An Efficient Cost-Based Graph Clustering Algorithm

Andrew King
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada

February 7, 2005

Abstract

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.

We present the Restricted Neighbourhood Search Clustering Algorithm, a powerful and efficient stochastic algorithm for clustering graphs. The algorithm uses principles of local search algorithms including restricted (variable) neighbourhood search, tabu search, and diversification schemes to ensure good performance and speed. This report includes a detailed description of the algorithm and experimental results.

Contents

Chapter 1

Introduction

This paper is concerned with describing the Restricted Neighbourhood Search Clustering (RNSC) Algorithm, a clustering algorithm that uses local search methods to iteratively improve a clustering on a graph in an efficient way.

The subject of cluster analysis is concerned with the organization of networks into natural subnetworks. In graph theory, this generally means that we are looking to group the vertices of a graph into highly intra-connected subsets.

Here we present an efficient graph clustering algorithm based on a specific case of the graph clustering problem: We wish to *partition* the vertex set into subsets that are both densely intra-connected and sparsely inter-connected. That is, vertices are very likely to be connected by an edge if they are in the same partition, but they are not likely to be conencted if they are in different partitions.

For the purposes of this report, we will omit a lengthy motivation for the problem. Graph clustering has a multitude of applications in many different areas, and being able to cluster graphs using little time and space is a very important problem. Motivation can be found in King's thesis [7], which also describes a prototype of RNSC.

The Restricted Neighbourhood Search Clustering Algorithm, as you might guess, uses the restricted neighbourhood search meta-heuristic, sometimes known as variable neighbourhood search – see the seminal paper by Mladenović and Hansen [11]. Consider a local search space. In our case, the elements of the search space are clusterings of the input graph (equivalently, partitionings of the vertex set), and two clusterings are adjacent in the search space if one can be reached from the other by moving a single vertex from one cluster to another. A straightforward local search algorithm will consider all possible neighbours of the current clustering, but this is often fruitless and inefficient. A restricted neighbourhood search algorithm, in constrast, will impose a restriction on the neighbourhood and will consider a much smaller neighbourhood for the next iteration.

It turns out that using this strategy with sensible performance criteria leads to a very effective graph clustering algorithm, which we will proceed to describe.

1.1 Definitions and Preliminaries

To judge the quality of a graph clustering algorithm in a reasonable way, we require one or more quantitative *performance criteria* – cost functions that we will apply to a clustering of a graph. We use two performance criteria that were first proposed by van Dongen and used in King's thesis [16, 18, 7]. The first is the *naive cost function*, which maps the set of clusterings of a graph into the non-negative integers. The second is the *scaled cost function*, which maps the set of clusterings of a graph into the non-negative rationals.

Before we define the cost functions, we must outline some terms. Consider a clustering $\mathcal{C}$ of $G = (V, E)$. For $\mathcal{C}$ there is an underlying partitioning of the vertex set V, and we can consider $\mathcal{C}$ to be the set of subgraphs of G induced on the partitions of V. Hence each cluster C has an edge set, a vertex set, a density value, etc. We denote the cluster containing a vertex v by C_v.

Given a clustering $\mathcal{C}$ on a graph G, a *cross-edge* is an edge with its endpoints in different clusters. We denote the number of cross-edges incident to a vertex v by $\#^1_{\text{out}}(G, \mathcal{C}, v)$. Conversely, we denote the number of vertices in C_v that are not adjacent to v by $\#^0_{\text{in}}(G, \mathcal{C}, v)$. We can now define our performance criteria.

