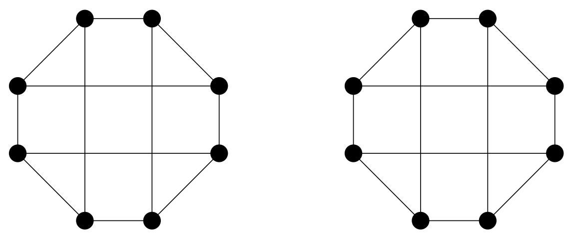

Figure 2.2: The graph $G = Q_3 + Q_3$.

is very dense, the average inter-connectivity will be very low in *any* reasonable clustering. This means that the modularization quality, in effect, rests solely upon the density of the clusters. The measure can still be useful, but becomes less a matter of clustering and more a matter of partitioning the graph into dense subgraphs.

An Example

Consider the graph G shown in Figure 2.2. It has two components, each of which is the hypercube Q_3. The immediate clustering that springs to mind simply consists of two clusters, the components. However, we can look at several other clusterings, each of which has some merit. We can partition the graph into copies of Q_3, copies of Q_2, copies of Q_1, or copies of Q_0 (singletons). Call these clusterings $\mathcal{C}_3, \mathcal{C}_2, \mathcal{C}_1$, and $\mathcal{C}_0$ respectively. They are all shown in Figure 2.3.

The naive cost function, the scaled cost function, and the modularization quality (modified for simple undirected graphs) are given for each of these clusterings in Table 2.2. Note that we consider the singleton clusters to have density 1 in the evaluation of modularization quality.

Even in this small example, the modularization quality's bias towards small, dense clusters is evident. While this may be useful for some purposes, it should not be considered for something like sparse biological data (see Section 4.1.2), where the density of graphs is in the neighbourhood of 10^{-3}. Also note that because Q_3 has a density of less than .5, the naive cost function sees $\mathcal{C}_0$ as being better than $\mathcal{C}_3$. Highly structured

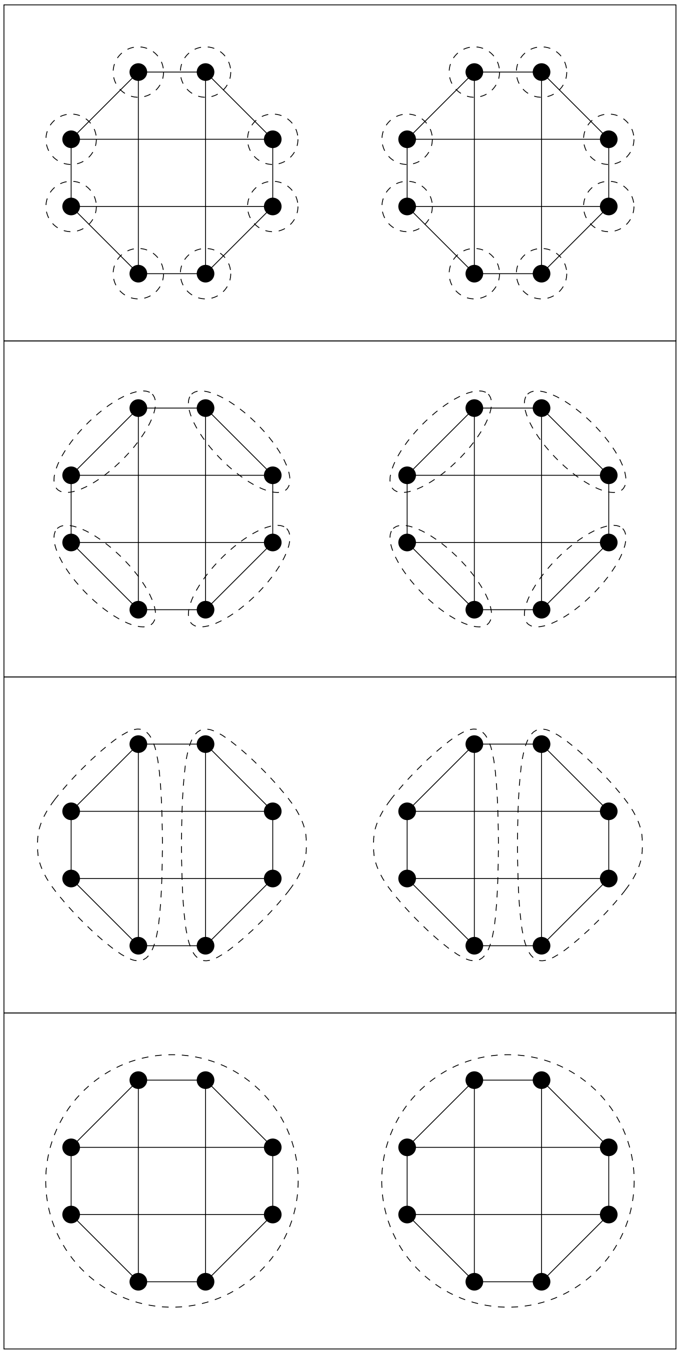

Figure 2.3: Clusterings of $G = Q_3 + Q_3$ (from top): $\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2$, and $\mathcal{C}_3$.

Clustering	Naive Cost	Scaled Cost	Modularization Quality
$\mathcal{C}_0$	24	40	$1 - \frac{1}{5} = .80000$
$\mathcal{C}_1$	16	40	$1 - \frac{1}{7} = .85714$
$\mathcal{C}_2$	16	32	$\frac{4}{6} - \frac{1}{12} = .58333$
$\mathcal{C}_3$	32	60	$\frac{12}{28} - 0 = .42857$

Table 2.2: Quantitative performance criteria for clusterings of $G = Q_3 + Q_3$.

graphs like this serve as good examples of the difficulties met when trying to evaluate the performance of a clustering algorithm. It is hard to say which clustering is the best one in this case, although $\mathcal{C}_3$ seems to be the most natural. This problem is discussed at length in van Dongen's Ph.D. thesis [64]. It should be noted that van Dongen's MCL algorithm, which deterministically creates a clustering based on network flow rather than a numerical cost function, gives $\mathcal{C}_3$ as its clustering of G. The MCL algorithm will be discussed later in the chapter, and serves as a contrast to RNSC in several examples.

2.2 Related Problems

We are, of course, considering a very narrow definition of the graph clustering problem, and there are several problems that are akin to the clustering problem. The problem of graph partitioning, a generalization of the min-cut problem, is discussed at length in [16]. Some forms of graph drawing are intimitely related to our model of graph clustering, but that will be discussed in Section 2.4. Graph colouring is related to clustering the complement of a graph in a certain way, and of course, there are several different models of graph clustering itself.

The graph partitioning problem, as presented by Elsner in [16], is as follows. Consider a graph $G = (V, E)$ with an edge weight function $W_e : E \to \mathbb{N}$ and a vertex weight function $W_v : V \to \mathbb{N}$. A valid k-partitioning $\mathcal{P} = (V_1, V_2, \ldots, V_k)$ of V is one with

$\sum_{v \in V_i} W_v(v)$ equal for all $i \in \{1, 2, \ldots, k\}$. Let $E_\times(\mathcal{P})$ be the set of cross-edges under a partitioning $\mathcal{P}$, that is, edges whose endpoints are in different partitions. Find a valid k-partitioning of V such that

$$\sum_{e \in E_\times(\mathcal{P})} W_e(e) \tag{2.8}$$

is minimized over all such partitionings. This problem is well-known to be NP-complete. There are many variations on it. For example in [20], the partitions in $\mathcal{P}$ have prescribed sizes and the vertices are unweighted.

We are concerned with two things in our model of clustering: low inter-connectivity and high intra-connectivity. That is, we want few edges between clusters and many edges within them. This concept generalizes naturally to edge-weighted graphs, and so the graph partitioning problem essentially becomes the same as the clustering problem, but giving no consideration to the intra-connectivity of the clusters. Of course, this begs the question of what kind of problems we can emulate by disregarding the clusters' inter-connectivity, focusing only on finding dense subgraphs.

In this case, a perfect matching would be a decomposition into copies of K_2, and in the simplest sense it would be an optimal clustering (although in this case it would make sense to consider the size of the clusters). In general, this problem would be a relaxed variant of the NP-complete problem Partition Into Cliques, which simply asks whether a graph can be partitioned into at most k partitions such that each partition induces a complete subgraph. This is clearly equivalent to Chromatic Number in the graph's complement [25]. Indeed, the RNSC algorithm has been modified and used to colour graphs. This application is discussed in Section 3.6.

Much work has been done in the area of identifying dense subgraphs, from the perspective of clustering and otherwise. Hartuv and Shamir, among others, proposed a clustering model based on high cluster connectivity [33], which will be discussed later in this chapter.

A closely related concept to that of highly connected subgraphs is that of a graph's

k-cores. A k-core of a graph is a maximum connected induced subgraph with minimum degree at least k. Not only are k-cores very easy to identify (in time $\mathcal{O}(n)$), but minimum degree is usually a good approximation of edge-connectivity. Clearly the 1-cores of a graph are the nontrivial connected components of the graph. If we want the k-cores of a graph, we can iteratively remove vertices from the graph until the minimum degree is $\geq k$. Seidman studied the properties of k-cores for the purposes of evaluating cohesion in social networks [59]. In that paper, he gave the following tight bound of the vertex-connectivity of a graph's k-core:

Theorem 3. (Seidman [1983]) *If $l \leq k$ and $p \leq 2k - l + 2$, then a k-core with p-points has connectedness at least l.*

The components of the remaining graph will be the k-cores of the original graph. Bader and Hogue employed the concept of k-cores in the prediction of protein complexes [5], the same problem to which King *et al.* applied the RNSC algorithm [41] and to which Pržulj *et al.* applied Hartuv and Shamir's HCS algorithm [56, 33].

It is also important to discuss the variants of the clustering problem itself. Our model considers simple, unweighted graphs and considers clusterings to be partitionings of the vertex set. Such clusterings can, of course, be considered for edge-weighted and even vertex-weighted graphs, and considerations can be made for loops. The more significant departure from our model, however, is in the notion of a clustering being a set of induced subgraphs that do not necessarily span the vertex set, and are not even necessarily disjoint. Such a model is used in [5, 33, 37] and elsewhere.

Clustering can also be viewed from a geometric standpoint. The k-median problem is as follows. Consider a set of points $X = \{x_1, x_2, \ldots, x_n\}$ in a metric space $\mathbb{M}$ under the metric m. Find k medians, i.e. k points $y_1, y_2, \ldots, y_k$ in $\mathbb{M}$, such that

$$\sum_{i=1}^{n} \min_{1 \leq j \leq k} m(x_i, y_j) \tag{2.9}$$

is minimized. This problem has countless variations, one of the most common being the facility location problem. Clustering a geometric graph is very closely related to finding a good k-median solution. The k-median problem and its various forms are discussed at length in [51]. Discrete location problems in general are discussed in [52]. In [13], de la Vega *et al.* offer polynomial time approximation schemes for three geometric clustering problems.

In [31], Good provides a comprehensive analysis of the various features of cluster analytical techniques. Although the subject of cluster analysis covers far more than clustering methods alone, the paper provides many criteria which are helpful in differentiating between clustering algorithms. For example, the concept of k-cores as clusters leads to a clustering method which is inherently hierarchical and deterministic, whereas the MCL algorithm is deterministic, but nonhierarchical. The entire area of clustering algorithms allows a multitude of variations on a single idea, some of which are more restricted to the model of graphs than others.

2.3 Previous Approaches

Beyond displaying a multitude of variations on the graph clustering problem, the previous approaches within the literature provide a wide range of methodologies and applications. Many approaches, like the RNSC algorithm, are stochastic algorithms, whereas others are deterministic.

2.3.1 Deterministic Techniques

Deterministic clustering techniques, while having the advantage of consistency, have unique output, a disadvantage in some cases.

Exhaustive Search

Given a cost function for clusterings of a graph, a trivial way of finding an optimal clustering is through an exhaustive search of all possible clusterings. This method is employed in [45] for graphs of size ≤ 15. The Bell numbers (see Table 2.1), which represent the number of clusterings of a graph, increase superexponentially, so exhaustive approaches to clustering are completely impractical in general. As a consequence, Mancoridis *et al.* turn to a genetic algorithm and a local search algorithm for clustering, both of which are discussed later in this section.

Flow Simulation

Stijn van Dongen's work in the Markov Cluster (MCL) algorithm provides a very fast clustering method that provides a natural clustering in weighted graphs [18, 63, 64]. However, its results are far from optimal in terms of both the scaled cost and the naive cost.

The general MCL algorithm is as follows. We are given G, a weighted graph with adjacency matrix M and no isolated vertices, along with $\{e_i\}_{i=1}^{\infty}$ and $\{r_i\}_{i=1}^{\infty}$, two sequences such that $e_i \in \mathbb{N}$ and is > 1, and $r_i \in \mathbb{R}^+$. Let M' be the adjacency matrix of the associated *Markov graph*. That is, M' is a column-normalized M:

$$M'_{pq} = \frac{M_{pq}}{\sum_i M_{iq}}. \tag{2.10}$$

Let $\Gamma : (\mathbb{R}^+, \mathbb{R}^{n\times n}) \rightarrow \mathbb{R}^{n\times n}$ be the operation such that

$$(\Gamma_r(M))_{pq} = \frac{(M_{pq})^r}{\sum_i (M_{iq})^r}. \tag{2.11}$$

Let $T_1 = M'$. For $k = 1, 2, \ldots$, we let $T_{2k} = (T_{2k-1})^{e_i}$, then let $T_{2k+1} = \Gamma_{r_k}(T_{2k})$. We halt when T_{2k+1} is a (near-) idempotent matrix. The output clustering is then dictated by deterministically analyzing T_{2k+1}. In undirected graphs, certain symmetries can lead the MCL algorithm to output an overlapping fractional clustering, which is then dealt with in some prescribed way.

The MCL algorithm is based on the paradigm of flow simulation. A graph is a network model describing flow between vertices; MCL uses flow expansion and inflation to produce a natural grouping of highly flow-connected vertices, which are of course clusters. Flow *expansion* is represented by the traditional matrix exponentiation; T_{2k} is the result of the expansion step. Flow *inflation* is represented by the index-wise exponentiation, also known as the *Hadamard-Schur* product; T_{2k+1} is the result of inflation. Expansion is used to propagate flow within the graph; inflation is used to magnify its relative strength, i.e. to strengthen flow where it is strong, and to weaken it where it is already weak.

The MCL algorithm can be modified in a number of different ways, such as adding weighted loops to each vertex, providing different exponent sequences $\{e_i\}_{i=1}^{\infty}$ and $\{r_i\}_{i=1}^{\infty}$, and so on. These modifications have such effects as changing the granularity of the clustering and reducing oscillation about a convergence point. The ability to modify the granularity of clusterings, in particular, is very useful. See, for example, Figure 2.3 in this thesis and Figure 18 in [64] for examples of graphs that admit clusterings of varying granularity.

The expansion step of MCL has complexity $\mathcal{O}(n^3)$, assuming some small bound on the expansion exponents e_i. The inflation has complexity $\mathcal{O}(n^2)$. However, the matrices T_i are generally very sparse, or at least the vast majority of the entries are *near* zero. *Pruning* in MCL involves setting near-zero matrix entries to zero, and can allow sparse matrix operations to improve the speed of the algorithm vastly. The algorithm performs relatively well, generally, by the scaled and naive cost functions previously discussed (though not as well as RNSC). That considered, MCL is very fast, particularly for sparse graphs, which are the norm for many clustering applications.

Some results of MCL are considered in Chapter 4. See Figure 2.4 for an example of its output, taken from [64] and compared to RNSC output.

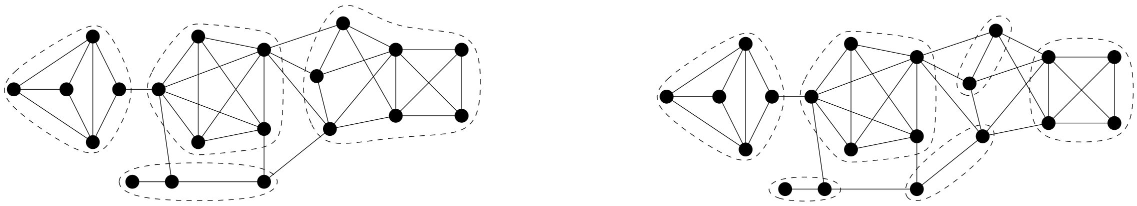

Figure 2.4: An example of MCL output (left) vs. RNSC output (right). The naive (scaled) cost of the MCL clustering is 18 (34.62) as opposed to 15 (33.92) for the RNSC clustering.

Multiple Cut

One well-known method for graph clustering, particularly common in the area of VLSI design, is the *ratio cut* method [57, 67]. The idea behind ratio cutting is to make a small edge-cut in a graph while ensuring that the cut components have nearly the same size. It is an adaptation of the Kernighan-Lin technique, in which the cut components must have prescribed, generally equal sizes [40].

For a weighted graph with c_{ij} being the capacity (or edge weight) between vertices i and j, the Kernighan-Lin approach is a heuristic for minimizing

$$\sum_{i \in V_1} \sum_{j \in V_2} c_{ij} \tag{2.12}$$

for partitions of V into V_1 and V_2 such that $|V_1| = |V_2|$, assuming $|V|$ is even. Ratio cut, however, allows flexibility in the partition sizes, heuristically minimizing

$$\frac{\sum_{i \in V_1} \sum_{j \in V_2} c_{ij}}{|V_1| \cdot |V_2|} \tag{2.13}$$

This is known as the *ratio* value of a bipartition. Note that finding the minimum weight cut and finding the minimum ratio cut are both NP-complete problems [67].

The restriction on partition sizes in the Kernighan-Lin approach can lead to poor performance, as seen in Figure 2.5 (from [67]). In this example, the min-cut approach and the ratio cut approach produce cuts of weight 13 and 2 respectively. The ratio values of these partitions are 13/25 and 2/24 respectively.

The expected ratio value of a cut in a uniform random graph is independent of the size of the two partitions; it is always the expected value of the edge weight. However, relaxing the Kernighan-Lin min-cut method so that the partitions can have different sizes will result in a lower expected cut weight for partitions of vastly different sizes [67].

In [57], Roxborough and Sen introduce the notion of multi-way ratio cut. This is basically an extension of the notion of ratio cut to a k-partitioning of the vertex set. Multi-way ratio cut attempts to minimize

$$\frac{\sum_{p=1}^{k} \sum_{i \in V_p} \sum_{i \notin V_p} c_{ij}}{\prod_{i=1}^{k} |V_k|}. \tag{2.14}$$

So instead of using iterative ratio cut to cluster a graph, a single application of multi-way ratio cut can suffice. This is a good example of a clustering scheme which seeks only to minimize inter-connectivity – the ratio ignores cluster intra-connectivity.

These cut algorithms are very simple and intuitive, and they are useful for a number of applications in which intra-connectivity is unimportant. For the graph in Figure 2.4, repeated 2-way ratio cut will result in the same clustering as MCL, as will a 4-way ratio cut. However, the RNSC output is not an optimal 6-way ratio cut.

2.3.2 Local Search Techniques

Randomized (*stochastic*) clustering methods may generate a number of good clusterings of a single graph with the same parameters. Thus, they have the obvious advantage that their output is in some ways richer, though perhaps less reliable, than the output of

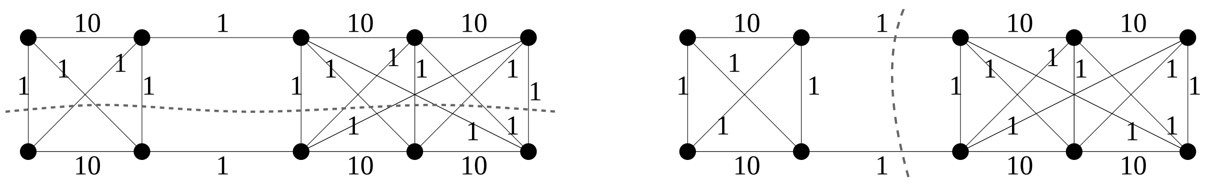

Figure 2.5: Partitions produced by the Kernighan-Lin approach and the ratio cut approach respectively.

deterministic techniques. Their individual outputs can be compared to one another and a final output, one that a deterministic method may never have reached, can be chosen.