The naive cost function for a clustering $\mathcal{C}$ of a graph G, denoted $C_n(G, \mathcal{C})$, is defined as

$$C_n(G, \mathcal{C}) = \frac{1}{2} \sum_{v \in V} \left(\#^1_{\text{out}}(G, \mathcal{C}, v) + \#^0_{\text{in}}(G, \mathcal{C}, v) \right). \tag{1.1}$$

This function is naive in the sense that it does not consider the relative importance of a vertex in a graph. For example, if a vertex is in a very large cluster, removing it will make relatively little qualitative difference to the clustering as compared to the situation in which the vertex is in a small cluster. To reflect this, we prefer to use the scaled cost function $C_s(G, \mathcal{C})$, defined as

$$C_s(G, \mathcal{C}) = \frac{n-1}{3} \sum_{v \in V} \frac{1}{|N(v) \cup C_v|} \left(\#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)\right). \tag{1.2}$$

Both of these cost functions look for both high intra-connectedness and low inter-connectedness among the clusters, but the scaled cost function better expresses what we are looking for. Unfortunately, it is much more time consuming for us to maintain our data structures when we are using the scaled cost function. This is discussed in Section 2.5.

RNSC benefits from using both cost functions: the naive cost function can be used to do fast preprocessing, finding a fairly good clustering very quickly. From this clustering, we can switch over to the scaled cost function and find a better clustering, making few iterations in the algorithm.

One thing should be noted for the purposes of understanding the data structures used in the algorithm: If we move one vertex from one cluster to another cluster (or form a singleton cluster with it), we will not change the naive or scaled cost function by more than $n-1$, where n is the number of vertices in the underlying graph. Furthermore, this bound of $n-1$ is achievable in the naive cost function and asymptotically achievable in the scaled cost function. This is proven in Section 2.1.2 of King's thesis [7], in which the issue of performance criteria is discussed in more depth.

Chapter 2

The RNSC Algorithm

In this chapter we will describe the details of the RNSC algorithm, including concerns regarding data storage and input parameters.

2.1 Overview of the RNSC Algorithm

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 $\{\mathcal{C}_s^{(i)} : 1 \leq i \leq \mathcal{N}_E\}$, 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.