Stochastic local search is a powerful tool for many combinatorial optimization problems, more effective for some than for others. The idea is to create a search space of all possible solutions and define a neighbourhood for each solution. In our case, the solutions are clusterings and two clusterings are neighbours of one can be reached from the other by moving a single vertex from one cluster to another, possibly creating or destroying a cluster of size 1 in the process. In this way, local search schemes attempt to iteratively improve the initial solution to reach a near-optimal solution. The most basic of these is the general hill climbing heuristic: while the current solution has a better neighbour, move to a random better neighbour. For clustering, the concept of "better" is quantified by a cost function, and we are searching for a local minimum. Mancoridis *et al.* use a hill climbing algorithm for clustering small graphs in [45]. More complex and better-performing heuristics include tabu search, genetic algorithms, and simulated annealing.

In [36], Hoos and Stützle compare systematic search algorithms with stochastic local search algorithms for 3-SAT, the propositional satisfiability problem with three literals in each clause. 3-SAT is a prominent problem in theoretical computer science, finding that for some classes of problems local search outperforms systematic methods. However, for many structured classes of the problem, including 3-SAT as a model of graph colouring, the systematic methods are better. In his PhD thesis, Hoos examines methods and applications for stochastic local search, particularly examining applications for SAT, constraint satisfaction problems, and the Hamilton cycle problem [35].

Local search techniques cannot enumerate the clusterings of a graph (which, in any case, are prohibitively large in number) and they are therefore useless for proving nonexistence of clusterings with a given property.

Tabu Search

Of all local search schemes, tabu search is the most closely related to RNSC. It was first proposed by Glover in [29] and is described in detail in [27, 28]. It is a meta-heuristic, one that guides local search heuristics. The idea behind it is to allow cost-based local search algorithms to enter, then leave local minima by preventing the search from retracing its steps and settling in a local minimum.

Tabu search achieves its goals through the maintenance of a *tabu list*, a list of states in the search space that are tabu, or forbidden. The method has been applied to colouring, the Traveling Salesman Problem (TSP), and scheduling problems, among other famous NP-hard computing problems popular in the field of operations research [27, 28, 34]. It works well as a widely-applicable extension of simple hill climbing techniques.

In the application of tabu search to cost-based clustering, tabu states would typically be clusterings of the graph. However, clusterings, particularly for sparse graphs, are highly localized. Because of this, great lengths would have to be taken in order to ensure that such a clustering algorithm would, in general, have no trouble leaving a local minimum. This matter is discussed in Chapter 3, as RNSC uses a tabu list, though it is not a list of clusterings. Further, if tabu states are clusterings, then there is some computational difficulty in checking that a state is not tabu. Of course, it would be unreasonable to store data on every clustering, as evidenced by the size of the Bell numbers.

Genetic Algorithms

Genetic algorithms are search algorithms that attempt to emulate the principles of natural selection and evolution. This concept is easy to apply to clustering, allowing clusters to be organisms and considering a lower cost to be representative of a higher level of evolution. When used properly, genetic algorithms can offer a balance between efficiency and robustness, and have been applied to many combinatorial optimization problems

[30]. Obviously, behind the principle of genetic algorithms there are many possible implementations for a given problem. Mancoridis *et al.* offer the following genetic clustering algorithm [45]:

1. Randomly generate a population of N clusterings of the graph.
2. Until there has been no improvement for t generations or every clustering in the population is a local optimum, repeat:
   - Randomly select a proportion p of the population and for each selected clustering, move to a better neighbouring partition if one exists.
   - Generate a new population of size N by selecting, with repetition, elements of the current population of pN clusterings. Make these selections randomly, with bias towards better clusterings.
3. The clustering with the best score in the final population will be the ouput solution.

The application of this algorithm was very slow (e.g. over an hour for a graph with 153 vertices and 103 directed edges), but the experiment had a chiefly qualitative, comparative bent. Also (and for this reason), their implementation was not likely optimized to exploit properties of the graph clustering problem, unlike RNSC. In fact, it may be interesting to employ the methodology and performance criteria of RNSC and apply them to create an efficient genetic clustering algorithm in the future.

Simulated Annealing

Annealing is a process common in the manufacture of glass and steel. It involves heating the material to a prescribed temperature, then lowering the temperature very slowly, so in the case of glass, stresses withing the structure are relieved rather than maintained upon return to room temperature. In the case of steel, the appearance of unwanted crystal structure is minimized. In [42], Kirkpatrick *et al.* draw on the connection between statistical mechanics and combinatorial optimization of large systems to provide an optimization heuristic called *simulated annealing*.

In simulated annealing for a combinatorial optimization problem, the "temperature" of the system is based on the applicable cost function for the problem. This temperature represents the probability that a given random move will be accepted if it lowers the cost of the solution. With a given temperature T, this probability is

$$\exp\left(\frac{C_0 - C_1}{k_\beta T}\right) \tag{2.15}$$

where C_0 is the cost before the move, C_1 is the cost after the move, and k_β is Boltzmann's constant, equal to 1.38×10^{-23} joules per kelvin. Hence a constant temperature of 0 represents a standard hill climbing algorithm. So the temperature is gradually lowered, generally after a prescribed number of attempted moves, until the cost fails to decrease over a given period of moves.

Simulated annealing is not without its shortcomings. The algorithm requires very many moves to be made, and progress is intentionally made very slowly. Further, the algorithm is, in its general form, memoryless. That is, it does not learn from its previous work.

Simulated annealing has been applied to circuit design [42], which is related to graph partitioning and hence graph clustering. Bandyopadhyay *et al.* applied simulated annealing to partitional geometric clustering, which in this case is equivalent to the k-median problem in Euclidean space [6]. Their SAKM algorithm was found to be superior to the then-standard K-means algorithm. Davidson and Harel applied simulated annealing to the problem of graph drawing using a composite energy (cost) function based on desirable qualities of a graph embedding with good results [12]. Fruchterman and Reingold developed a similar algorithm by using simulated annealing and modeling a graph with a physical spring-and-ring system [23]. Their algorithm is the drawing method used by BioLayout, a biological network layout program that is in essence a graph visualization tool [17].

Nascimento and Eades applied simulated annealing to graph clustering, but the focus of their work was the integration of user participation in graph clustering algorithms

[15]. A pure application of simulated annealing to graph clustering would certainly be interesting to see.

2.4 Applications

The practical applications for graph clustering are countless, simply because of the versatility of graphs as combinatorial models. Clustering can be appplied to any modeled system for a number of purposes. Many classes and styles of application appear in the literature.

A major area of application for graph clustering is the growing field of bioinformatics. Van Dongen, with Enright and Ouzounis, applied his MCL algorithm to form TRIBE-MCL, an approach for detecting protein families within large networks of biological data [18]. Pržulj *et al.* applied Hartuv and Shamir's HCS (highly-connected subgraph) algorithm to protein-protein interaction networks in the context of predicting protein complexes, which are known groups of highly interactive proteins [33, 56]. King *et al.* applied RNSC, along with data filtering, to the same problem with good results in [41].

Bader and Hogue applied the concept of graph cores to the protein complex prediction problem [5]. Similar methods were used in [2] and [37]. Because of the growing number of methods for large-scale data set generation, fast and accurate automated tools for analyzing this data are becoming increasingly important. There are many applications of graph theory to biological network analysis, some of which are described in [7] and [54].

The problem of physical VLSI circuit design has been studied extensively for decades. Because of the growing density and complexity of such circuits, good algorithms have been necessary to help design them sensibly. A basic clustering model for VLSI circuit design follows the idea that the circuit must be partitioned into separate components which are sparsely inter-connected. These are the clusters in the graph model, as described in [42] in the context of simulated annealing, and [57, 67] in the context of graph partitioning. Such partitioning methods have been employed with great success. Elsner [16] and Falkner *et al.* [20] have provided surveys of computational partitioning.

Graph clustering is also directly applicable to database organization. If we want similar data to be clustered, we can simply model data components as vectors and their similarities as edges. The same applies for software modularization. Given a set of source code components and their interdependencies, Mancoridis *et al.* explored clustering as a method of software organization using exhaustive search, hill climbing, and genetic algorithms [45]. Further, they explored the generation of meta-graphs from clustered graphs, in which clusters become vertices. In this way they were able to generate hierarchical organizations of source code.

In this same vein, Aksoy and Haralick used clustering to improve on image grouping and retrieval within a database [1]. Similarities were determined by an image's properties within a feature space, and an image similarity graph was generated and clustered. For retrieval, in which a query image is input and similar images are output, they added the restriction that the output images must be similar not only to the input image, but also to each other.

Clustering has also been applied to image segmentation, a problem which is detailed in Chapter III of [21], surveyed specifically in [32]. Given an image, the problem of segmenting it amounts to discerning the physical objects within it. That is, partitioning the image into regions that are likely to represent individual objects. Wu and Leahy applied graph clustering to this problem by modeling pixels as vertices and similarity between the pixels [69]. In this case, closed contours of edgels (which represent feature borders within the image) were modeled to represent minimum cuts in the graph. A flow-based partitioning algorithm was then used to identify separate entities within the image. As is common in computer vision applications, a hierarchical method was employed as a sort of divide-and-conquer method.

Graph drawing, the problem of embedding graphs nicely in Euclidean space, is also linked to graph clustering in many instances. By clustering a graph, we can identify groups of vertices that should generally be embedded close to one another in order if we want to embed the graph neatly in two-dimensional Euclidean space. Again, the notion of meta-graphs works here, where we are also interested in how the clusters should be laid out with respect to one another. Davidson and Harel's application of simulated annealing to graph drawing employs an energy (cost) function that is based on the sum of a multitude of factors, some of which, such as minimizing edge length, are highly representative of graph clustering ideals [12]. Roxborough and Sen, among others, applied clustering to graph drawing [57].

Of course, the problem of graph colouring as modeled by an intra-connectivity based graph clustering ideal is another direct application of graph clustering, and is one of the most basic application of clustering to pure graph theory itself. Hence it is evident that clustering can be applied to such problems as practical scheduling and possibly even identification of homomorphisms and design colouring, themselves being generalizations of graph colouring. [34] and [48] offer some insight to the application of local search to such problems.

Indeed, there is an enormous demand for graph clustering algorithms of all kinds, and there is a wealth of applications for them. Their importance in bioinformatics, computational vision, operations research, and data management is undeniable.

Chapter 3

The RNSC Algorithm

3.1 Overview

RNSC is a local search clustering algorithm. A single experiment of the method as applied to a graph G consists of several basic stages:

1. Either read or randomly generate an initial clustering $\mathcal{C}_0 \in \mathcal{C}[G]$.

2. Apply the naive cost function to the clustering and data structures. Attempt to minimize the naive cost by modifying the clustering one *move* at a time, reaching a best naive clustering $\mathcal{C}_n$.

3. Do the same for the scaled cost: Starting with the naive clustering $\mathcal{C}_n$, apply the scaled cost function to the clustering and data structures and attempt to minimize the scaled cost by making one move at a time. The best scaled clustering, the output of the experiment, is denoted $\mathcal{C}_s$.

This is a very high-level description of a single experiment. Generally, more than one experiment will be run, generating a set $\{\mathcal{C}_s^{(i)}\}_{i=1}^{\mathcal{N}_E}$, where $\mathcal{N}_E$ is the number of experiments. The output clustering will then be $\mathcal{C}_F$, the element of this set with the lowest scaled cost.

Step 2 in this list is the *naive cost scheme*, in which only the integer naive cost function given in Equation 2.3 is considered. Likewise, step 3 is the *scaled cost scheme*, which uses the scaled cost function for clusterings given in Equation 2.5. Figures 3.1, 3.2, and

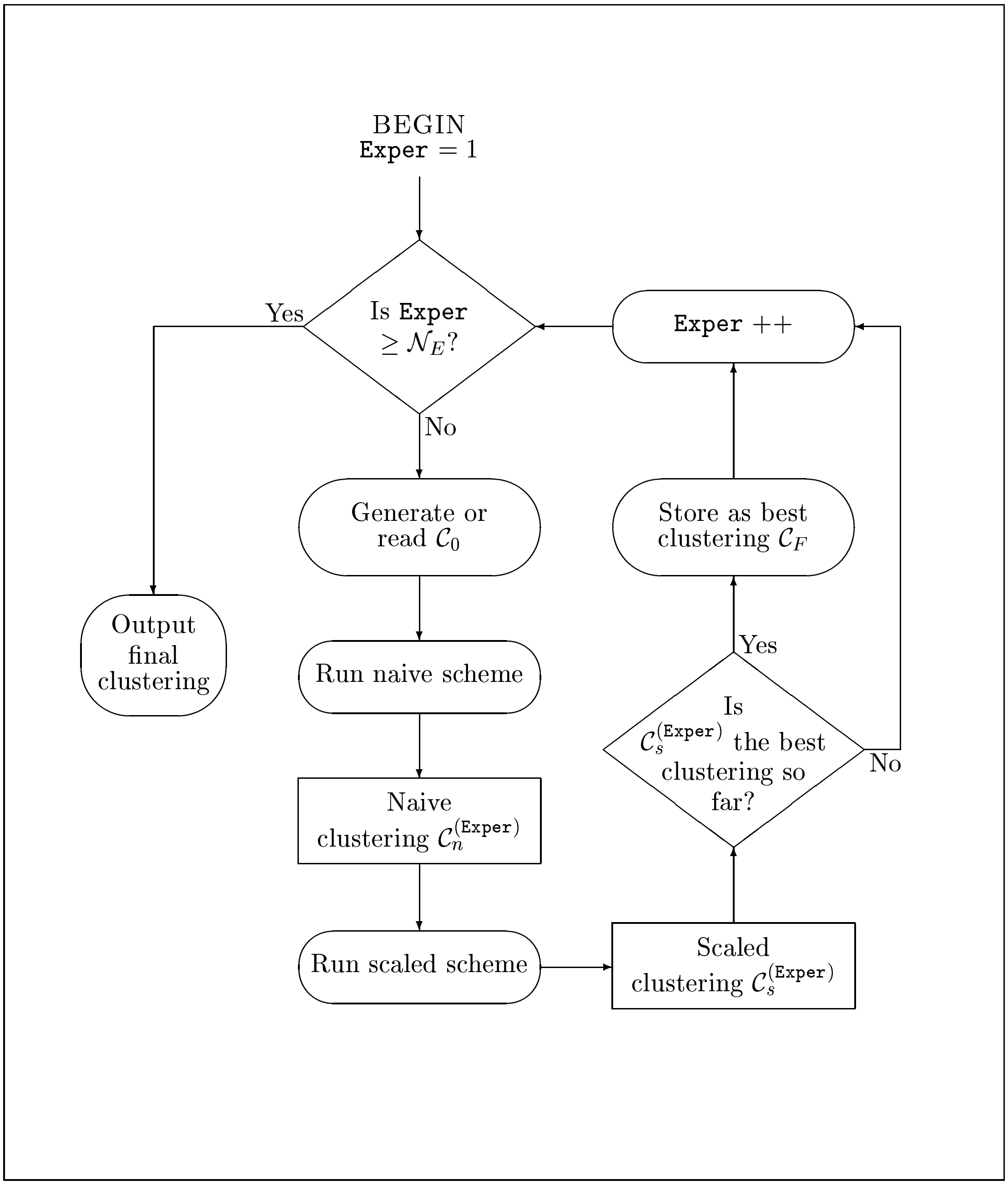

Figure 3.1: The RNSC algorithm.

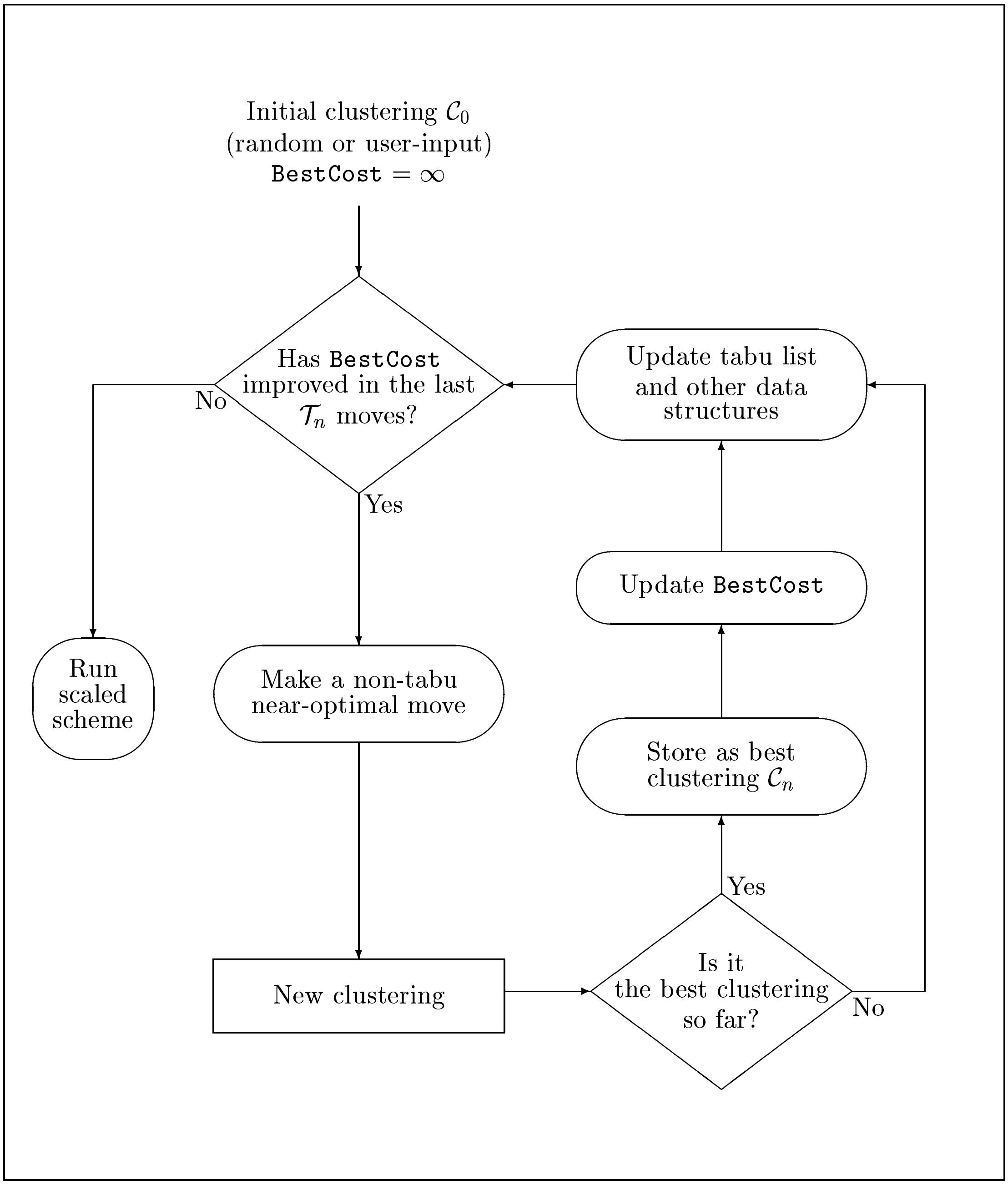

Figure 3.2: The RNSC naive cost scheme.

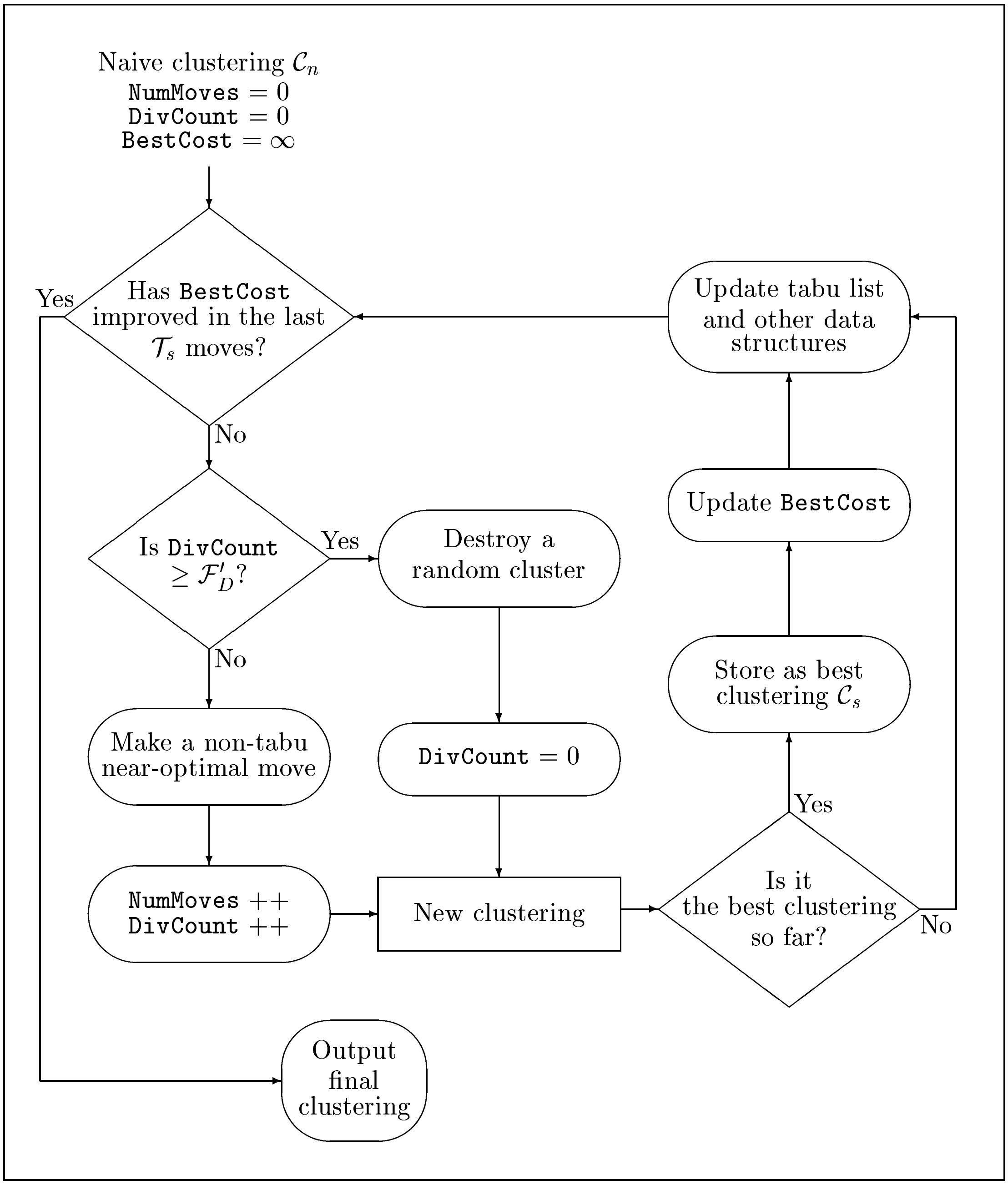

Figure 3.3: The RNSC scaled cost scheme.

3.3 show the general structure of the entire RNSC algorithm, the naive cost scheme, and the scaled cost scheme, respectively.

Of course, we must discuss several issues regarding the details of the algorithm. There are several options that must be specified to tailor the RNSC method's strategical approach to minimizing the scaled cost function in particular. Since we are generally only using the naive cost scheme as a fast preprocessor for the scaled scheme, we need not reach an exceptionally good minimum in the naive cost function; minima in the two cost functions often correspond to similar but notably distinct clusterings (see Figure 3.4).

Since RNSC is highly specialized to deal with the specific problem of clustering, it is important to detail the data management schemes and how they relate to the naive and scaled cost functions. It is also necessary to discuss the actual move types that we will make between neighbouring clusters and how they affect the clustering algorithm. This chapter is concerned with the theoretical and structural issues of the algorithm, while Chapter 4 presents experimental results regarding the relative performance of the algorithm and the tailoring of the input parameters to optimize this performance.

3.2 Move Types

Every move type that RNSC uses involves moving a single vertex from one cluster to another, possibly emptying a cluster or creating a singleton cluster in the process. As detailed in Section 3.3, RNSC uses a constant number of clusters, and the clusters may be empty. Therefore in a graph $G = (V, E)$, the neighbourhood of a clustering $\mathcal{C}$, denoted $N(\mathcal{C})$, consists of $|V| \cdot (\mathcal{N}_C - 1)$ moves, where $\mathcal{N}_C$ is the number of clusters. This is because there are $|V|$ vertices, each of which can be moved to any cluster that it does not occupy, i.e. one of $\mathcal{N}_C - 1$ clusters.

We consider several kinds of moves in the RNSC algorithm. A *global move* is a move that results in a near-optimal change in cost. A *diversification* move is one that in some way shuffles the clustering randomly. Diversification can be approached in a few different ways. An *intensification* move is a move with a good associated cost that is chosen from a restricted portion of the current clustering's neighbourhood. Intensification is not explored experimentally in this thesis, but it is discussed in this section. We also discuss issues related to forbidden (tabu) moves, which can help the algorithm to escape cycling and local minima.

First, some terminology: The set of possible clusterings on a graph G is denoted $\mathcal{C}[G]$. For a given clustering $\mathcal{C}$, the set of available moves from $\mathcal{C}$ is denoted $M[\mathcal{C}]$. The open neighbourhood of $\mathcal{C}$ in the search space (i.e. the set of clusterings that can be reached from $\mathcal{C}$ by making a single move in $M[\mathcal{C}]$) is denoted $N(\mathcal{C})$.

3.2.1 Global Moves

In RNSC, global moves are the key to improving the clustering and reducing the cost. A global move is defined as a random move with *near-optimal cost*, so we need to be clear about the meaning of "near-optimal cost", optimal cost being the greatest available decrease in clustering cost resulting from an available move.

As explained in Section 2.1.2, the change in the cost of a clustering resulting from a single move lies in $[1-|V|, |V|-1]$ for the naive cost function and in $(1-|V|, |V|-1)$ for the scaled cost function. In the naive scheme, "near-optimal cost" is equivalent to "optimal cost". When the clustering is good, there tend to be many optimal moves, so there is a large degree of randomness within this choice. In the scaled scheme, moves are effectively put in bins according to their value when rounded up to an integer. A near-optimal move, or a move with near-optimal cost, is one that is in the best nonempty bin, so the cost of a near-optimal move is within 1 of the cost of an optimal move. Just as in the naive scheme, there are usually many near-optimal moves when the clustering is good.

For the naive scheme, we define the sets of *equivalent-cost* moves as those moves which incur the same change in cost. For the scaled scheme, we define these sets as the sets of moves whose change in cost rounds up to the same integer.

We should consider, then, the inherent lack of randomness that occurs when the clustering is not good. In this case, the costs of moves that improve the clustering will have a wide distribution, so there will not be many near-optimal moves. However, the more a move improves the cost of a clustering, the more likely that move is to be effectively made in a near-optimal clustering, so in this sense the lack of randomness matters little. It is not so important that we have a wide variety of near-optimal moves to randomly choose from in this situation; they are all likely to be made eventually anyway.

3.2.2 Random Moves (Diversification)

The principle of diversification is very straightforward. In order to prevent circulation around a high local minimum, the algorithm makes some random moves to climb out of it, in hopes of settling to a lower minimum. This is similar in spirit to the idea behind simulated annealing, but here we wish to push the temperature up periodically rather than controlling the rate of cooling.

We consider two diversification schemes: In the first, we make a fixed number of moves with a given frequency. Each move places a random vertex in a random cluster, where both the vertex and the cluster are chosen from a uniform distribution.

In this scheme, there are two diversification parameters to consider. First is the diversification frequency $\mathcal{F}_D$. This is simply the length of our diversification period. Second is the diversification length $\mathcal{L}_D$. This is the number of diversification moves that are made at the end of the diversification period. So each cycle consists of $\mathcal{F}_D - \mathcal{L}_D$ global and intensification moves followed by $\mathcal{L}_D$ diversification moves. We call this scheme *shuffling diversification*.

The problem with shuffling diversification is the dimensionality of the clustering problem for large graphs. If RNSC uses this diversification scheme, what often happens is that many vertices are moved randomly during the diversification phase, most of which are unassociated with one another. As a result, the structure of the clusters changes very little. When the vertices are no longer tabu (see Section 3.2.4), they are often returned to their original positions, since in general doing this will improve the score a good deal. This behaviour turns out to be a huge problem with shuffling diversification, so we must amend our approach to address it.

The second scheme, which proves to be more sensible in most cases, chooses a fixed number of clusters and demolishes them, i.e. each vertex in these clusters is moved to a random cluster. By imposing certain tabu restrictions on the moved vertices, we can ensure that significant structural change is more likely to occur under this diversification scheme, which we call *destructive diversification*. Destructive diversification also involves making fewer moves than does shuffling diversification at similar levels of effectiveness, and moves become very costly in large graphs under the scaled cost scheme. Chapter 4 offers empirical evidence of these points.

3.2.3 Restricted Neighbourhood Moves (Intensification)

Intensification moves are moves that are chosen from a *restricted neighbourhood*, that is, a prescribed subset of the current clustering's neighbourhood in the search space. The idea behind intensification is to reduce the search for a good move by only considering moves within a certain subset of those possible moves. However, as seen in Section 3.4 and elsewhere, the bulk of computational work done by the RNSC algorithm involves not searching, but rather updating the space. This updating scheme used by RNSC eliminates the need to stochastically search for a good move. Rather, the algorithm has the moves sorted by their resultant change in cost, so searching the list is unnecessary. Although it is, for this reason, impractical to apply it to RNSC, we will describe one simple hypothetical intensification scheme. First, though, we must define our restricted neighbourhoods.

Definition 3.2.1. *Suppose we are searching for a good clustering. Let $\mathcal{C} \in \mathcal{C}[G]$ be our current clustering and let $M \in M[\mathcal{C}]$ be the movement of a vertex v from cluster C_i to cluster C_j. We define the* restricted neighbourhood induced by M*, denoted $\mathcal{R}(M)$, as the union of the following sets of moves:*

- *The movement of any vertex to or from C_i*
- *The movement of any vertex to or from C_j.*

Note that this includes any movement of v itself. We define these restricted neighbourhoods as such for the following reason:

Theorem 4. *Given a move $M \in M[\mathcal{C}]$ that moves a vertex v, $\mathcal{R}(M)$ is exactly the set of moves for which M may affect the resultant change in cost. This applies to both the naive cost function and the scaled cost function.*

The proof of this theorem is given at the end of the chapter after further background is given.

One scheme of intensification step, designed to reach local minima quickly, would be as follows:

1. While there is a cost-improving move
   (a) Make a global move M.
   (b) While there is a cost-improving move in $\mathcal{R}(M)$
      i. Make a near-optimal move M' chosen from $\mathcal{R}(M)$.
      ii. Set $M := M'$ and set $\mathcal{R}(M) := \mathcal{R}(M')$.

This would work well in a scheme under which it is necessary to search for near-optimal moves. However, we bypass such searching in favour of other methods.

3.2.4 Forbidden (Tabu) Moves

The general notion of tabu search as proposed by Glover in [27, 28] involves, as applied to the clustering problem, a forbidden set of clusterings defined as an explicit set, in relation to the current clustering, or in some other way. Because of the nature of the problem, it would be both ineffective and impractical to maintain an explicit set of tabu *clusterings*. Similarly, because of the high level of structural symmetry that is often seen in large instances, it is ineffective (and memory intensive) to maintain a list of tabu *moves*.

We can also consider forbidding the movement of a set of *vertices* for a certain period of time. It turns out that for large graphs, particularly sparse graphs (in which many clusters are needed), it makes far more sense to have forbidden vertices than it does to have forbidden moves. RNSC implements the tabu list as a first-in, first-out queue of vertices. That is, a vertex, once moved, is not to be moved until a certain number of other vertices have been moved.

The idea of the tabu list is to prevent cycling around a local minimum. In this spirit, we also forbid moving a vertex in a cluster of size 1 to an empty cluster. Such a move clearly has no effect on the clustering, but is a move which, in practice, appears with great frequency, enough so that it can spoil our choice of tabu length and waste significant amounts of computation time if not guarded against. As discussed in Section 3.3.2, we can generally give a good approximation of the number of clusters needed, so guarding against the redundant movement of singleton clusters needs not have a significant impact on computation time.

Having chosen our form of tabu list, we must choose our length of tabu list. We want the list to be long enough to prevent the algorithm from getting stuck in a high local minimum, but not so long that it hinders the algorithm's ability to return to another minimum. This must be handled with particular care to the diversification parameters and is discussed from a theoretical standpoint in Section 3.3.4 and from a practical standpoint in Chapter 4.

3.3 Parameters and Options

The RNSC method considers a number of different input parameters that can be tailored to suit the instance of the problem at hand. The length of the search, the maximum number of clusters, and the choice of moves to make, among others, are options that we must consider.

3.3.1 Search Length

Obviously in the presence of sufficient random shuffling (diversification) and cycle breaking, a local search algorithm is more likely to find a globally optimal solution the longer it searches. Unfortunately, it is in general impossible to tell what an optimal cost is. Furthermore, for large graphs an extravagant search length is prohibitively expensive in terms of computation time. What we need to do is find a balance between speed and thoroughness. In practice, our search length will be dictated by the rate of improvement of the best solution.

For the naive scheme, making a move is relatively fast and, near a local minimum, often largely inconsequential. When we are using the naive cost as a preprocessor, all we want is a *good* clustering, not necessarily one of the best. In this case, RNSC merely improves the naive cost without diversification until no improvement in minimum cost is found for $\mathcal{T}_n$ moves. $\mathcal{T}_n$ is the *naive stopping tolerance.*

Consider the example shown in Figure 3.4. The clustering in Figure 3.4b is an optimal naive clustering (as is the clustering in Figure 3.4a), so the naive scheme may choose it as the best clustering found, $\mathcal{C}_n$. However, the unique optimal scaled clustering is the one shown in Figure 3.4a. These two clusterings have no clusters in common, and the distance between them in the search space is i moves. This is an artificial example, but similar structure is often seen in sparse biological networks, for example the protein-protein interaction networks discussed in Section 4.1.2. Such structure is also described a)

b)

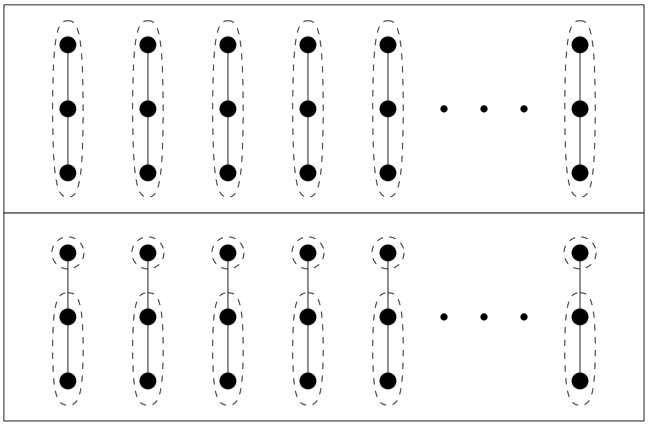

Figure 3.4: In this graph on $3i$ vertices, both clusterings are optimal in the naive scheme, with cost i. However, only the clustering in (a) is optimal in the scaled scheme. It has scaled cost $\frac{4}{6}(i-\frac{1}{3})$, whereas the clustering in (b) has scaled cost $\frac{5}{6}(i-\frac{1}{3})$. Note that the clusterings have no cluster in common.

in [50, 60], although not in the context of connected components of a graph.

For the scaled scheme, we generally use diversification and a tabu list. We dictate a scaled stopping tolerance $\mathcal{T}_s$. Experimental results for these tolerances are given in Chapter 4. In practice, increasing the experiment length beyond a certain point makes little difference.

3.3.2 Cluster Limits

RNSC operates based on an upper limit in the number of non-empty clusters. That is, there is a constant number of clusters $\mathcal{N}_C$, and the clusters can be empty. The number of clusters in a good clustering (i.e. one which is output as $\mathcal{C}_s$ by RNSC) varies depending on the structure of the graph, theoretically between 1 and $|V|$. Practical values are given in Chapter 4.

When considering the data structures used by RNSC in the current implementation (see Section 3.4), we can see that an excess of empty clusters will result in not only an unnecessarily large memory footprint, but also poor efficiency. This could lead to the conclusion that a dynamic cluster list would be more suited to the task, but this is not necessarily the case. We can approximate the number of clusters needed for the scaled scheme by using the much faster naive scheme. A dynamic data structure would be more complicated and costly to maintain. So if we can approximate (on the side of too many) the necessary number of clusters before the run of the algorithm, either with the naive scheme or with *a priori* knowledge of graphs with similar structural properties, it is better to use static data structures where the number of clusters is concerned.

3.3.3 Diversification

Recall from Section 3.2.2 that we have two types of diversification to consider: shuffling diversification and destructive diversification. If we choose to perform shuffling diversification, two integer parameters are needed: $\mathcal{F}_D$ and $\mathcal{L}_D$. $\mathcal{F}_D$ is the shuffling diversification frequency and $\mathcal{L}_D$ is the shuffling diversification length. That is, the last $\mathcal{L}_D$ of every $\mathcal{F}_D$ moves are random.

In general, the amount of diversification needed in this case is largely affected by two values: the number of clusters and the size of the graph. This is intuitive, because each random move chooses a vertex uniformly, then moves it to a uniformly chosen cluster. These moves have a greater effect if vertices are moved to or from smaller clusters, and if the graph has many vertices, each move will have a relatively small impact. This is supported by empirical data in Chapter 4.

As mentioned before, shuffling diversification makes a very small structural impact in large graphs unless a huge amount of diversification is done. For this reason, we favour destructive diversification. Destructive diversification takes one integer parameter, $\mathcal{F}'_D$, the destructive diversification frequency. Although we have the option to destroy any number of clusters upon reaching a diverification step, we choose to destroy only one at a time. So we will make $\mathcal{F}'_D$ global moves, then destroy a cluster, then make $\mathcal{F}'_D$ global moves, and so on. We destroy a cluster by choosing a vertex v uniformly at random, then moving every vertex in C_v to a uniformly chosen cluster.

The rationale behind choosing the cluster to destroy with probability proportional to its size (as is the effect when choosing by a uniform random vertex) lies in the relative impact of destroying a cluster. We want to choose to destroy an important cluster with high probablility. We then want to distribute its vertices to the other clusters uniformly.

Section 4.2.2 details our experimental results with respect to diversification parameters. If we perform too much diversification, we risk hindering the algorithm's attempts to find a local minimum. If we do not perform enough, we increase our chances of settling in a high local minimum.

3.3.4 The Tabu List

We consider two parameters for the tabu list: the *tabu length* $\mathcal{L}_T$ and the *tabu tolerance* $\mathcal{T}_T$. As with our diversification parameters, we seek to strike a happy medium. On one side, if too few vertices are tabu, the list will have little effect; it will do little to prevent the algorithm from cycling. If this is the case, even good diversification parameter values will likely not be able to remedy the problem if there are many clusters. On the other hand, if too many vertices are tabu we may end up in a position where many desirable moves are tabu.

The tabu length is simply the length of the tabu list. In a simple tabu scheme (i.e. when $\mathcal{T}_T = 1$), the performance of the algorithm can be very sensitive to changes in $\mathcal{L}_T$ (see Chapter 4). To reduce this sensitivity and increase the depth of memory represented by the tabu list, we allow vertices to be placed on the tabu list in multiplicity, forbidding the movement of a vertex if it is in the tabu list $\mathcal{T}_T$ times.

Recall that because of the nature of the problem, our tabu list is a list of vertices, and therefore obviously cannot exceed $(\mathcal{T}_T \cdot |V|) - 1$ in length. Good values for $\mathcal{L}_T$ and $\mathcal{T}_T$ depend largely on the size of the graph, as well as the choice of diversification parameter

Structure	Size	Description
`AdjMatrix`	$n \times n$	Symmetric adjacency matrix
`AdjList`	n lists	Neighbourhood lists for each vertex
`Degree`	n	The degree of every vertex

Table 3.1: Static graph data structures.

values. The tabu list and diversification scheme can be very powerful when combined effectively, and that means that their parameter values respect one another. More detail is given in Chapter 4.