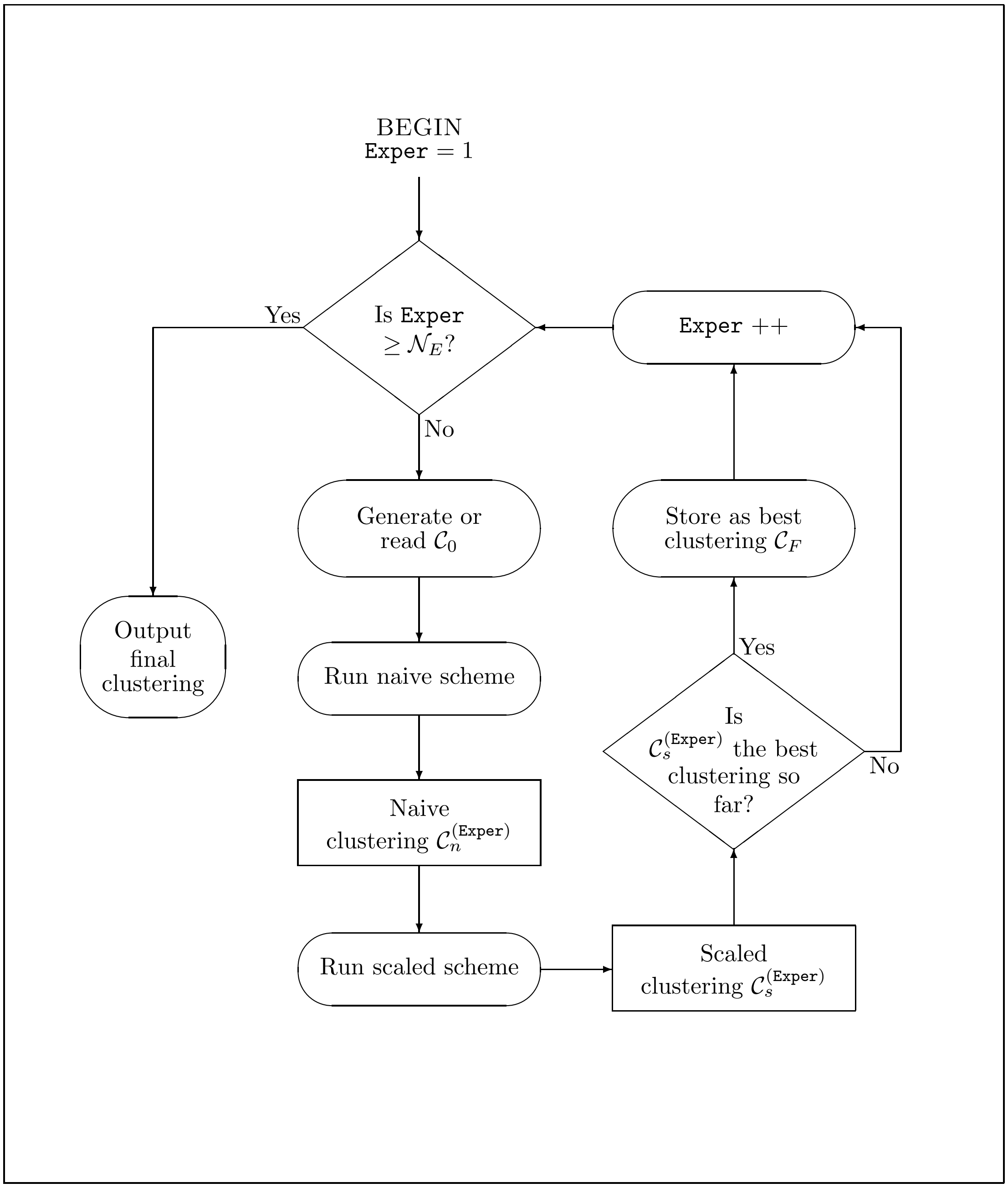

Figure 2.1: The RNSC algorithm.