3.4 Data Structures

The RNSC algorithm uses many redundant data structures in order to save computation time. For example, the graph G is stored as both an adjacency matrix and an adjacency list. In this section, we describe the data structures used by RNSC and how they are used.

3.4.1 Graph Data Structures

The data structures relating to the graph G itself are very straightforward. The vertices are represented as the integers $0, 1, \ldots, n-1$ (where $n = |V|$), and the graph is stored both in the adjacency matrix format (`AdjMatrix`) and adjacency list format (`AdjList`). Although this is very space-inefficient, particularly for sparse graphs, we are more or less resigned to space requirements of $\mathcal{O}(n^2)$. Empirically, if the graph is sparse then $\mathcal{N}_C$ will have size on the order of n, in which case `DegreeInCluster`, `MoveList`, and `MoveListPtr` will have size on the order of n^2. If the graph is not sparse, then $m = |E|$ will have size on the order of n^2, in which case `AdjList` will have size on the order of n^2. Having only one of `AdjList` and `AdjMatrix` will result in slower performance for the algorithm.

RNSC also keeps a record of the degree of each vertex, held in `Degree`. These graph data structures are listed in Table 3.1.

3.4.2 Search Data Structures

The data structures related to the search itself are much more complex than the graph data structures. They fall into three categories: structures related to the current clustering $\mathcal{C}$, structures related to the current set of available moves $M[\mathcal{C}]$, and structures related to the tabu list.

Of the structures related to the current clustering $\mathcal{C}$, several are obvious and straightforward. Like the vertices, the clusters are represented by the integers $0, 1, \ldots, \mathcal{N}_C - 1$. `WhichCluster` simply stores the cluster that each vertex is in. Conversely, `ClusterList` is a set of $\mathcal{N}_C$ linked lists, each of which lists the vertices in a given cluster. So that we can update `ClusterList` quickly, we provide a pointer for each vertex v to its location in the applicable cluster C_v. These are stored in `ClusterListPtr`. `ClusterSize` is simply a vector containing the size of each cluster. Finally, it is helpful to store the number of neighbours a vertex v has in each cluster. This is stored in the $n \times \mathcal{N}_C$ matrix `ClusterDegree`. The cluster-related data structures are summarized in Table 3.2. `CostNumerator` and `CostDenominator` are vectors as shown in the table, and their use is described in Section 3.5.

Now, on to the structures related to $M[\mathcal{C}]$, the set of possible moves. Recall from the notion of *equivalent-cost moves* described in Section 3.2.1 that each move $M \in M[\mathcal{C}]$ causes a change in naive cost $\delta_n(M)$ and a change in scaled cost $\delta_s(M)$. The main data structure is `MoveList`. This is a linked list in which every node represents a move and contains three data fields: the vertex being moved, the destination cluster, and the associated change in cost, either $\delta_n(M)$ or $\delta_s(M)$.

We maintain this list in an order sorted by the moves' associated integer costs, least to greatest. If we are working under the naive scheme, this is simply $\delta_n(M)$. If we are

Structure	Size	Description
`ClusterList`	$\mathcal{N}_C$ lists	Element lists for each cluster
`ClusterListPtr`	n	Pointer to each vertex in `ClusterList`
`WhichCluster`	n	Each vertex's cluster number
`ClusterSize`	n	The size of each cluster
`ClusterDegree`	$n \times \mathcal{N}_C$	The number of neighbours each vertex has in each cluster
`CostNumerator`	n	$\alpha_v = \#^1_{\text{out}}(G, \mathcal{C}, v) + \#^0_{\text{in}}(G, \mathcal{C}, v)$ for each vertex
`CostDenominator`	n	$\beta_v = \lvert N(v) \cup C_v \rvert$ for each vertex

Table 3.2: Data structures related to the current clustering $\mathcal{C}$.

working under the scaled scheme, the integer cost is $\lceil \delta_s(M) \rceil$. So in the naive scheme the list *is* sorted by cost, and in the scaled scheme the list is not sorted by cost, but if a move M' follows a move M immediately in the list, then $\delta_s(M) - \delta_s(M') < 1$.

The size of $M[\mathcal{C}]$ is $n(\mathcal{N}_C - 1)$; we can move any vertex to any cluster that it is not already in. Hence the length of `MoveList` is $n(\mathcal{N}_C - 1)$. We use several data structures to manage this list quickly. `NumWithCost` gives the number of moves with each associated integer cost, and `FirstWithCost` gives a pointer to the first move in the list with a given

Structure	Size	Description
`MoveList`	$n(\mathcal{N}_C - 1)$	The list of possible moves, sorted by cost
`NumWithCost`	$2n$	The number of possible moves with given cost
`FirstWithCost`	$2n$	Pointers to the first `MoveList` node with given cost
`MoveListPtr`	$n \times \mathcal{N}_C$	Pointers to the `MoveList` node for each move (some are `NULL`)

Table 3.3: Structures related to the set of available moves $M[\mathcal{C}]$.

Structure	Size	Description
`TabuList`	$\mathcal{L}_T$	The cyclic tabu list
`TabuVertices`	n	The tabu status vector

Table 3.4: Tabu data structures.

associated integer cost. As shown in Section 2.1.2 and specifically proven in Theorem 1, each move has an associated integer cost between $1-n$ and $n-1$. Hence both of these structures have size $2n$, including a sentinel node at the end of the list. `MoveListPtr`, another large data structure, contains a pointer to each move in the list, as indexed by vertex and destination cluster. This is a necessity for managing `MoveList` quickly. Since we cannot move a vertex v from C_v to C_v, `MoveListPtr` contains v null pointers.

The data structures applicable to $M[\mathcal{C}]$ are listed in Table 3.3. Of course, the issue of maintaining these structures must be addressed (see Section 3.5).

The data structures related to the tabu list are simple. `TabuList` is a cyclic linked list of length $\mathcal{L}_T$ that contains vertex labels. `TabuVertices` holds the tabu status of each vertex (i.e. whether or not it is tabu). Initially, the list is empty. With each move made, the moved vertex is added to the list and it becomes tabu. When the list is full, each time a vertex is added to the list, one is removed (and its status returns to non-tabu). The effect is a straightforward first-in first-out queue of tabu vertices. The data structures are given in Table 3.4.

Miscellaneous Data Structures

The RNSC algorithm manages many other sundry data structures in order to monitor its progress, gather statistics, etc. None of these are particularly interesting or relevant to the algorithm itself, so they do not warrant further mention in this thesis.

3.5 Cost Functions: Data Maintenance

The key to RNSC is the fact that its data structures are tailored to the clustering problem, and they can be updated quickly. Without this property, we would be left with a basic local search algorithm.

It is useful at this point to reconsider our cost functions, as given in Equations 2.3 and 2.5, and assign a couple of terms. We define α_v, the *cost numerator* of v, to be $\#^1_{\text{out}}(G, \mathcal{C}, v) + \#^0_{\text{in}}(G, \mathcal{C}, v)$, and we define β_v, the *cost denominator* of v, to be $|N(v) \cup C_v|$. So we have

$$C_n(G, \mathcal{C}) = \frac{1}{2} \sum_{v \in V} \alpha_v \tag{3.1}$$

and

$$C_s(G, \mathcal{C}) = \frac{n-1}{3} \sum_{v \in V} \frac{\alpha_v}{\beta_v} \tag{3.2}$$

for our two cost functions. Note that these α_v and β_v values are stored in `CostNumerator` and `CostDenominator` as in Table 3.2. Although `CostNumerator` is applicable to the naive cost scheme, the two vectors are only used in the scaled scheme.

3.5.1 The Naive Cost Function

The naive cost function (given in Equation 2.3) is very simple to update and maintain. Recall Definition 3.2.1 and Theorem 4, which establish the definition of a move's *induced restricted neighbourhood* and its exact composition. How δ_n changes for every move in this neighbourhood is much easier to see than how δ_s changes; we discuss it in this section.

Initializing

Initializing the naive cost of a move is simple. If we are moving vertex v from C_i to C_j (as in Figure 3.5), it is not hard to see what the change in cost will be. Recall from Chapter 2 that the naive cost of a clustering $\mathcal{C}$ on G is the size of the symmetric difference between the edge set of G and the edge set of the graph *suggested by* $\mathcal{C}$. Our move changes $\mathcal{C}$ to

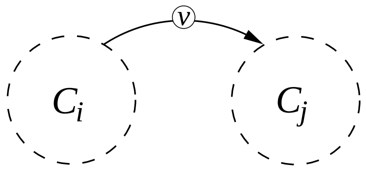

Figure 3.5: Moving vertex v from cluster C_i to cluster C_j.

a new clustering $\mathcal{C}'$, but the only areas where this symmetric difference will change are edges (in the difference) incident to v.

The number of edges incident to v in the symmetric difference between the edge sets of G and the graph suggested by $\mathcal{C}'$ is $\alpha_v = ((|V(C_i)| - 1) - |N_i(v)|) + (|N(v)| - |N_i(v)|)$, where $N(v)$ is the open neighbourhood of v and $N_i(v)$ is the open neighbourhood of v in C_i, i.e. $N(v) \cup V(C_i)$. The first term in the sum is the number of vertices in C_i that are not adjacent to v. The second term is the number of vertices adjacent to v that are not in C_i.

Once we have established this fact, it becomes clear that the change in cost caused by our move, call it M, is simply the difference between α_v after the move and α_v before the move. Substituting C_j for C_i in the last equation gives us the number of such bad edges after the move, so our change in naive cost $\delta_n(M)$ is

$$\begin{aligned}\delta_n(M) &= \quad (|V(C_j)| + |N(v)| - 2|N_j(v)| - 1) \\ &\quad - (|V(C_i)| + |N(v)| - 2|N_i(v)| - 1) \\ &= |V(C_j)| - |V(C_i)| + 2(|N_i(v)| - |N_j(v)|). \end{aligned} \tag{3.3}$$

We have all of these figures stored in `ClusterSize`, `ClusterDegree`, and `Degree`. We do not need to search these, so initializing the naive cost for the moves in $N(\mathcal{C}_0)$ is fast and simple.

Updating

Let M be our move, and suppose it moves v from cluster C_i to cluster C_j. Then our restricted neighbourhood, $\mathcal{R}(M)$, consists of any move to or from C_i or C_j, including, of course, any movement of v itself. Let M' be a movement other than M. Let $\delta_n(M')$ be the change in naive cost that M' would cause before M, and let $\delta'_n(M')$ be the change in naive cost that M' would cause after M.

Suppose M' moves vertex u from cluster C_k to cluster C_l. What we need to consider is the difference between $\delta'_n(M')$ and $\delta_n(M')$. From Equation 3.3, we can see that if C_i, C_j, C_k, and C_l are distinct clusters, then M will have no effect on $\delta_n(M')$. We will consider the changes case by case. Assume that $u \neq v$ for now. Note that $\sim$ denotes adjacency and $\not\sim$ denotes nonadjacency.

- If $C_k = C_i$, $|V(C_k)|$ will decrease by 1 under M, and $|N_k(u)|$ will decrease by 1 if and only if $u \sim v$.
- If $C_k = C_j$, $|V(C_k)|$ will increase by 1 under M, and $|N_k(u)|$ will increase by 1 if and only if $u \sim v$.
- If $C_l = C_i$, $|V(C_l)|$ will decrease by 1 under M, and $|N_l(u)|$ will decrease by 1 if and only if $u \sim v$.
- If $C_l = C_j$, $|V(C_l)|$ will increase by 1 under M, and $|N_l(u)|$ will increase by 1 if and only if $u \sim v$.

No other figures applicable to Equation 3.3 will change. From the equation, we can see that these points will cumulate as shown in Table 3.5. Suppose $u = v$, then. It is clear that the cost after M' will be the same whether or not M is done, so in this case $\delta'_n(M') - \delta_n(M') = -\delta_n(M)$.

So we know exactly how to update the cost changes. Because of the setup of the data structures, we can update in a small constant amount of time per move in $\mathcal{R}(M)$. Note

$C_k = C_i$?	$C_k = C_j$?	$C_l = C_i$?	$C_l = C_j$?	$u \sim v$?	$\delta'_n(M') - \delta_n(M')$
✓	×	×	×	✓	-1
✓	×	×	×	×	$+1$
✓	×	×	✓	✓	-2
✓	×	×	✓	×	$+2$
×	✓	×	×	✓	$+1$
×	✓	×	×	×	-1
×	✓	✓	×	✓	$+2$
×	✓	✓	×	×	-2
×	×	✓	×	✓	$+1$
×	×	✓	×	×	-1
×	×	×	✓	✓	-1
×	×	×	✓	×	$+1$

Table 3.5: The effect of a move M on the change in naive cost associated with another move M'.

that the size of $\mathcal{R}(M)$ is

$$\begin{aligned} |\mathcal{R}(M)| &= |V(C_i)| \cdot (\mathcal{N}_C - 2) + |V(C_j)| \cdot (\mathcal{N}_C - 2) + (n - |V(C_i)|) + (n - |V(C_j)|) \\ &= (|V(C_i)| + |V(C_j)|) \cdot (\mathcal{N}_C - 3) + 2n. \end{aligned} \tag{3.4}$$

$|C_i| \cdot (\mathcal{N}_C - 2)$ is the number of moves from C_i to another vertex other than C_j. Likewise $|V(C_j)| \cdot (\mathcal{N}_C - 2)$ is the number of moves from C_j to another vertex other than C_i. Finally, $(n - |V(C_i)|) + (n - |V(C_j)|)$ is the number of moves *into* C_i or C_j. Note that the average cluster size is $n/\mathcal{N}_C$, so this update, on average, updates around $4n$ moves. Each individual update can be done very quickly once a move is made.

3.5.2 The Scaled Cost Function

The scaled cost function, unfortunately, cannot be maintained as quickly or easily. Considerably more work is needed, both to initialize and to update after a move. The restricted neighbourhood induced by a move M is the same for the scaled cost function as for the naive cost function, but the costs in the move list are much more expensive to manipulate.

Initializing

We will use Equation 3.2 to make things clearer when considering the scaled cost. The first thing to do when we initialize the move list is to initialize the cost numerator and the cost denominator, α_v and β_v. Let v be in C_i. Note that β_v is simply $d(v)+|V(C_i)|-N_i(v)$, so we can find it very easily. Similarly, $\alpha_v = |V(C_i)| + d(v) - 2|N_i(v)| - 1$, so computing α_v is not a problem either. They must both be maintained appropriately when a move is made.

From Section 3.5.1, we know that if we make a move M that moves v from C_i to C_j, α_u will only change for $u \in C_i \cup C_j$, including v. It is also easy to see that β_u will only change if u is in C_i or C_j. Call the cost numerator for u after M is made α'_u. Call the

$u \in C_i$?	$u \in C_j$?	$u \sim v$?	$\alpha'_u - \alpha_u$	$\beta'_u - \beta_u$	$\frac{\alpha'_u\beta_u - \alpha_u\beta'_u}{\beta'_u\beta_u}$
✓	×	✓	$+1$	0	$+1/\beta_u$
✓	×	×	-1	-1	$(\alpha_u - \beta_u)/(\beta_u^2 - \beta_u)$
×	✓	✓	$+1$	0	$-1/\beta_u$
×	✓	×	-1	$+1$	$(\beta_u - \alpha_u)/(\beta_u^2 + \beta_u)$

Table 3.6: The effect of a move M on α_u and β_u.

cost denominator for u after M is made β'_u. Now we can consider $\delta_s(M)$, the change in scaled cost caused by M, as

$$\delta_s(M) = \frac{n-1}{3} \sum_{u \in C_i \cup C_j} \left(\frac{\alpha'_u}{\beta'_u} - \frac{\alpha_u}{\beta_u} \right) = \frac{n-1}{3} \sum_{u \in C_i \cup C_j} \left(\frac{\alpha'_u\beta_u - \alpha_u\beta'_u}{\beta'_u\beta_u} \right) \tag{3.5}$$

This would be simple to calculate were it not for the fact that we don't know what α'_u and β'_u are. Assuming that $u \neq v$, they are as given in Table 3.6. If $u = v$, then we can compute α'_v and β'_v just as we computed α_v and β_v.

As we can see from the values of the table, initializing these costs is considerably more troublesome for the scaled cost scheme than it is for the naive cost scheme. This difficulty will carry over to the task of updating the move list.

Updating

As with the naive cost function, we must update $\delta_s(M')$ for all moves M' in the restricted neighbourhood $\mathcal{R}(M)$ when we make a move M. Here, let M' move u from C_k to C_l where M moves v from C_i to C_j. Let α'_w and β'_w be the cost numerator and denominator of w after M' as they would be without M being made. Let α''_w and β''_w be the cost numerator and denominator of w after M' is made following M. Let $\delta'_s(M')$ be the change in cost caused by M' after move M is made.

Following Equation 3.5, then, we get

$$\begin{aligned}
\delta'_s(M') &= \frac{n-1}{3}\sum_{w\in C_k\cup C_l}\left(\frac{\alpha''_w}{\beta''_w}-\frac{\alpha'_w}{\beta'_w}\right)\\
&= \delta_s(M')+\frac{n-1}{3}\sum_{w\in C_k\cup C_l}\left(\frac{\alpha''_w}{\beta''_w}-\frac{2\alpha'_w}{\beta'_w}+\frac{\alpha_w}{\beta_w}\right) \qquad (3.6)
\end{aligned}$$

It is, in fact, faster to recompute $\delta'_s(M')$ from scratch using our stored α and β values than it is to actually update it. Hence, the process of updating the costs is simplified by the demands of the problem. It is this updating stage that makes the scaled cost scheme so much slower than the naive cost scheme.

Theorem 4 Revisited

At this time we will restate and prove Theorem 4 from Section 3.2.3.

Theorem. *Given a current clustering $\mathcal{C}$ and a move $M \in M[\mathcal{C}]$ that moves a vertex v, $\mathcal{R}(M)$ is exactly the set of moves for which M may affect the resultant change in cost (unless M does not change the cost of the clustering, in which case moves involving v will remain unchanged). This applies to both the naive cost function and the scaled cost function.*

Proof. Let M move v from C_i to C_j. Then $\mathcal{R}(M)$ is defined as any move that moves a vertex to or from C_i or C_j. Consider M', which moves u from C_k to C_l. If C_i, C_j, C_k, and C_l are distinct clusters, we can see from Equation 3.3 that $\delta_n(M')$ will not change when the move M is made. Also when the four clusters are distinct, consider Equation 3.5 for $\delta_x(M')$. We can see that when M is made neither α_w nor β_w changes for any $w \in C_k \cup C_l$. The same is true for α'_w and β'_w. Therefore $\delta_s(M')$ will not change.

We must now show that $\delta_n(M')$ and $\delta_s(M')$ can change for $M' \in \mathcal{R}(M)$. If $u = v$, then $\delta_n(M') = \delta'_n(M') + \delta_n(M)$, and $\delta_s(M_v) = \delta'_s(M_v) + \delta_s(M)$, so if $\delta_n(M) \neq 0$ then $\delta_n(M')$ will change, and if $\delta_s(M) \neq 0$ then $\delta_s(M')$ will change. These equations come from the fact that the cost of a clustering is invariant of the set of moves used to reach it.

If $u \neq v$, then Table 3.5 shows that $\delta_n(M')$ will change for every $M' \in \mathcal{R}(M)$. A trivial example suffices to show that $\delta_s(M')$ can change. Consider a graph $G = (\{v_1, v_2\}, \emptyset)$ and a clustering $\mathcal{C}$ of G consisting of an empty cluster C_1 and a cluster C_2 containing both vertices. Let M move v_1 into C_1. $\mathcal{R}(M)$ consists of moving v_1 to C_2 and moving v_2 into C_1, moves that originally would not reduce the cost of the clustering. After M is made, both reduce the cost. $\square$

3.6 Colouring Graphs with RNSC

RNSC was developed as a graph clustering algorithm. However, because the cost functions are reasonably flexible, the algorithm can be easily modified to search for a proper colouring of a graph. Note that clustering G is akin to colouring the complement $\overline{G}$. Henceforth when we discuss colouring, it will be in the graph G, not $\overline{G}$.

3.6.1 The Colouring Cost Function

Recall the form of the naive cost function given in Chapter 2:

$$C_n(G, \mathcal{C}) = \frac{1}{2} \sum_{v \in V} \left(\#^1_{\text{out}}(G, \mathcal{C}, v) + \#^0_{\text{in}}(G, \mathcal{C}, v) \right)$$

We can modify this very easily so that a cost of zero corresponds with a clustering in which all clusters are actually independent sets. We call this cost function the *colouring cost function* $C_C(G, \mathcal{C})$, and define it as

$$C_c(G, \mathcal{C}) = \sum_{i=1}^{\mathcal{N}_C} \binom{|V(C_i)|}{2} - \frac{1}{2} \sum_{v \in V} \#^0_{\text{in}}(G, \mathcal{C}, v). \tag{3.7}$$

That is, $C_c(G, \mathcal{C})$ is the total number of edges whose endpoints are in the same cluster. We can then consider the clusters to be colour classes, and the colouring is proper if and only if the colouring cost is 0. In this way, we can run the coloured cost scheme, which runs even faster than the naive cost scheme, in search of a proper $\mathcal{N}_C$-colouring of G.

We must now discuss how to initialize and update the change in colouring cost resulting from a move M. Both problems can be solved through simple adaptations of the naive analogues.

Initializing

Consider a move M that moves vertex v from C_i to C_j. M changes the number of edges within C_i by $-N_i(v)$ and changes the number of edges within C_j by $N_j(v)$. Therefore we have the change in colouring cost due to M, denoted $\delta_c(M)$. It is

$$\delta_c(M) = N_j(v) - N_i(v). \tag{3.8}$$

We can therefore initialize the cost for each move very easily.

Updating

Updating the move list for the colouring cost scheme is easier than updating in the naive scheme. We already have in mind our move M, which moves vertex v from C_i to C_j. Following the method of updating the naive cost scheme move list, we consider a move M', which moves vertex u from C_k to C_l. We wish to compare $\delta_c(M')$, the change in cost resulting from M' before M is made, to $\delta_c'(M')$, the change in cost resulting from M' after M is made. As we can see from Equation 3.8, we need only consider changes in $N_k(u)$ and $N_l(u)$. Assume for now that $u \neq v$.

As with the naive cost scheme, we consider when $N_k(u)$ and $N_l(u)$ will change.

- If $C_k = C_i$, $|N_k(u)|$ will decrease by 1 if and only if $u \sim v$.

- If $C_k = C_j$, $|N_k(u)|$ will increase by 1 if and only if $u \sim v$.

- If $C_l = C_i$, $|N_l(u)|$ will decrease by 1 if and only if $u \sim v$.

$C_k = C_i$?	$C_k = C_j$?	$C_l = C_i$?	$C_l = C_j$?	$u \sim v$?	$\delta'_c(M') - \delta_c(M')$
✓	×	×	×	✓	+1
✓	×	×	✓	✓	+2
×	✓	×	×	✓	−1
×	✓	✓	×	✓	−2
×	×	✓	×	✓	−1
×	×	×	✓	✓	+1

Table 3.7: The effect of a move M on the change in colouring cost associated with another move M'.

- If $C_l = C_j$, $|N_l(u)|$ will increase by 1 if and only if $u \sim v$.

Since $\delta'_c(M') - \delta_c(M') = (N'_l(u) - N_l(u)) - (N_k(u) - N'_k(u))$, we get the changes in $\delta_c(M')$, as given in Table 3.7. Note that these changes are the opposite of the corresponding changes in $\delta_n(M')$ if $u \sim v$, and 0 if $u \not\sim v$. Therefore updating the move list in the colouring cost scheme can be done quickly and easily.

Because colouring is not the primary function of the RNSC algorithm, it is not explored very deeply in the experiments, but some results are given in Chapter 4.

3.6.2 Heuristic Concerns

As in the naive clustering scheme, we use destructive diversification and a tabu list of vertices when colouing with RNSC. Avanthay *et al.* developed a very similar colouring algorithm that incorporates a number of heuristic diversification scheme [4]. One of these, their *empty-refill class neighbourhood* strategy, is very similar to destructive diversification. The issue of choosing suitable parameter values for RNSC colouring is addressed in Chapter 4.

3.7 The Complexity of a Move

This section details the computational and complexity issues of making a move within RNSC, using the algorithm and data structures discussed in this chapter.

Choosing a Move

Choosing a diversification move is much easier than choosing a global move. In the case of shuffling diversification, we merely choose a vertex at random, then choose a new cluster at random to move the vertex to. Obviously this can be done in constant time. In the case of destructive diversification, we simply do this for every vertex in a given cluster. If a chosen vertex is tabu, we move it anyway; moving a tabu vertex in a diversification move does not prevent the tabu list from performing its function, which is to help the RNSC algorithm climb out of minima.

When choosing a global move, we are concerned with choosing a move with low change in cost. We are also concerned with choosing the move with a certain amount of randomness and ensuring that the move is not tabu. Since choosing a move takes a negligible amount of time relative to updating the data structures, we can be a little bit extravagant. We make a given number of attempts to make a random near-optimal move. If we cannot find one that is not tabu, we choose the first non-tabu move in the list.

In practice, the number of attempts does not need to be large. if there are many moves in a cost bin (i.e. with the same integer cost), then the odds of a given move in the bin being tabu are small (our tabu list is generally short). If there are only a few, then we are likely to find a non-tabu move in the bin randomly if one exists. In practice, 20 is a reasonable number of attempts.

Updating in the Naive Cost Scheme

Updating is fairly fast in the naive cost scheme. As mentioned in Section 3.5.1, the size of $\mathcal{R}(M)$ for a move M is $(|V(C_i)| + |V(C_j)|) \cdot (\mathcal{N}_C - 3) + 2n$, which is on average around $4n$. As seen in Table 3.5, we know exactly how to update the cost for a move in $\mathcal{R}(M)$ if the vertex being moved is not v, the vertex moved by M. If the vertex being moved is v, we simply subtract $\delta_n(M)$ from the cost.

Because `MoveList` is a linked list and we have `MoveListPtr` helping us find entries quickly, we can update `MoveList` in constant time for every node in the list that is moved. We also modify `FirstWithCost`, `NumWithCost`, and `MoveListPtr` in constant time as needed. In the naive scheme, `ClusterDegree` is the only non-trivial data structure that needs to be updated. Note that it only changes for neighbours of v: it increases by one for C_j and decreases by one for C_i.

So updating the move list occupies the overwhelming majority of computation time in updating the naive data structures. The complexity of a move in the naive cost scheme is $\mathcal{O}(n)$, the size of $\mathcal{R}(M)$.

Updating in the Scaled Cost Scheme

The scaled cost scheme is much more time consuming to update. In general, the data structures are updated in the same way as they are in the naive cost scheme. However, calculating the change in $\delta_s(M')$ for a move in $\mathcal{R}(M)$ is expensive, and unfortunately must be done for every such M' (of which, as in the naive cost scheme, there are around $4n$ on average).

To find the new value of $\delta_s(M')$ for a move in $\mathcal{R}(M)$, we simply recalculate it as before. Given that M' moves u from C_k to C_l, this involves calculating the values given in Table 3.6 $|V(C_k)| + |V(C_l)|$ times. $\mathcal{R}(M)$ consists of moving anything to or from C_i

or C_j. Considering moves to C_i or C_j, we get

$$\begin{aligned}\sum_{M' \in \mathcal{R}(M)} |V(C_k)| + |V(C_l)| &= (n - |V(C_i)|)|V(C_i)| + (n - |V(C_j)|)|V(C_j)| + \\ &\quad \sum_{k \neq i} |V(C_k)|^2 + \sum_{k \neq j} |V(C_k)|^2. \end{aligned} \tag{3.9}$$

Considering moves from C_i or C_j (not including moves between the two, which we have already counted), we get

$$\begin{aligned}\sum_{M' \in \mathcal{R}(M)} |V(C_k)| + |V(C_l)| &= (n - |V(C_i)|)|V(C_i)| + (n - |V(C_j)|)|V(C_j)| + \\ &\quad \mathcal{N}_C(|V(C_i)| + |V(C_j)|) \end{aligned} \tag{3.10}$$

The value in Equation 3.9 is bounded by $\mathcal{O}(n^2)$, as is the value in Equation 3.10. In practice, they are not generally as large as this.

It is relevant at this time to note that after the naive cost scheme has been run, most global moves involve small clusters, as large clusters tend to be more stable in the transition between the two cost functions. This is particularly important when clustering scale-free graphs, in which there tend to be many small clusters and a few clusters that are much larger than the rest.

Note that the values in Table 3.6 are computations that are repeated many times. α_v is at most $n - 1$ and β_v is at most n. Hence updating in the scaled cost scheme can be expedited significantly with the use of lookup tables.

Chapter 4

Experimental Results

4.1 Test Data

For the purposes of analyzing the performance of the RNSC algorithm, we consider several types of graphs. Some types are randomly generated, and some are deterministically generated. We use the deterministically generated graphs for colouring; the graphs that we generate are typically hard to colour. Since there is no discrete quality of a clustering being "good" or "bad", there is no equivalent notion of a graph being "hard to cluster". However, it is worth noting that MCL, for one, performs poorly on sparse geometric graphs.

We also test the performance of RNSC on known data. For clustering, we use protein-protein interaction networks for yeast, fruitfly, and worm. For colouring, we attempt to colour a design-based graph, as well as several DIMACS benchmark cases.

4.1.1 Random Graphs

We consider three types of random graphs: Erdös-Rényi random graphs, scale-free random graphs, and geometric random graphs.

Erdös-Rényi Graphs

Erdös-Rényi graphs (see [19]) result from the first and most basic form of random graph construction. Such a graph is constructed from an independent set by considering every pair of vertices in the graph and connecting them with probability p, where $0 \leq p \leq 1$. We denote such a graph of order n by $G_E(n, p)$.

Scale-Free Graphs

Scale-free graphs are believed to be good general models for certain types of biological graphs, worldwide web graphs, and other naturally-occurring networks [7, 8, 37]. In scale-free graphs, vertex degrees tend to follow a power-law distribution, meaning that there are many vertices with very small degree, and a small number of *hubs*, which have relatively very high degree [7].

We use a simple formulation of scale-free graphs. Given integers n and k such that $n > k$, we construct the scale-free random graph $G_S(n, k)$ as follows:

We number the vertices $1, 2, \ldots, n$ and let $G^{(k)}$ be an independent set containing the first k vertices. For $i = k+1, k+2, \ldots n$, we construct $G^{(i)}$ from $G^{(i-1)}$ by adding vertex i to the graph and joining it to k random vertices in $G^{(i-1)}$, choosing a vertex v with probability proportional to $1 + deg_{G^{(i-1)}}(v)$ and not allowing multiedges. That is, we put each vertex v in a bucket $1 + deg_{G^{(i-1)}}(v)$ times and draw k nonidentical vertices. So $G^{(k+1)}$ is a claw, and $G_S(n, k) = G^{(n)}$ is always connected. $G_S(n, k)$ contains $k(n-k)$ edges.

Because of the way we attach new vertices to old vertices, those vertices that have high degree are likely to have higher and higher degree as vertices are added, while those vertices with low degree are not likely to be attached to added vertices. This preferential attachment of added vertices results in the power-law degree distribution, with the emergence of a small number of hubs and many low-degree vertices [7].

Geometric Graphs

Geometric random graphs make good test data for several reasons: First, they are suitable models for such classic combinatorial optimization problems as unweighted TSP. Second, they are easy to visualize. For example, given a clustering of a two-dimensional geometric graph, it is easy to see, to an extent, whether or not the clustering is suitable. Finally, recent research by Pržulj *et al.* has shown that geometric graphs are in some ways a better model for protein-protein interaction networks than the currently popular model of scale-free graphs [55].

We run the RNSC algorithm on two-dimensional geometric graphs. The vertices are n locations on an $l \times w$ grid chosen using a uniform distribution. Two vertices are adjacent if the Euclidean distance between them is less than d. We denote such a random graph $G_G(n, l, w, d)$.

4.1.2 Protein-Protein Interaction Networks

Protein-protein interaction (PPI) networks are biological networks whose vertices are proteins in an organism's proteome (the set of proteins produced by a cell) and whose edges are interactions between the proteins. Clustering these graphs effectively can lead to better understanding of the proteome through protein complex identification and functional grouping [5, 8, 37, 41, 54, 55, 56, 66].

Von Mering *et al.* provide four PPI graphs of varying size and accuracy for the yeast (*S. cerevisiae*) proteome. The smallest has 988 vertices and 2455 edges, while the largest has 5321 vertices and 78,390 edges [66]. From an interaction set given in [26], we derive two PPI graphs for the fruitfly (*D. melanogaster*) proteome: one containing all interactions, and one containing interactions with confidence level of at least 0.5. Li *et al.* provide a PPI graph for the worm (*C. elegans*) proteome. Table 4.1 presents a summary of these graphs.

Description	Name	$\|V\|$	$\|E\|$	Density	Source
Yeast 1	Y_{2k}	988	2455	5.035×10^{-3}	[66]
Yeast 2	Y_{11k}	2401	11,000	3.818×10^{-3}	[66]
Yeast 3	Y_{45k}	4687	45,000	4.098×10^{-3}	[66]
Yeast 4	Y_{78k}	5321	78,390	5.538×10^{-3}	[66]
Fly 1	F_{5k}	4602	4637	4.380×10^{-4}	[26]
Fly 2	F_{20k}	6985	20,007	8.202×10^{-4}	[26]
Worm 1	W_{5k}	3115	5222	1.077×10^{-3}	[44]

Table 4.1: Protein-protein interaction graphs clustered by RNSC.

4.1.3 Colouring Benchmark Graphs

For colouring algorithms, the most difficult graphs to colour are often those that are highly structured and highly symmetric. We use the RNSC algorithm to colour three such types of graphs.

Johnson Graphs

The *Johnson graph* $J(n, w, d) = (V, E)$ is defined as follows. The vertex set V is the set of binary strings of length n with weight w (i.e. there are exactly w `1`s in each string). Two vertices in the Johnson graph are adjacent if and only if the *Hamming distance* between them (the number of positions in which the strings differ) is less than d.

In a proper colouring of $J(n, w, d)$, each colour class is a binary code of length n, weight w, and distance d. Finding the chromatic number of these graphs is closely related to the problem of finding maximal-size codes with the given specifications n, w, and d, which is discussed in [61].

Often the complementary definition of the Johnson graph is given, in which edges exist when vertices have Hamming distance at least d. In this case the problem is similar to that of finding large cliques. However, in the cases that we consider, the density of the Johnson graph is less than 0.5, meaning that our definition contains fewer edges and therefore RNSC runs faster.

Several Hamming graphs (like Johnson graphs but with all 2^n binary strings of length n in the vertex set) appear in the DIMACS *clique finding* benchmark set [62]. These graphs use the complementary definition discussed above.

Leighton Graphs

The *Leighton graphs* are random graphs with a perscribed number of vertices and a prescribed chromatic number, as constructed in [43]. Twelve such graphs appear in the DIMACS graph colouring benchmark set [62]. Each of these benchmark graphs has density less than .2.

A Design-Based Graph

We attempt to use the RNSC algorithm to decompose the triples of a 9-set into a minimum number of mutually orthogonal resolutions.

Consider the set of 84 triples of a 9-set. Now consider triples of these triples whose union spans the 9-set. That is, disjoint triples of these subsets, which we can call super-triples. A given triple is contained in $\binom{9-3}{3}/2 = 10$ super-triples. Hence the number of these super-triples is $(84 \times 10)/3 = 280$. Let these 280 super-triples be the vertices of our graph G_{Triple}, and let two vertices be adjacent if and only if they share a triple.

We wish to colour G_{Triple}, which is the incidence graph of the design described above (the triples are the points, and the super-triples are the blocks). There exists a 10-colouring of this graph, where each colour class is a resolution of 3-subsets of a 9-set. This colouring is an extremal configuration, the unique 10-colouring.

4.2 Parameter Training and Statistical Results

We want to know what parameter values result in good performance for the RNSC algorithm. We must therefore examine empirical results, coupled with common sense, to find good values to choose for a given graph type. These include the cluster limit $\mathcal{N}_C$, diversification parameters $\mathcal{F}_D$, $\mathcal{L}_D$, and $\mathcal{F}'_D$, and the search and tabu length parameters $\mathcal{T}_n$, $\mathcal{T}_s$, $\mathcal{L}_T$, and $\mathcal{T}_T$.

4.2.1 Cluster Limits

The parameter that is basically independent of the other parameters, and should therefore be investigated first, is $\mathcal{N}_C$, the maximum number of clusters. To examine the number of clusters emerging in the RNSC algorithm, we used a test bed of generated Erdös-Rényi, scale-free, and geometric random graphs. The Erdös-Rényi graphs used were $G_E(300, 10^{-x/5})$ for $x = 1, 2, \ldots, 20$. The scale-free graphs used were $G_S(500, x^2)$ for $x = 1, 2, \ldots, 16$, and the geometric graphs used were $G_G(500, 100, 100, x)$ for $x = 1, 2, \ldots, 10, 15, 20, 30, 40, 50$.

Good values of $\mathcal{N}_C$ are highly dependent on the graph that is being clustered. For example, in the PPI graphs, each of which has very low density, the number of clusters in a good clustering can be greater than $n/2$. Although it may seem counterintuitive to have many singleton clusters in a good clustering of a graph with no isolated vertices, this simply indicates that a vertex has no overwhelming connection to any other single cluster. For example, consider a large clique with a pendant vertex v adjacent to a member. Even though every neigbour of v is in the clique, which would naturally be a cluster, it is not adjacent to a significant number of vertices in the clique. Hence v will remain a singleton cluster to preserve the density of the other cluster.

Figures 4.1, 4.2, and 4.3 show the number of clusters in the final clustering for Erdös-Rényi, scale-free, and two-dimensional geometric random graphs of varying density on

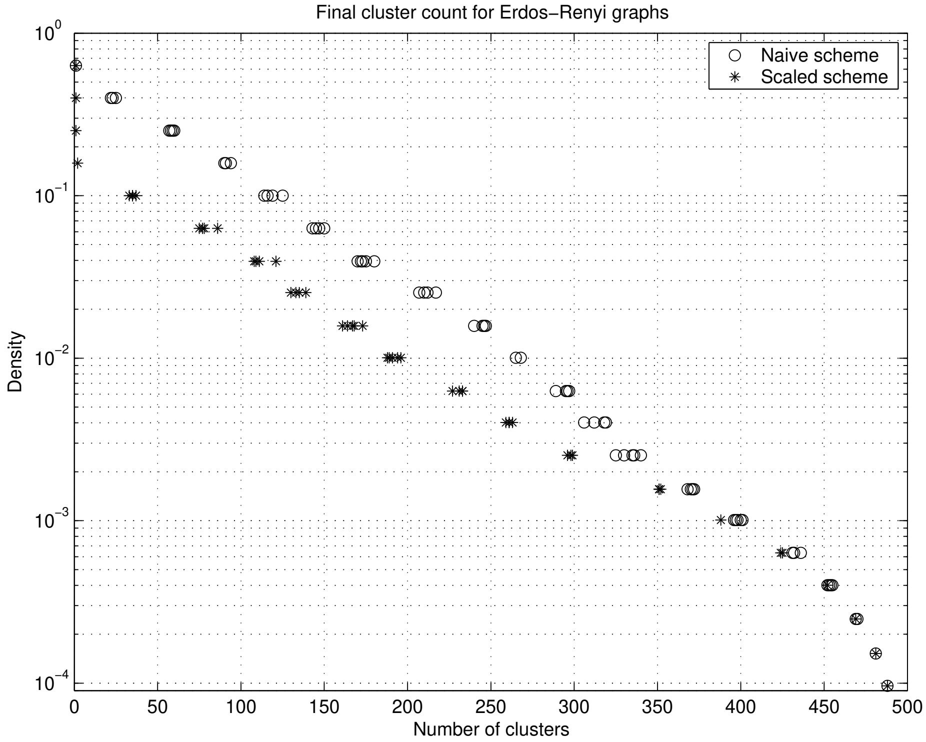

Figure 4.1: Final cluster counts for Erdös-Rényi graphs of varying density on 500 vertices.