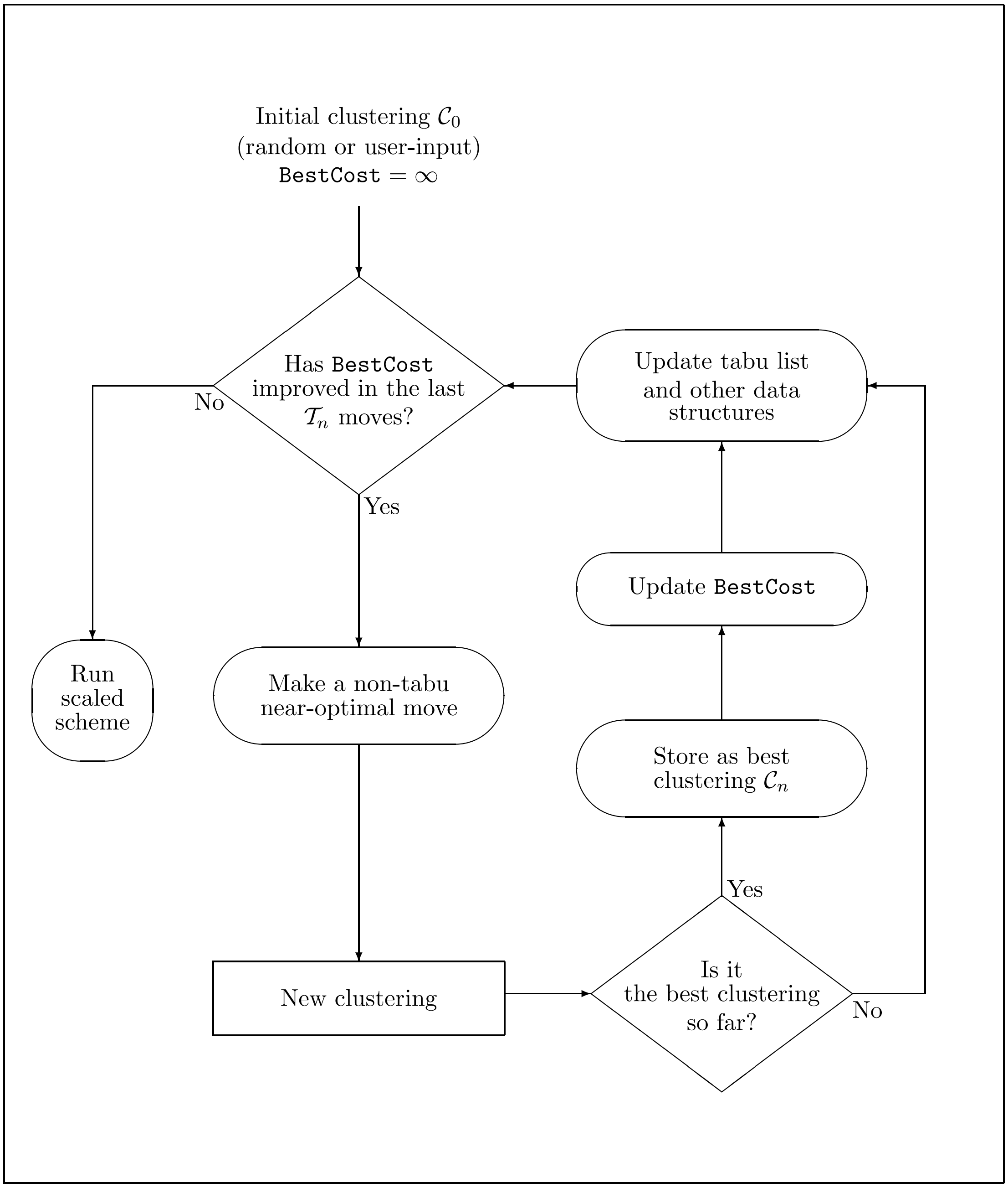

Figure 2.2: The RNSC naive cost scheme.