500 vertices, respectively. The number of clusters is shown for each $\mathcal{C}_n$ and $\mathcal{C}_s$ over five experiments.

What we can see immediately is that the number of clusters in $\mathcal{C}_n$ is generally greater than the number of clusters in C_s, and never smaller. Let us consider a smaller example, an instance of $G_G(200, 100, 100, 20)$ shown in Figure 4.4. We call this instance $G_G^{200,d}$. In a trial run of RNSC, $\mathcal{C}_N$ contains 21 clusters and $\mathcal{C}_S$ contains only 13. These clusterings are shown in Figure 4.5 along with the result of the MCL algorithm applied to the same graph. The naive, scaled, and MCL clusterings have naive costs 1001, 1073, and 1987 and scaled costs 5785.3, 5348.6, and 6523.4 respectively. Clearly the RNSC output is better than the MCL output according to both performance criteria.

When we add a vertex v (a singleton cluster) to a large cluster C of which only a small fraction is in $N(v)$, α_v increases dramatically and α_u increases by one for every

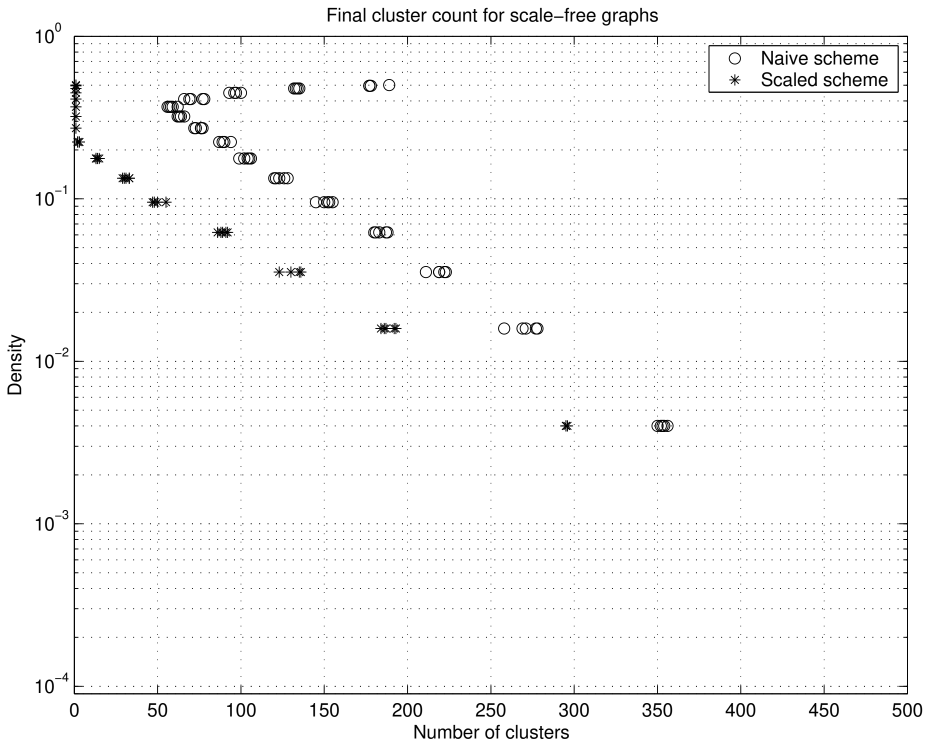

Figure 4.2: Final cluster counts for scale-free graphs of varying density on 500 vertices.

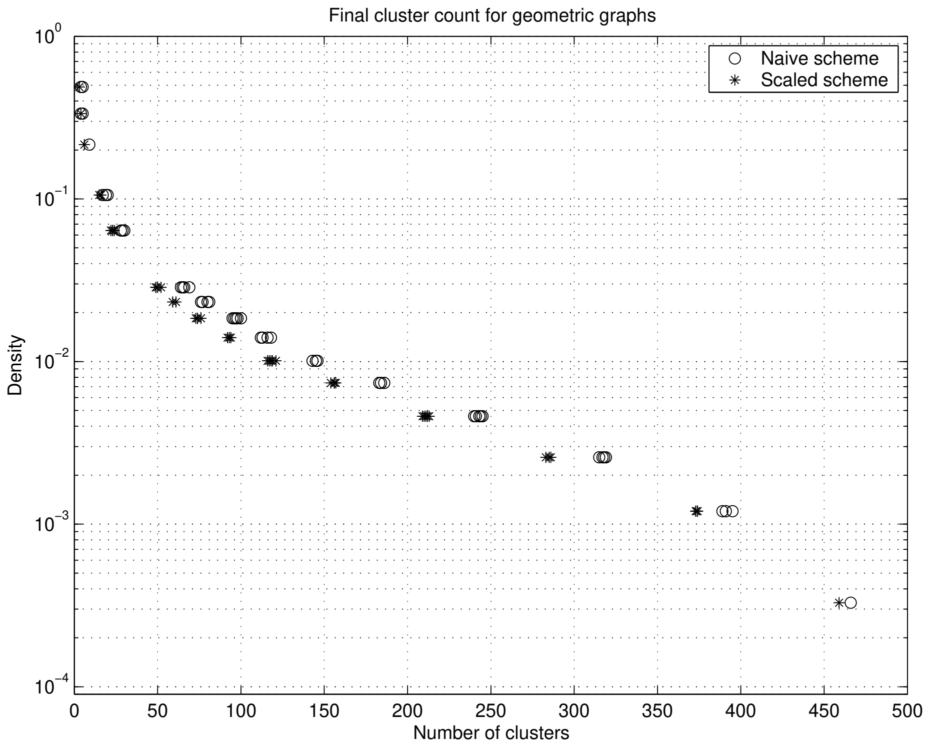

Figure 4.3: Final cluster counts for geometric graphs of varying density on 500 vertices.

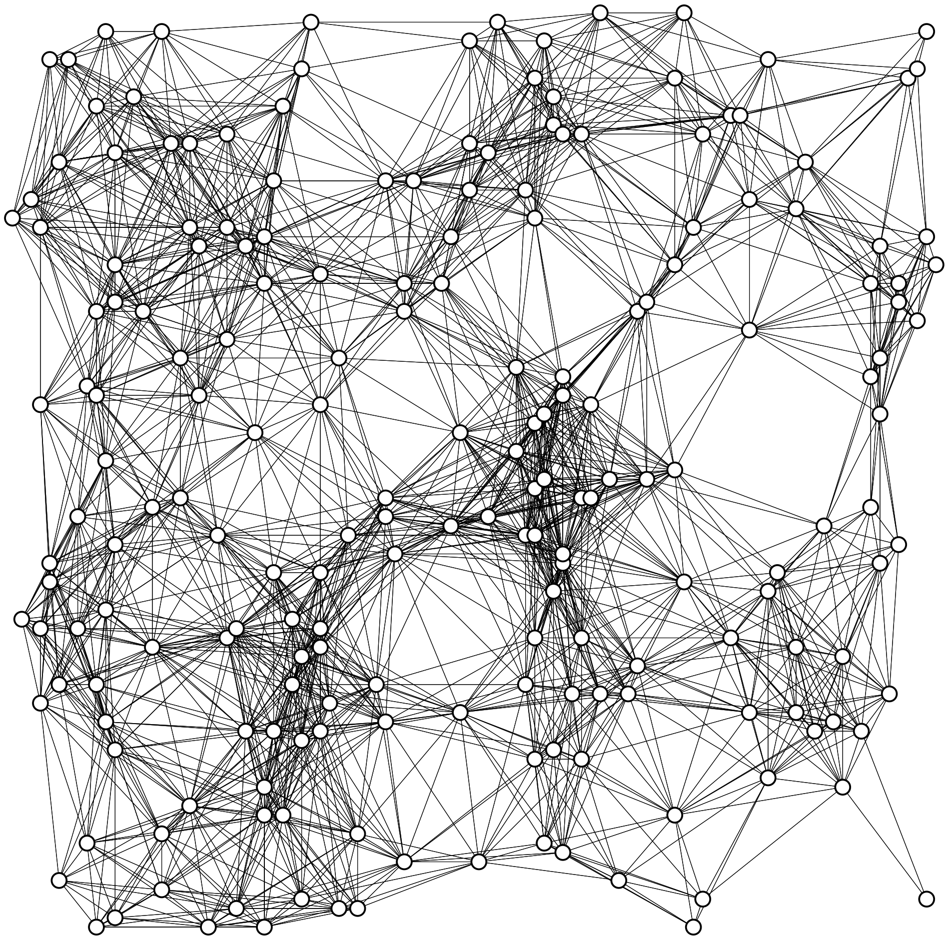

Figure 4.4: An instance $G_G^{200,d}$ of $G_G(200, 100, 100, 20)$.

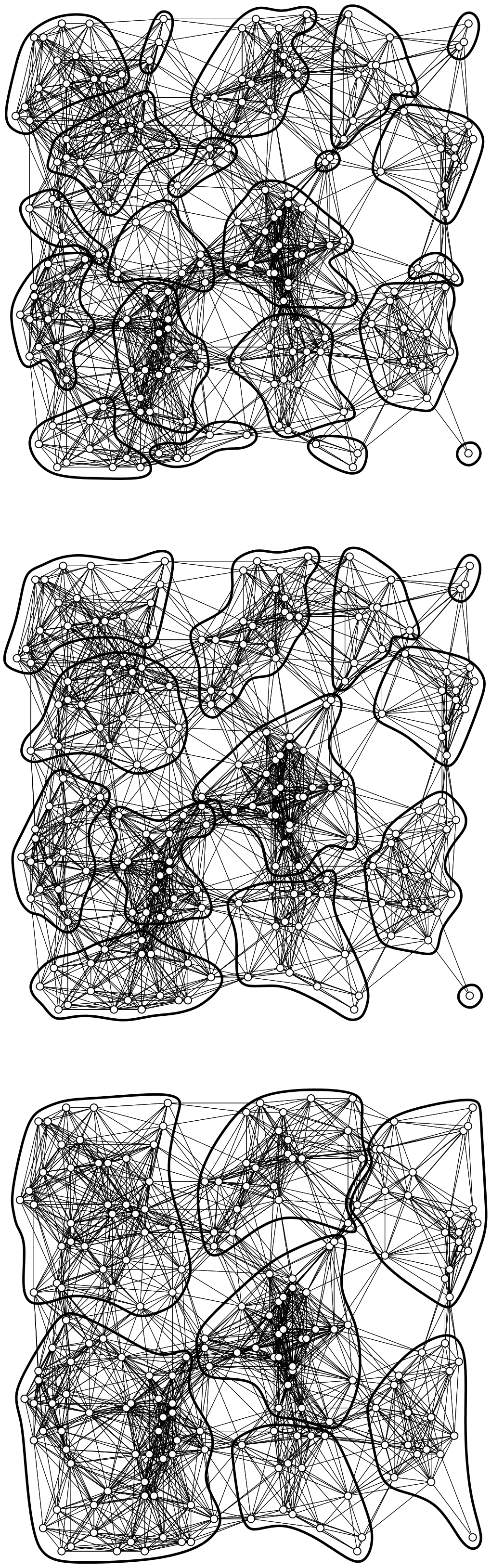

Figure 4.5: $\mathcal{C}_N$ (top), $\mathcal{C}_S$ (middle), and the MCL clustering for $G_G^{200,d}$.

		$\mathcal{F}_D = 10$	$\mathcal{F}_D = 20$	$\mathcal{F}_D = 40$	$\mathcal{F}_D = 60$	$\mathcal{F}_D = 100$
Destructive	Cost	47248 (92)	47281 (103)	47343 (110)	47395 (96)	47479 (128)
	Moves	1037 (329)	992 (359)	821 (310)	776 (297)	**759** (298)
Shuffling	Cost	47510 (90)	47544 (127)	47244 (96)	47429 (101)	47371 (90)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$	Moves	1085 (179)	1068 (199)	1491 (415)	1337 (311)	1418 (359)
Shuffling	Cost	47302 (82)	47270 (72)	47220 (73)	47246 (103)	47430 (89)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/5 \rfloor$	Moves	932 (174)	975 (166)	996 (203)	1100 (252)	967 (303)
Shuffling	Cost	47275 (85)	**47229** (66)	47247 (104)	47244 (81)	47244 (109)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/10 \rfloor$	Moves	790 (169)	899 (208)	866 (216)	835 (246)	877 (264)
No diversification	Cost					47458 (93)
	Moves					603 (140)

Table 4.2: Mean final scaled cost and scaled scheme length for various diversification schemes on an instance $G_E^{100,s}$ of $G_E(500, 10^{-2})$ over twenty runs. Standard deviation is given in brackets.

u in $V(C) \cup \overline{N(v)}$. Thus it is very bad for the naive cost (which will only decrease if v is adjacent to over half of the vertices originally in C). However, β_u also increases dramatically and β_u increases by one for every u in $V(C) \cup \overline{N(v)}$, hence the scaled cost may actually decrease. For this reason, the naive cost function will never have a local minimum containing a cluster of density less than 0.5. In spite of this, we can see that for these cases the very sparse graphs generally have close to the same number of clusters for $\mathcal{C}_N$ and $\mathcal{C}_S$. When the graph is sparse, the neighbours of a given vertex are unlikely to be divided among several clusters.

4.2.2 Diversification

As mentioned in Chapter 3, we opt for destructive diversification instead of shuffling diversification in the scaled scheme, and do not consider diversification at all in the naive scheme. Tables 4.2, 4.3, and 4.4 show some statistics on the minimum cost reached and the number of scaled moves performed by RNSC on an Erdös-Rényi graph, a scale-free

		$\mathcal{F}_D = 10$	$\mathcal{F}_D = 20$	$\mathcal{F}_D = 40$	$\mathcal{F}_D = 60$	$\mathcal{F}_D = 100$
Destructive	Cost	71408 (727)	70964 (126)	70966 (215)	70925 (162)	70901 (120)
	Moves	**1175** (453)	1306 (376)	1323 (446)	1408 (545)	1453 (475)
Shuffling	Cost	70966 (85)	70987 (77)	70879 (81)	70980 (74)	70991 (79)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$	Moves	2016 (512)	1850 (369)	2583 (642)	2582 (589)	2540 (779)
Shuffling	Cost	70907 (79)	70902 (65)	70860 (95)	70840 (82)	70959 (65)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/5 \rfloor$	Moves	1552 (308)	1582 (253)	1838 (448)	2163 (469)	1774 (497)
Shuffling	Cost	70902 (83)	70830 (56)	70845 (81)	**70819** (57)	70889 (105)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/10 \rfloor$	Moves	1356 (366)	1700 (286)	1672 (515)	1782 (482)	1566 (445)
No diversification	Cost					71012 (90)
	Moves					1089 (447)

Table 4.3: Mean final scaled cost and scaled scheme length for various diversification schemes on an instance of $G_S(500, 25)$ over twenty runs. Standard deviation is given in brackets.

		$\mathcal{F}_D = 10$	$\mathcal{F}_D = 20$	$\mathcal{F}_D = 40$	$\mathcal{F}_D = 60$	$\mathcal{F}_D = 100$
Destructive	Cost	34118 (204)	33996 (110)	34134 (189)	34085 (134)	34068 (153)
	Moves	679 (252)	539 (84)	378 (98)	415 (157)	**349** (76)
Shuffling	Cost	34307 (132)	34325 (162)	33986 (98)	34109 (76)	34055 (113)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$	Moves	653 (123)	581 (162)	696 (220)	627 (208)	608 (257)
Shuffling	Cost	34153 (103)	34060 (134)	**33962** (89)	34035 (130)	34050 (116)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/5 \rfloor$	Moves	514 (158)	605 (134)	559 (212)	620 (240)	407 (125)
Shuffling	Cost	34048 (101)	34029 (126)	33988 (78)	34005 (110)	34066 (148)
$\mathcal{L}_D = \lfloor \mathcal{F}_D/10 \rfloor$	Moves	447 (180)	456 (126)	461 (165)	499 (230)	476 (235)
No diversification	Cost					34083 (163)
	Moves					419 (174)

Table 4.4: Mean final scaled cost and scaled scheme length for various diversification schemes on an instance $G_G^{500,d}$ of $G_G(500, 100, 100, 20)$ over twenty runs. Standard deviation is given in brackets.

		$\mathcal{T}_T = 1$	$\mathcal{T}_T = 2$	$\mathcal{T}_T = 3$
$\mathcal{L}_T = 0$	Cost	34384 (201)	–	–
	Moves	201 (11)	–	–
$\mathcal{L}_T = 3$	Cost	34271 (235)	–	–
	Moves	226 (13)	–	–
$\mathcal{L}_T = 5$	Cost	34156 (222)	34279 (234)	34235 (200)
	Moves	238 (22)	213 (15)	224 (27)
$\mathcal{L}_T = 10$	Cost	34165 (166)	34196 (142)	34275 (184)
	Moves	290 (70)	224 (16)	234 (23)
$\mathcal{L}_T = 20$	Cost	34027 (132)	34194 (187)	34097 (127)
	Moves	357 (118)	240 (33)	309 (87)
$\mathcal{L}_T = 50$	Cost	33993 (80)	34138 (154)	34118 (139)
	Moves	397 (105)	323 (60)	329 (120)
$\mathcal{L}_T = 100$	Cost	34122 (88)	34015 (77)	34018 (105)
	Moves	241 (26)	369 (160)	410 (146)

Table 4.5: Mean final scaled cost and scaled scheme length for various tabu parameter values on an instance $G_G^{500,d}$ of $G_G(500, 100, 100, 20)$ over twenty runs. Standard deviation is given in brackets.

graph, and a geometric graph, respectively. We set $\mathcal{T}_n$ to 10, $\mathcal{T}_s$ to 10, and the tabu length to 15 for these experiments.

It is evident that the choice of diversification scheme does not make a huge difference to the performance of the clustering algorithm. It is no surprise that setting $\mathcal{L}_D$ to $\lfloor \mathcal{F}_D/3 \rfloor$ is a costly choice; we spend only twice as much time clustering as diversifying in this case. In general, if the diversification frequency is reasonably large, destructive diversification will result in a faster run than will shuffling diversification. For very large graphs, this can be a significant concern.

4.2.3 Tabu Length

With the tabu length $\mathcal{L}_T$ and tabu tolerance $\mathcal{T}_T$, the dependence of the RNSC algorithm is more evident than with the diversification parameters. Tables 4.6, 4.5, and 4.7 present

		$\mathcal{T}_T = 1$	$\mathcal{T}_T = 2$	$\mathcal{T}_T = 3$
$\mathcal{L}_T = 0$	Cost	47783 (118)	–	–
	Moves	337 (16)	–	–
$\mathcal{L}_T = 3$	Cost	47782 (137)	–	–
	Moves	341 (18)	–	–
$\mathcal{L}_T = 5$	Cost	47653 (136)	47755 (131)	47750 (105)
	Moves	378 (50)	349 (28)	352 (29)
$\mathcal{L}_T = 10$	Cost	47603 (161)	47739 (129)	47748 (124)
	Moves	462 (109)	354 (28)	359 (34)
$\mathcal{L}_T = 20$	Cost	47375 (139)	47662 (100)	47652 (128)
	Moves	736 (179)	391 (28)	477 (91)
$\mathcal{L}_T = 50$	Cost	47192 (100)	47424 (112)	47486 (118)
	Moves	867 (219)	566 (78)	677 (149)
$\mathcal{L}_T = 100$	Cost	47252 (64)	47358 (89)	47307 (91)
	Moves	726 (139)	783 (228)	902 (265)

Table 4.6: Mean final scaled cost and scaled scheme length for various tabu parameter values on an instance $G_E^{500,s}$ of $G_E(500, 10^{-2})$ over twenty runs. Standard deviation is given in brackets.

		$\mathcal{T}_T = 1$	$\mathcal{T}_T = 2$	$\mathcal{T}_T = 3$
$\mathcal{L}_T = 0$	Cost	85968 (153)	–	–
	Moves	255 (9)	–	–
$\mathcal{L}_T = 3$	Cost	85920 (150)	–	–
	Moves	265 (32)	–	–
$\mathcal{L}_T = 5$	Cost	85943 (153)	85941 (142)	85890 (88)
	Moves	258 (8)	259 (10)	258 (7)
$\mathcal{L}_T = 10$	Cost	85949 (120)	85898 (121)	85989 (132)
	Moves	257 (11)	265 (31)	261 (26)
$\mathcal{L}_T = 20$	Cost	85946 (150)	85861 (125)	86008 (196)
	Moves	256 (9)	262 (8)	255 (7)
$\mathcal{L}_T = 50$	Cost	85884 (105)	85848 (111)	86019 (188)
	Moves	285 (42)	294 (39)	263 (13)
$\mathcal{L}_T = 100$	Cost	85888 (89)	85849 (117)	85854 (132)
	Moves	340 (108)	291 (50)	338 (89)

Table 4.7: Mean final scaled cost and scaled scheme length for various tabu parameter values on yeast PPI graph Y_{2k} over twenty runs. Standard deviation is given in brackets.

experiments with varying tabu parameters for a random geometric graph, an Erdös-Rényi random graph, and Y_{2k}, the yeast PPI network on 988 proteins, which shares key properties with both scale-free and geometric graphs [8, 55].

Although adjusting the tabu tolerance does not seem to provide any advantage in general, adjusting the tabu length does have an effect. For clustering, it seems that a tabu tolerance of 1 and a tabu length of $|V|/10$ will give good results, however this comes at a cost of experiment length.

When coupling diversification with tabu length, it is important that the tabu list be long enough to hold the diversification moves in place until the local minimum is escaped. Because of the structural focus in destructive diversification, this is less of a concern than it is with shuffling diversification.

4.2.4 Search Length

Because scaled moves are much more costly than naive moves, we use the naive scheme as a preprocessor. We are therefore interested in knowing how far apart $\mathcal{C}_n$ and $\mathcal{C}_s$ are in the search space. More importantly, since we choose our own parameters for the tabu list, stopping tolerance, and diversification, how many scaled moves do we make before we hit a local minimum?

Figures 4.6, 4.7, 4.8, and 4.9 show results for varying stopping tolerances $\mathcal{T}_n$ and $\mathcal{T}_s$ for $G_G^{200,d}$ (with 200 vertices), an instance $G_G^{500,d}$ of $G_G(500, 100, 100, 20)$, an instance of $G_S(500, 25)$, and the yeast interaction network Y_{2k}. For these runs, a tabu length of $|V|/10$ was used with no diversification.

It is evident and intuitive that in general, a higher stopping tolerance will result in a longer search and a lower cost for $\mathcal{C}_S$. The best cost, however, does not decrease strictly as $\mathcal{T}_s$ increases. With no diversification, much of the performance of the RNSC algorithm rests on random chance. This is, in a way, encouraging, because it suggests that running multiple experiments can lower the scaled cost of $\mathcal{C}_F$. We would prefer to

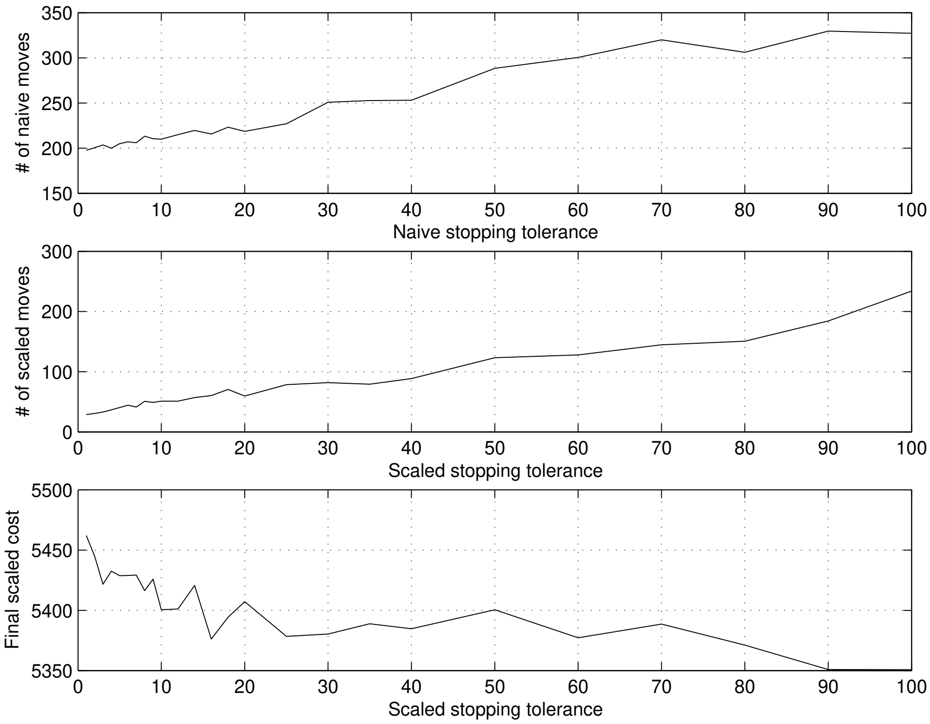

Figure 4.6: Average naive run length, scaled run length, and best scaled cost for 10 runs each on varying stopping tolerance values for $G_G^{200,d}$.

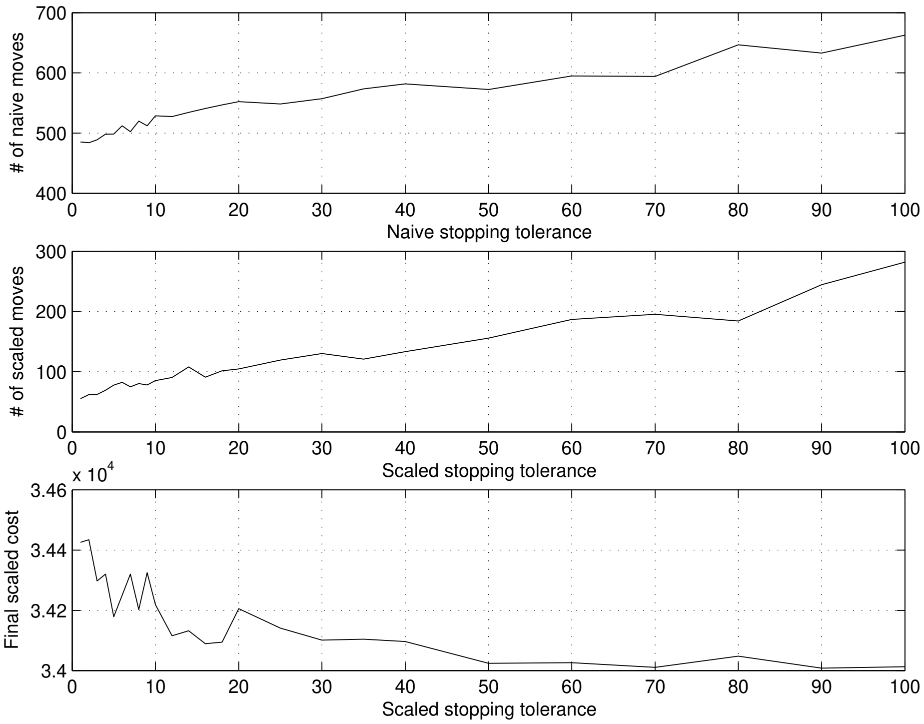

Figure 4.7: Average naive run length, scaled run length, and best scaled cost for 10 runs each on varying stopping tolerance values for $G_G^{500,d}$.

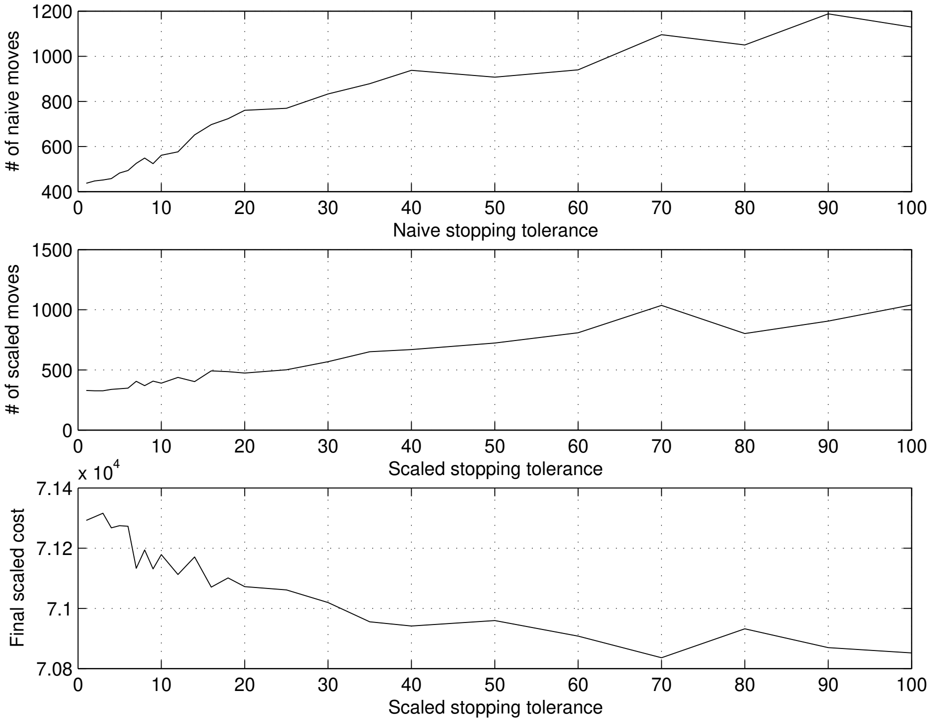

Figure 4.8: Average naive run length, scaled run length, and best scaled cost for 10 runs each on varying stopping tolerance values for an instance of $G_S500, 25$.

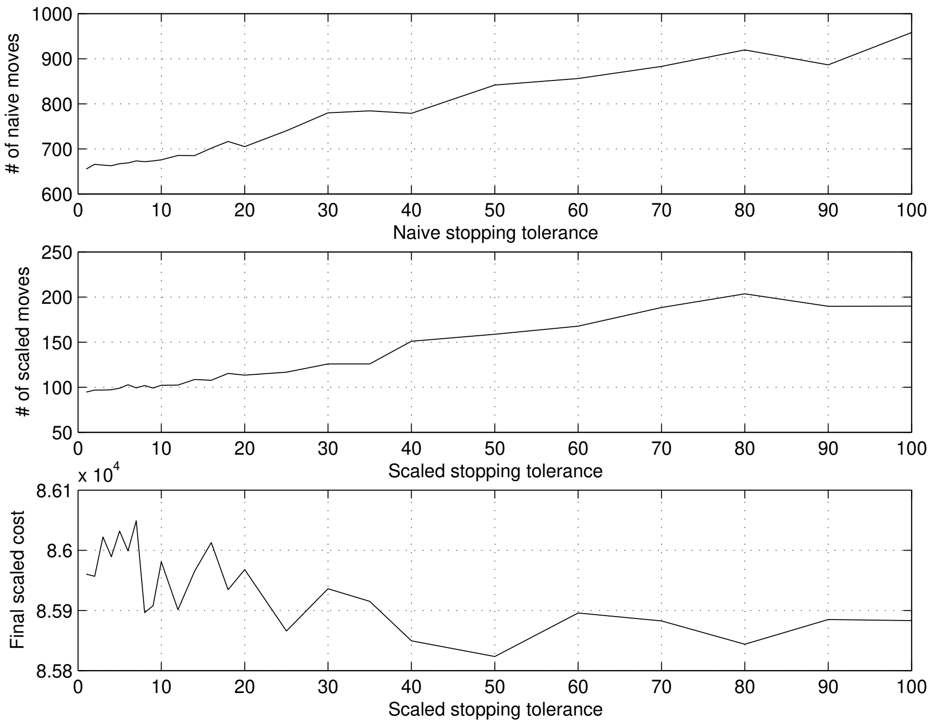

Figure 4.9: Average naive run length, scaled run length, and best scaled cost for 10 runs each on varying stopping tolerance values for Y_{2k}.

have the algorithm perform near-optimally in every experiment, but this is unrealistic for a local search algorithm applied to a problem as complex as cost-based clustering.

We can see that with the exception of the scale-free graph, the number of scaled moves made before a minimum is found (i.e. the length of the scaled scheme with $\mathcal{T}_t = 1$) is more or less proportional to the number of vertices in the graph. In the scale-free graph, we can see from Figure 4.2 that $\mathcal{C}_n$ contains roughly 150 clusters while $\mathcal{C}_s$ contains roughly 50 clusters. Essentially, the entire clustering must be rearranged during the scaled scheme.

Stopping tolerance should be considered in two ways: Without diversification, it represents the depth of local minimum for which we are willing to settle, provided that the tabu list is long enough. With diversification, however, a long stopping tolerance can reflect the rigorousness of the search. A longer scaled stopping tolerance translates to more diversification steps before $\mathcal{C}_s$ is chosen. Either way, it is a tradeoff between computation time constraints and search quality.

4.3 Performance Results

We already know that the scaled scheme of the RNSC algorithm is extremely computationally demanding. How does it perform, though, in general? This section examines some computational performance experiments for clustering and colouring with RNSC. All experiments are run on a P4 2.8GHz processor.

4.3.1 Random Graphs

We tested the RNSC algorithm on several Erdös-Rényi, geometric, and scale-free random graphs. The test bed consisted of three graphs of each type and size: one of density $\approx .01$ (sparse, denoted $G_*^{|V|,s}$), one of density $\approx .03$ (medium density, denoted $G_*^{|V|,m}$), and one of density $\approx .1$ (dense, denoted $G_*^{|V|,d}$). The sizes used were 200 vertices and 500 vertices.

Each run had parameters $\mathcal{T}_n = 10$, $\mathcal{T}_s = 30$, $\mathcal{T}_t = 1$, $\mathcal{L}_t = 25$, and $\mathcal{N}_E = 20$. The

Graph	Time	Time per scaled move	# of clusters	Scaled cost	Scaled cost of MCL output
$G_E^{200,s}$	0.150s	1.76×10^{-3}	109	3741	5400
$G_E^{200,m}$	0.198s	1.09×10^{-3}	61	7931	9225
$G_E^{200,d}$	0.273s	1.08×10^{-3}	29	10491	12404
$G_G^{200,s}$	0.129s	2.40×10^{-3}	90	2132	5143
$G_G^{200,m}$	0.155s	2.04×10^{-3}	47	4158	5527
$G_G^{200,d}$	0.233s	3.01×10^{-3}	14	5341	6523
$G_S^{200,s}$	0.127s	2.43×10^{-3}	123	4631	4745
$G_S^{200,m}$	0.193s	1.12×10^{-3}	76	7760	10139
$G_S^{200,d}$	0.328s	1.54×10^{-3}	33	10296	11900

Table 4.8: Performance results for random graphs on 200 vertices.

Graph	Time	Time per scaled move	# of clusters	Scaled cost	Scaled cost of MCL output
$G_E^{500,s}$	1.464	4.65×10^{-3}	188	46917	54750
$G_E^{500,m}$	1.819	4.28×10^{-3}	112	65724	76474
$G_E^{500,d}$	7.187	8.54×10^{-3}	26	72219	81094
$G_G^{500,s}$	1.040	8.52×10^{-3}	116	25594	34122
$G_G^{500,m}$	1.690	1.01×10^{-2}	47	34580	41918
$G_G^{500,d}$	2.651	2.61×10^{-2}	16	38712	45064
$G_S^{500,s}$	1.272	5.64×10^{-3}	198	45327	62878
$G_S^{500,m}$	1.825	4.02×10^{-3}	134	64517	76013
$G_S^{500,d}$	16.14	3.05×10^{-3}	44	70935	74800

Table 4.9: Performance results for random graphs on 500 vertices.

Graph	Method	Time	Time per scaled move	# of clusters	Scaled cost
Y_{2k}	RNSC	4.03s	.0032s	391	8.56×10^4
	MCL	0.38s	n/a	427	1.44×10^5
Y_{11k}	RNSC	68.3s	.193s	972	7.67×10^5
	MCL	3.42s	n/a	850	1.02×10^6
Y_{45k}	RNSC	916s	.956s	1815	4.07×10^6
	MCL	61.5s	n/a	1199	4.93×10^6
Y_{78k}	RNSC	1071s	.983s	1811	5.26×10^6
	MCL	102s	n/a	1179	6.49×10^6

Table 4.10: Performance results for yeast protein-protein interaction graphs.

sparse, medium, and dense graphs of size 200 used $\mathcal{N}_C = 150$, 100, and 50 respectively. Those of size 500 used $\mathcal{N}_C = 240$, 160, and 80 respectively. The resulting output clusterings have much lower cost than the MCL output in every case. The difference in some cases, particularly geometric graphs, is surprisingly large.

4.3.2 Protein-Protein Interaction Networks

We ran the MCL algorithm on the four protein-protein interaction networks for yeast (see graphs Y_{2k}, Y_{11k}, Y_{45k}, and Y_{78k} in Table 4.1). We then compared the RNSC algorithm with Van Dongen's MCL algorithm, a natural clustering algorithm based on adjacency matrix products [64]. MCL is extremely fast, but as its performance is based in natural network flow, it will generally result in higher-cost clusterings. For all experiments, MCL, which is a deterministic algorithm, was run with the default parameters.

Table 4.10 shows the experimental results for the PPI networks. They were run with $\mathcal{T}_s = 30$, $\mathcal{T}_T = 1$, $\mathcal{L}_T = |V|/20$, and $\mathcal{F}'_D = 50$. Y_{2k} was run with $\mathcal{N}_C = 400$ and $\mathcal{N}_E = 10$. Y_{11k} was run with $\mathcal{N}_C = 1000$ and $\mathcal{N}_E = 10$. Y_{45k} and Y_{78k} were run with $\mathcal{N}_C = 2000$ and $\mathcal{N}_E = 3$. For these graphs it is evident that RNSC is much slower than MCL but

Graph	$\chi(G)$	$\mathcal{N}_C$	Mean cost (StdDev)	Success rate	Average time
le450_25a	25	25	0.04 (0.20)	.96	3.27s
le450_25b	25	25	0 (0)	1.00	1.74s
le450_25c	25	26	6.02 (1.58)	.00	27.6s
		27	0.78 (0.78)	.42	18.3s
le450_25d	25	26	6.28 (1.81)	.00	24.7s
		27	1.02 (0.88)	.32	20.0s
$J(8,4,3)$	6	6	0 (0)	1.00	0.52ms
$J(13,3,3)$	11	12	1.64 (1.09)	.14	4.50s
G_{Triple}	10	11	0.08 (0.27)	.92	0.73s

Table 4.11: Performance results for colouring benchmark graphs.

outperforms it in terms of the scaled cost scheme.

4.3.3 Colouring Benchmark Graphs

As a colouring algorithm, RNSC fails in that it is not specifically designed to overcome the problems encountered in graphs that are hard to colour. Its diversification schemes are designed for general structural shuffling, not overcoming symmetries. In spite of its generality, it performs reasonably well on the easier testbed examples. The algorithm was run on several examples, with $\mathcal{N}_E = 50$, $\mathcal{T}_T = 1$, $\mathcal{L}_T = 3$, $\mathcal{F}_D = 50$, $\mathcal{L}_D = 5$, and $\mathcal{T}_n = 200{,}000$. Table 4.11 summarizes the algorithm's performance.

The table gives experimental results for the best colourings found by RNSC. The Leighton graphs le450_25c and le450_25d have been properly 26-coloured by some recent algorithms, but RNSC was unable to match this result. For le450_25c the best cost found was 3, and for le450_25d the best cost found was 2. The algorithm does run very fast, and it shows the same promise in the restricted neighbourhood search meta-heuristic as applied to colouring that is seen in [4].

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis we described previous approaches to the problem of graph clustering and presented a stochastic cost-based local search algorithm for it. The RNSC algorithm that we presented is specific to clustering simple unweighted graphs according to two cost functions: a naive cost function, which can be implemented very easily but is very limited in its expressiveness, and a scaled cost function, which is computationally intensive but considers more information about the graph to provide a better clustering. Since a clustering that is locally optimal within the search space under the naive cost function is often close to a locally optimal clustering under the scaled cost function, we use the naive cost function to reach a preliminary solution clustering from a random initial clustering. We then improve it by manipulating the clustering under the scaled cost function. The search space of clusterings is traversed by changing the cluster of one vertex at a time.