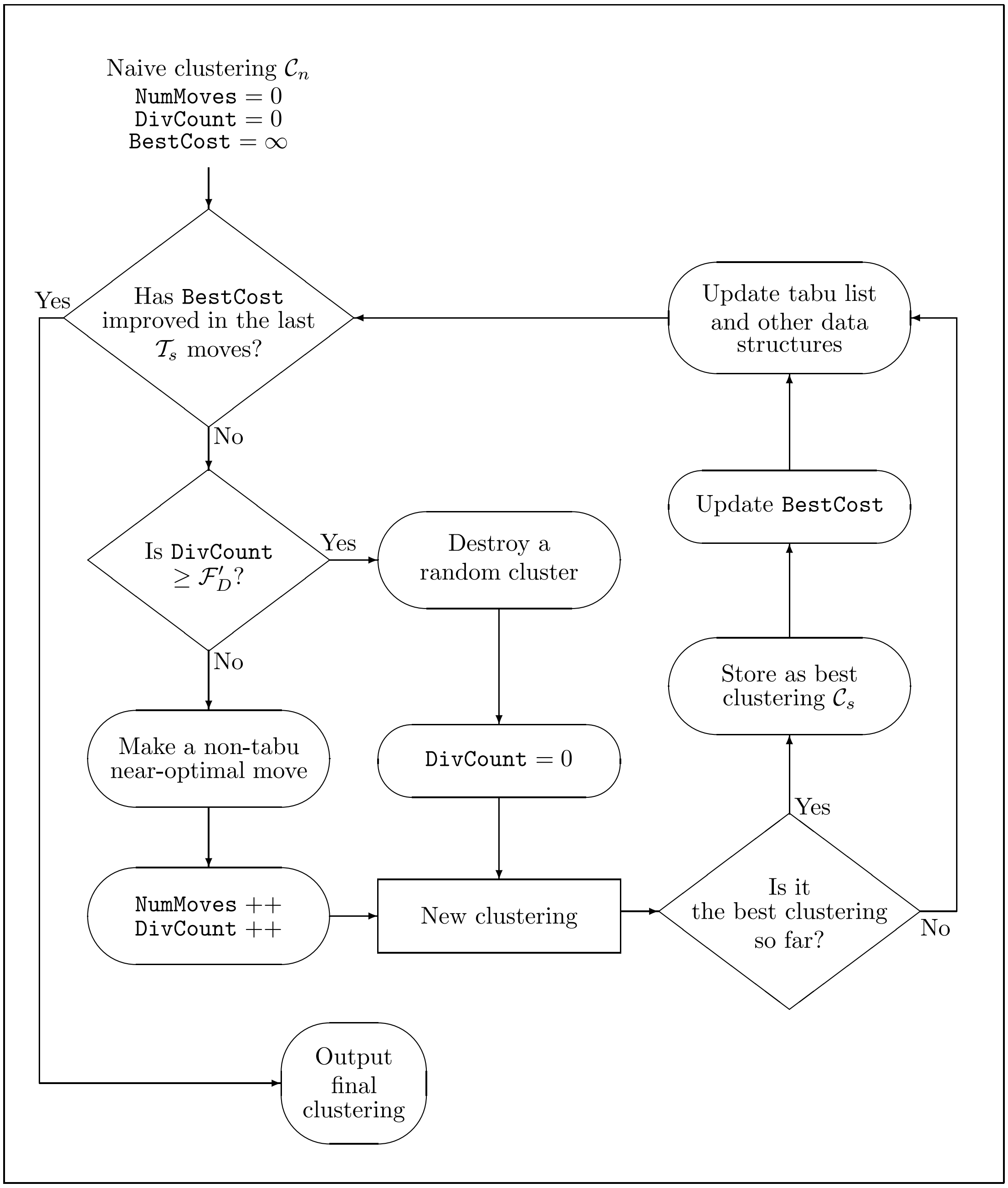

Figure 2.3: The RNSC scaled cost scheme.

Step 2 in this list is the *naive cost scheme*, in which only the integer naive cost function given in Equation 1.1 is considered. Likewise, Step 3 is the *scaled cost scheme*, which uses the scaled cost function for clusterings given in Equation 1.2. Figures 2.1, 2.2, and 2.3 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 2.4).

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 3 presents experimental results regarding the performance of the algorithm.

2.2 Move Types

In the RNSC algorithm, each move involves moving a single vertex from one cluster to another, possibly emptying a cluster or creating a singleton cluster in the process. However, we do not allow all possible moves. We only allow a vertex to be moved into an empty cluster or a cluster that contains one of its neighbours. It does not serve our interests to move a vertex to a cluster that contains none of its neighbours; such a move *never* does as well, in terms of either the naive cost function or the scaled cost function, as moving the vertex to an empty cluster. This is easy to see, recalling Equations 1.1 and 1.2. We can give a very rough upper bound for the number of possible moves, given a clustering of a graph: it is easy to see that there must be fewer than $2m + n$ possible moves, where m and n are the number of edges and the number of vertices, respectively.

Given a clustering $\mathcal{C}$, the *restricted neighbourhood* of $\mathcal{C}$ is the set of clusterings that can be reached from $\mathcal{C}$ by making one permissible move. We denote it by $\mathcal{R}(\mathcal{C})$.

We consider several kinds of moves in the RNSC algorithm. A *standard 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 in hopes of escaping a high local minimum in the search space. Diversification can be approached in two ways. Further, any move may or may not be *tabu* (forbidden). Maintaining a list of tabu moves (a *tabu list*) helps to prevent cycling about a local minimum.

First, some terminology: The set of possible clusterings on a graph G is denoted by $\mathcal{C}[G]$. For a given clustering $\mathcal{C}$, the set of available moves from $\mathcal{C}$ is denoted by $M[\mathcal{C}]$. This is almost equivalent to the clustering's restricted neighbourhood.

2.2.1 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 mentioned in Section 1.1, the change in the cost of a clustering resulting from a single move lies in $[1-n, n-1]$ for the naive cost function and in $(1-n, n-1)$ for the scaled cost function. In the naive scheme, "near-optimal cost" is equivalent to "optimal cost". 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. When the clustering is good, there tend to be many near-optimal moves, so there is a large degree of randomness within this choice.

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.

2.2.2 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 (see [9]), 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 moves followed by $\mathcal{L}_D$ diversification moves. We call this scheme *shuffling diversification*.

The second scheme chooses a fixed number of clusters and demolishes them, i.e. each vertex in these clusters is moved to a random cluster (via a permissible move). We call this diversification scheme *destructive diversification*.

When any vertex may be moved to any cluster, shuffling diversification is not as effective as destrictive diversification [7]. When the move list is restricted, however, the opposite is true. This is likely because when a cluster is destroyed in destructive diversification, all vertices in the cluster have a high likelihood of being moved to the same cluster. This is supported by empirical evidence in Chapter 3.

2.2.3 Forbidden (Tabu) Moves

The general notion of tabu search as proposed by Glover involves, as applied to the clustering problem, a forbidden set of clusterings defined as an explicit set [4, 5]. 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*.

Instead, then, we make vertices tabu and forbid any movement of a vertex in the tabu list. 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. If the list is too short, then cycling will not be prevented. If the list is too long, the algorithm will suffer from undue restriction. The issue of tabu length must be handled with particular care and in particular with consideration of the diversification parameters. This is explored in Section 2.3.4 as well as in the experimental runs.

2.3 Parameters and Options

The RNSC method uses several input parameters that can be tailored to suit the instance of the problem at hand. We must consider search length, diversification parameters, and tabu parameters, all of which can greatly affect the performance of the algorithm.

2.3.1 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 practically impossible to tell what an optimal cost is for a large graph. 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

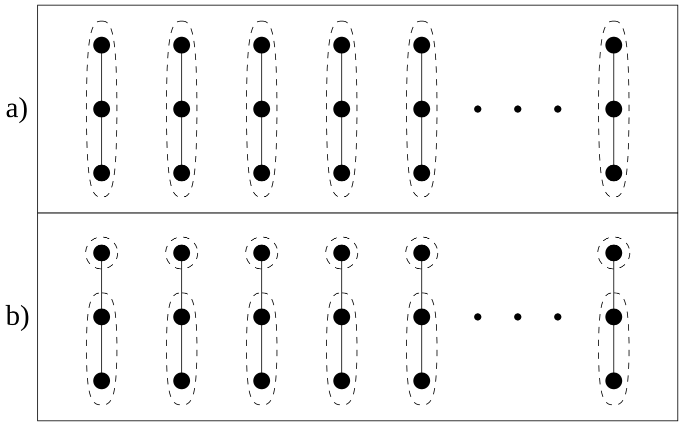

Figure 2.4: In this graph on $3i$ vertices, both clusterings are optimal in the naive scheme, with cost i. However, only the clustering in (a) is optimal in the scaled scheme. It has scaled cost $\frac{4}{6}(i - \frac{1}{3})$, whereas the clustering in (b) has scaled cost $\frac{5}{6}(i - \frac{1}{3})$. Note that the clusterings have no cluster in common.

speed and thoroughness. In practice, our search length will be dictated by the frequency of improvement of the best solution.

For the naive scheme, making a move is relatively fast, but near a local minimum it is 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 2.4. The clustering in Figure 2.4b is an optimal naive clustering (as is the clustering in Figure 2.4a), 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 2.4a. 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, e.g. the protein-protein interaction networks discussed by King *et al.* [8]. Such structure is also described by Milo *et al.* [10] and Shen-Orr *et al.* [15], 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 3. In practice, increasing the experiment length beyond a certain point makes little difference to the scaled cost of the final clustering.