Since it is a local search optimization algorithm, RNSC does not have a worst-case bound on running time. Making a move in the naive scheme takes $\mathcal{O}(n)$ time and making a move in the scaled scheme takes $\mathcal{O}(n^2)$ time, as a worst case bound. The number of moves made is generally $\mathcal{O}(n)$, making the entire algorithm run in $\mathcal{O}(n^3)$ time. However, this bound is most realistic when considering sparse graphs, i.e. when the number of clusters is very large. Adjusting the approach may vastly improve the speed of RNSC for sparse graphs without causing a significant detriment to the quality of the search (see the next section). The algorithm's running time is aided by a plethora of data structures; RNSC has a memory requirement of $\mathcal{O}(|V|^2)$.

In spite of the fact that the algorithm does not run particularly fast, its advantage is that it, unlike previous clustering algorithms, searches according to the performance criteria, i.e. the cost functions. Other approaches may consider the cost functions after the clustering has been computed by some other natural means (e.g. flow simulation, as in the MCL algorithm [64]). However, these approaches are inherently disadvantaged in terms of the quality, as measured by the performance criteria, of the clusterings they produce.

The performance of the RNSC algorithm is aided by some common local search techniques such as a tabu list, diversification, and multiple experiments. These techniques lengthen the search process but produce better results. The algorithm can also be modified slightly to colour graphs rather than clustering them. This method was implemented with moderate success. Overall, the RNSC produces low-cost clusterings at a high cost of computation time. It shows promise in its ability to extend to weighted graphs and the possibility of modification that would result in a much faster approach for sparse graphs.

5.2 Future Work

There are three main horizons with respect to further research in cost-based graph clustering by restricted neighbourhood search: modifying the algorithm to run on weighted graphs, researching bias in the cost functions, and reducing the neighbourhood of a clustering in order to expedite the process for sparse graphs.

5.2.1 An Extension to Weighted Graphs

Changing the RNSC algorithm so that it works for general weighted graphs is a simple matter of modifying the cost functions. If $w_{u,v}$ is the weight of the edge between vertices u and v, then α_v, the cost numerator for v in a simple unweighted graph, could be altered to reach γ_v, the cost numerator for v in a weighted graph. Define γ_v as

$$\gamma_v = \sum_{u \notin C_v} w_{u,v} + \sum_{u \in C_v} (1 - w_{u,v}). \tag{5.1}$$

With the new cost numerator γ_v defined, we can let the naive cost function be

$$C_n(G, \mathcal{C}) = \frac{1}{2} \sum_{v \in V} \gamma_v \tag{5.2}$$

and we can let the scaled cost function be

$$C_s(G, \mathcal{C}) = \frac{n-1}{3} \sum_{v \in V} \frac{\gamma_v}{\beta_v}. \tag{5.3}$$

This can be implemented in the same way that the algorithm is implemented for unweighted graphs; there would be a slight slowdown because the operations are more complicated. It would be useful to implement the RNSC algorithm for weighted graphs. For example, the application of the RNSC algorithm to protein complex prediction could use the confidence values for the connections in the protein-protein interaction networks as edge weights.

5.2.2 Bias in the Cost Functions

Our current cost numerator α_v is defined as $\#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)$. We are interested to know what might happen if we assign weights w_0 and w_1 and let α_v be $w_1 \cdot \#^1_{\mathrm{out}}(G, \mathcal{C}, v) + w_0 \cdot \#^0_{\mathrm{in}}(G, \mathcal{C}, v)$. In terms of the unweighted cost numerator, cross-edges and absent in-cluster edges are both considered equally bad. The weights can change this.

The use of these weights introduces *bias* in the cost functions, and can be useful in two major ways: First, some applications may want larger clusters that do not need to be extremely dense. Introducing these weights, i.e. setting w_0 to be smaller than w_1, can facilitate this. Second, by changing the definition of α_v for the naive cost only, it may be possible to make $\mathcal{C}_n$ closer to $\mathcal{C}_s$. Doing this for very sparse graphs, for example, might involve ranking all graphs of size ≤ 5 by their scaled and naive cost and attempting to bias the naive cost function to make these rankings similar. Because the graph is sparse, most clusters will be very small, so $\mathcal{C}_n$ will hopefully be very close to $\mathcal{C}_s$ if the naive cost function is biased suitably.

5.2.3 Reducing the Neighbourhood of a Clustering

When a move M is made by the RNSC algorithm, the restricted neighbourhood of the move, i.e. the set of moves for which the change in cost must be updated, consists of any move to or from one of the two clusters involved in M. The restricted neighbourhood of M, denoted $\mathcal{R}(M)$, has size $(|V(C_i)| + |V(C_j)|) \cdot (\mathcal{N}_C - 3) + 2|V|$, where C_i and C_j are the source and destination clusters and $\mathcal{N}_C$ is the maximum number of clusters used by the algorithm.

It stands to reason that the majority of good moves will either involve moving a vertex to the cluster containing one of its neighbours or moving it to an empty cluster. If we consider these as our only possible moves, not only will our move list have size bounded by $2m + n$, where $n = |V|$ and $m = |E|$, but we will have smaller restricted neighbourhoods to update. Let $N_m^{\mathcal{C}}(v)$ be the *meta-neighbourhood of v under $\mathcal{C}$*, defined as the set of clusters (not including C_v) that contain a neighbour of v. Let $d_m^{\mathcal{C}} = |N_m^{\mathcal{C}}(v)|$ be the *meta-degree of v under $\mathcal{C}$*.

If we consider only the moves described above, the restricted neighbourhood of M will be the union of the following sets of moves:

- The movement of any vertex v with $C_i \in N_m^{\mathcal{C}}(v)$ to C_i

- The movement of any vertex v with $C_j \in N_m^{\mathcal{C}}(v)$ to C_j
- The movement of any vertex $v \in C_i$ to a cluster in its meta-neighbourhood
- The movement of any vertex $v \in C_j$ to a cluster in its meta-neighbourhood

Hence we have the loose bound

$$|\mathcal{R}(M)| \leq 2 \sum_{v \in C_i} d(v) + 2 \sum_{v \in C_j} d(v). \tag{5.4}$$

These restricted neighbourhoods will become very small when the clustering is good, as many edges will lie within clusters.

For sparse graphs, where the meta-neighbourhoods of vertices will be very small and $\mathcal{N}_C$ is very large, this method could yield a vast improvement in speed over the current RNSC method. However, it remains to be seen whether or not the quality of the search will be significantly worsened by reducing the algorithm's freedom. Further research is certainly warranted.

Appendix A

Glossary of Notation

Graph Data (for graph G)

V	Vertex set
E	Edge set
n	Order = $\lvert V \rvert$
m	Edge order = $\lvert E \rvert$
$\mathcal{C}[G]$	The set of clusterings on G
$u \sim v$	Vertices u and v are adjacent
$u \not\sim v$	Vertices u and v are not adjacent

Random Graphs

$G_E(n,p)$	Erdös-Rényi random graph on n vertices with edge probability p
$G_S(n,k)$	Scale-free random graph on n vertices with $k(n-k)$ edges
$G_G(n,l,w,d)$	Geometric random graph on n vertices on an $l \times w$ grid with maximum edge length d

Clustering and Move Data (for clustering $\mathcal{C}$ on G)

C_i	Cluster in $\mathcal{C}$ with label i for $1 \leq i \leq \mathcal{N}_C$
C_v	Cluster containing v for $v \in V$
$C_n(G, \mathcal{C})$	Naive cost of $\mathcal{C}$
$C_s(G, \mathcal{C})$	Scaled cost of $\mathcal{C}$
$C_c(G, \mathcal{C})$	Colouring cost of $\mathcal{C}$
$\#^0_{\text{in}}(G, \mathcal{C}, v)$	Number of good edges missing from v
$\#^1_{\text{out}}(G, \mathcal{C}, v)$	Number of bad edges incident to v
$M[\mathcal{C}]$	Set of moves possible from $\mathcal{C}$
$N(\mathcal{C})$	Open neighbourhood of $\mathcal{C}$ in the search space
$\mathcal{R}(M)$	Restricted neighbourhood of a move M
$\delta_n(M)$	Change in naive cost due to M
$\delta_s(M)$	Change in scaled cost due to M
$\delta_c(M)$	Change in colouring cost due to M
α_v	Cost numerator for a vertex v
β_v	Cost denominator for a vertex v

RNSC Parameters

$\mathcal{T}_n$	Naive stopping tolerance
$\mathcal{T}_s$	Scaled stopping tolerance
$\mathcal{N}_E$	Number of experiements
$\mathcal{N}_C$	Number of clusters
$\mathcal{L}_T$	Tabu length
$\mathcal{T}_T$	Tabu tolerance
$\mathcal{F}_D$	Shuffling diversification frequency
$\mathcal{L}_D$	Shuffling diversification length
$\mathcal{F}'_D$	Destructive diversification frequency

Bibliography

[1] S. Aksoy and R. M. Haralick. Graph-theoretic clustering for image grouping and retrieval. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, volume 1*, pages 63–68, Colorado, June 1999.

[2] Md. Altaf-Ul-Amin, K. Nishikata, T. Koma, T. Miyasato, Y. Shinbo, Md. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori, and S. Kanaya. Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences. *Genome Informatics*, 14:498–499, 2003.

[3] A. Amir and M. Lindenbaum. A generic grouping algorithm and its quantitative analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(2):168–185, 1998.

[4] C. Avanthay, A. Hertz, and N. Zufferey. A variable neighbourhood search for graph coloring. *European Journal of Operational Research*, 151:379–388, 2003.

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

[6] S. Bandyopadhyay, U. Maulik, and M. K. Pakhira. Clustering using simulated annealing with probabilistic redistribution. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 15(2):269–285, 2001.

[7] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[8] A.-L. Barabási and Z. N. Oltvai. Network biology: Understanding the cell's functional organization. *Nature Reviews Genetics*, 5:101–113, 2004.

[9] E. Boros and P. L. Hammer. On clustering problems with connected optima in euclidean spaces. *Discrete Mathematics*, 75:81–88, 1989.

[10] U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. In *Proc. 11th Europ. Symp. Algorithms (ESA '03), Lecture Notes in Computer Science*, volume 2832, pages 568–579. Springer-Verlag, 2003.

[11] C. Cooper and A. Frieze. A general model of web graphs. In *Proceedings of ESA 2001*, pages 500–511, 2001.

[12] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, October 1996.

[13] W. Fernandez de la Vega, M. Karpinski, C. Kenyon, and Y. Rabani. Approximation schemes for clustering problems. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 50–58, 2003.

[14] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, Upper Saddle River, NJ, 1999.

[15] Hugo A D do Nascimento and Peter Eades. A system for graph clustering based on user hints. In Peter Eades and Jesse Jin, editors, *Selected papers from Pan-Sydney Workshop on Visual Information Processing*, Sydney, Australia, 2001. ACS.

[16] Ulrich Elsner. Graph partitioning – a survey. Technical Report 393, Technische Universitat Chemnitz, 1997.

[17] A. J. Enright and C. A. Ouzounis. Biolayout – an automatic graph layout algorithm for similarity visualization. *Bioinformatics*, 17(9):853–854, 2001.

[18] A. J. Enright, S. M. van Dongen, and C. A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.

[19] P. Erdös and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci.*, 5:17–61, 1960.

[20] Julie Falkner, Franz Rendl, and Henry Wolkowitz. A computational study of graph partitioning. *Mathematical Programming*, 66(2):211–239, 1994.

[21] O. D. Faugeras, editor. *Fundamentals in Computer Vision – An Advanced Course.* Cambridge University Press, Cambridge, 1983.

[22] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, 1962.

[23] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991.

[24] P. Gajer, M. Goodrich, and S. Kobourov. A fast multi-dimensional algorithm for drawing large graphs, 2000. Submitted to GD 2000.

[25] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., New York, 1979.

[26] L. Giot, J. S. Bader, C. Brouwer, A. Chaudhuri, B. Kuang, Y. Li, Y. L. Hao, C. E. Ooi, B. Godwin, E. Vitols, G. Vijayadamodar, P. Pochart, H. Machineni, M. Welsh, Y. Kong, B. Zerhusen, R. Malcolm, Z. Varrone, A. Collis, M. Minto, S. Burgess, L. McDaniel, E. Stimpson, F. Spriggs, J. Williams, K. Neurath, N. Ioime, M. Agee, E. Voss, K. Furtak, R. Renzulli, N. Aanensen, S. Carrolla, E. Bickelhaupt,

Y. Lazovatsky, A. DaSilva, J. Zhong, C. A. Stanyon, R. L. Finley, Jr., K. P. White, M. Braverman, T. Jarvie, S. Gold, M. Leach, J. Knight, R. A. Shimkets, M. P. McKenna, J. Chant, and J. M. Rothberg. A protein interaction map of Drosophila melanogaster. *Science*, 302(5651):1727–1736, 2003.

[27] F. Glover. Tabu search, part I. *ORSA Journal on Computing*, 1(3):190–206, Summer 1989. "ORSA" is called Informs today.

[28] F. Glover. Tabu search, part II. *ORSA Journal on Computing*, 2(1):4–32, Winter 1990. "ORSA" is called Informs today.

[29] F. Glover, C. McMillan, and B. Novick. Interactive decision software and computer graphics for architectural and space planning. *Annals of Operations Research*, 5:557–573, 1985.

[30] D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, Reading, Mass., 1989.

[31] I. J. Good. The botryology of botryology. In J. van Ryzin, editor, *Classification and Clustering*, pages 73–94. Academic Press, New York, 1977.

[32] R. M. Haralick. Image segmentation survey. In O. D. Faugeras, editor, *Fundamentals in Computer Vision – An Advanced Course*, pages 209–224. Cambridge University Press, Cambridge, 1983.

[33] E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4–6):175–181, 2000.

[34] A. Hertz and D. de Werra. Using tabu search techniques for graph colouring. *Computing*, 39(4), 1987.

[35] H. H. Hoos. *Stochastic Local Search – Methods, Models, Applications*. PhD thesis, Darmstadt University of Technology, 1998.

[36] H. H. Hoos and T. Stützle. Systematic vs. local search for SAT. In Wolfram Burgard, Thomas Christaller, and Armin B. Cremers, editors, *KI-99: Advances in Artificial Intelligence, 23rd Annual German Conference on Artificial Intelligence, Bonn, Germany*, pages 298–293, September 1999.

[37] X. Hu and J. Han. Discovering clusters from large scale-free network graph. In *ACM SIG KDD Second Workshop on Fractals, Power Laws and Other Next Generation Data Mining Tools*, August 2003.

[38] D. Johnson and M. Trick, editors. *Cliques, Coloring and Satisfiability – Second DIMACS Implementation Challenge, October 11-13, 1993.* American Mathematical Society, Providence, RI, 1996.

[39] R. Kannan, S. Vampala, and A. Vetta. On clusterings – good, bad and spectral. In *41st Annual Symposium on Foundations of Computer Science*, November 2000.

[40] B. W. Kernighan and S. Lin. An efficient heuristic procedure for paritioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[41] A. D. King, N. Pržulj, and I. Jurisica. Protein complex prediction via cost-based clustering. Accepted to *Bioinformatics* May 14, 2004.

[42] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[43] F. T. Leighton. A graph colouring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84:489–505, 1979.

[44] S. Li, C. M. Armstrong, N. Bertin, H. Ge, S. Milstein, M. Boxem, P.-O. Vidalain, J.-D. J. Han, A. Chesneau, T. Hao, D. S. Goldberg, N. Li, M. Martinez, J.-F. Rual, P. Lamesch, L. Xu, M. Tewari, S. L. Wong, L. V. Zhang, G. F. Berriz, L. Jacotot, P. Vaglio, J. Reboul, T. Hirozane-Kishikawa, Q. Li, H. W. Gabel, A. Elewa,

B. Baumgartner, D. J. Rose, H. Yu, S. Bosak, R. Sequerra, A. Fraser, S. E. Mango, W. M. Saxton, S. Strome, S. van den Heuvel, F. Piano, J. Vandenhaute, C. Sardet, M. Gerstein, L. Doucette-Stamm, K. C. Gunsalus, J. W. Harper, M. E. Cusick, F. P. Roth, D. E. Hill, and M. Vidal. A map of the interactome network of the metazoan C. elegans. *Science*, 303(5657):540–543, 2004.

[45] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98)*, Piscataway, NY, 1998. IEEE Press.

[46] D. W. Matula. Graph theoretic techniques for cluster analysis algorithms. In J. van Ryzin, editor, *Classification and Clustering*, pages 95–129. Academic Press, New York, 1977.

[47] H. W. Mewes, D. Frishman, U. Guldener, G. Mannhaupt, K. Mayer, M. Mokrejs, B. Morgenstern, M. Munsterkotter, S. Rudd, and B. Weil. Mips: a database for genomes and protein sequences. *Nucleic Acids Research*, 30(1):31–34, 2002.

[48] I. F. Mihaila. Design coloring algorithms. Master's thesis, University of Toronto, 1995.

[49] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303:1538–1542, 2004.

[50] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298:824–827, 2002.

[51] P. B. Mirchandani. The p-median problem and generalizations. In P. B. Mirchandani and R. L. Francis, editors, *Discrete Location Theory*, pages 55–117. Wiley-Interscience, New York, 1990.

[52] P. B. Mirchandani and R. L. Francis, editors. *Discrete Location Theory*. Wiley-Interscience, New York, 1990.

[53] N. Mladenović and P. Hansen. Variable neighbourhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

[54] N. Pržulj. Graph theory approaches to protein interaction data analysis. In I. Jurisica and D. Wigle, editors, *Knowledge Discovery in High-Throughput Biological Domains*. Interpharm/CRC, 2004.

[55] N. Pržulj, D. G. Corneil, and I. Jurisica. Modeling interactome: Scale-free or geometric? Technical Report 321/04, Department of Computer Science, University of Toronto, 2004.

[56] N. Pržulj, D. Wigle, and I. Jurisica. Functional topology in a network of protein interactions. *Bioinformatics*, 20(3):340–348, 2004.

[57] T. Roxborough and A. Sen. Graph clustering using multiway ratio cut. In G. Di Battista, editor, *Proc. 5th Int. Symp. Graph Drawing, volume 1353 of Lect. Notes in Comput. Sci.*, pages 291–296. Springer-Verlag, New York/Berlin, 1997.

[58] F. Ruskey. Combinatorial generation. Unpublished manuscript. Working Version (1j), 2001.

[59] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5:269–287, 1983.

[60] S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature Genetics*, 31:64–67, 2002.

[61] N. J. A. Sloane. Unsolved problems in graph theory arising from the study of codes. *Graph Theory Notes of New York*, 18:11–20, 1989.

[62] M. A. Trick. Second dimacs challenge test problems. In D. Johnson and M. Trick, editors, *Cliques, Coloring and Satisfiability – Second DIMACS Implementation Challenge, October 11-13, 1993.* American Mathematical Society, Providence, RI, 1996.

[63] S. M. van Dongen. A cluster algorithm for graphs. Technical Report INS-R0010, Centrum voor Wiskunde en Informatica, May 2000.

[64] S. M. van Dongen. *Graph Clustering by Flow Simulation.* PhD thesis, University of Utrecht, May 2002.

[65] J. Van Ryzin, editor. *Classification and Clustering. Proceedings of an Advanced Seminar Conducted by the Mathematics Research Center, The University of Wisconsin at Madison, May 3–5, 1976.* Academic Press, New York, 1977.

[66] C. von Mering, R. Krause, B. Snel, M. Cornell, S. G. Oliver, S. Fields, and P. Bork. Comparative assessment of large-scale data sets of protein-protein interactions. *Nature*, 417(6887):399–403, 2002.

[67] Y. C. Wei and C. K. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer Aided Design*, 10(7):911–921, 1991.

[68] Douglas B. West. *Introduction to Graph Theory, Second Edition.* Prentice Hall, Upper Saddle River, NJ, 2001.

[69] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993.