2.3.2 Cluster Limits

A clustering can clearly have no more than n nonempty clusters. However, there are instances in which we may wish to impose an upper bound on the number of clusters allowed. The RNSC algorithm may take a parameter $\mathcal{N}_C$ representing the maximum number of nonempty clusters allowed.

2.3.3 Diversification

Recall from Section 2.2.2 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 is largely affected by two values: the number of clusters and the size of the graph.

Section 3.2.1 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.

2.3.4 The Tabu List

We consider two parameters for the tabu list: the *tabu length* $\mathcal{L}_T$ and the *tabu tolerance* $\mathcal{T}_T$. Every time a move is made, the vertex being moved is put into the tabu list. If the vertex is in the list $\mathcal{T}_T$ times, it can no longer be moved until an instance of it is removed from the list.

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 performance of the RNSC algorithm can be very sensitive to changes in $\mathcal{L}_T$. Increasing the tabu tolerance reduces this sensitivity and increase the depth of memory represented by the tabu list. 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 values.

2.4 Data Structures

The RNSC algorithm depends on several data structures for the graph and the local search space that can communicate with one another efficiently. In this section, we describe the data structures used by RNSC and how they are used.

2.4.1 Graph Data Structures

The data structures relating to the graph G itself are straightforward. The vertices are represented as the integers $0, 1, \ldots, n-1$ (where $n = |V|$), and the graph is stored in adjacency list format (`AdjList`). Since RNSC is most useful for large, sparse graphs, using an adjacency matrix is prohibitively expensive in terms of memory requirements. RNSC also keeps a record of the degree of each vertex, held in `Degree`. These graph data structures are listed in Table 2.1.

In order to use the adjacency list more efficiently, we keep it sorted so that given a vertex v, the neighbours of v which are not in C_v come first in the list, followed by those neighbours of v which are in C_v. For each vertex, we maintain a pointer to the first in-cluster neighbour of v. We store these in `AdjHead2`.

When a vertex v is moved, we must make many operations based on whether or not certain vertices are adjacent to v. To this end, we maintain a single row of the adjacency matrix corresponding to the vertex being moved. It is stored in `AdjVector`. `CostNumerator` and `CostDenominator` are vectors as shown in the table, and their use

Structure	Size	Description
`AdjList`	n lists	Neighbourhood lists for each vertex
`Degree`	n	The degree of every vertex
`AdjHead2`	n	A pointer to the first in-cluster neighbour of each vertex
`AdjVector`	n	Whether or not each vertex is adjacent to the vertex being moved
`CostNumerator`	n	$\alpha_v = \#^1_{\mathrm{out}}(G, \mathcal{C}, v) + \#^0_{\mathrm{in}}(G, \mathcal{C}, v)$ for each vertex
`CostDenominator`	n	$\beta_v = \lvert N(v) \cup C_v \rvert$ for each vertex

Table 2.1: Graph data structures.

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

Table 2.2: Data structures related to the current clustering $\mathcal{C}$.

is described in Section 2.5; they are used to maintain costs.

2.4.2 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 permissible 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. `ClusterList` is a set of $\mathcal{N}_C$ linked lists, each of which lists the vertices in a given cluster. To facilitate quick updating of `ClusterList`, for each vertex v we provide a pointer 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. The cluster-related data structures are summarized in Table 2.2.

Now, on to the structures related to $M[\mathcal{C}]$, the set of possible moves. Recall from

Structure	Size	Description
`MoveList`	$< 2m + n$	The list of possible moves, pseudo-sorted by cost
`NumWithCost`	$2n$	The number of possible moves with given cost
`FirstWithCost`	$2n$	Pointers to the first `MoveList` node with given cost
`MovesOf`	n lists	Pointers to `MoveList` for each vertex being moved
`MovesTo`	$\mathcal{N}_C$ lists	Pointers to `MoveList` for each destination cluster

Table 2.3: Structures related to the set of available moves $M[\mathcal{C}]$.

the notion of equivalent-cost moves described in Section 2.2.1 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 $\delta_s(M)$ depending on whether we are in the naive scheme or the scaled scheme.

We maintain this list in an order sorted by the moves' associated integer costs (i.e. pseudo-sorted), least to greatest. If we are working under the naive scheme, this is simply $\delta_n(M)$. If we are 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 less than $2m - n$. 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 associated integer cost. As shown in Section 1.1, each move has an associated integer cost between $1 - n$ and $n - 1$, hence both of these structures have size $2n$.

Further, we provide two other orderings of the move list, as it were. `MovesOf` contains, for each vertex, a list of pointers to the nodes in `MoveList` that involve moving the vertex. Similarly, `MovesTo` contains, for each cluster, a list of pointers to the moves that have that cluster as their destination. Both `MovesOf` and `MovesTo` have the same size as `MoveList`.

Structure	Size	Description
`TabuList`	$\mathcal{L}_T$	The cyclic tabu list
`TabuVertices`	n	The tabu count for each vertex

Table 2.4: Tabu data structures.

The data structures applicable to $M[\mathcal{C}]$ are listed in Table 2.3. Of course, the issue of maintaining these structures must be addressed (see Section 2.5).

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 count of each vertex (i.e. the number of times that it appears in the tabu list). Initially, the list is empty. With each move made, the moved vertex is added to the list and it becomes tabu if its tabu count exceeds $\mathcal{T}_T$, the tabu tolerance. 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 2.4.

2.5 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 1.1 and 1.2, 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

$$C_n(G, \mathcal{C}) = \frac{1}{2} \sum_{v \in V} \alpha_v \tag{2.1}$$

and

$$C_s(G, \mathcal{C}) = \frac{n-1}{3} \sum_{v \in V} \frac{\alpha_v}{\beta_v} \tag{2.2}$$

for our two cost functions. Note that these α_v and β_v values are stored in `CostNumerator` and `CostDenominator` as in Table 2.2. `CostNumerator` is applicable to the naive cost scheme, but both are used in the scaled scheme.

2.5.1 The Wake of a Move

Most of the work done by the RNSC algorithm involves updating the move list when a move is made, so it is very important to know exactly which nodes in the list need to be moved and modified within the data structures. To this end, we define the *wake* of a move.

Definition 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* wake *of M, denoted $\mathcal{W}(M)$, as the union of the following sets of moves:*

- *Any permitted move of a vertex to or from C_i*
- *Any permitted move of a vertex to or from C_j.*

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

Theorem 1. *Given a move M, $\mathcal{W}(M)$ 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.*

This is proved as Theorem 4 in King's thesis [7] and can be taken for granted.

Let us note at this point the size of $\mathcal{W}(M)$, given a move M that moves v from C_i to C_j. It is the number of possible moves into or out of C_i or C_j, so we know that as far as rough bounds go, we have

$$|\mathcal{W}(M)| \leq \sum_{u \in C_i \cup C_j} (\#^1_{\mathrm{out}}(G, \mathcal{C}, u) + 1) \tag{2.3}$$

$$\leq \sum_{u \in C_i \cup C_j} (\deg(u) + 1). \tag{2.4}$$

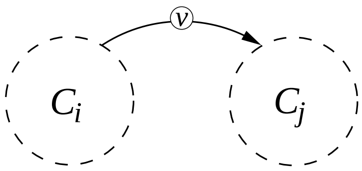

Figure 2.5: Moving vertex v from cluster C_i to cluster C_j.

2.5.2 The Naive Cost Function

The naive cost function (given in Equation 1.1) is very simple to update and maintain. How δ_n changes for the wake of a move 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 2.5), it is not hard to see what the change in cost will be. Recall 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 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. $N(v) \cap V(C_i)$. 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 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

$$\begin{aligned}\delta_n(M) &= \quad (|V(C_j)| + |N(v)| - 2|N_j(v)| - 1) \\ &\quad - (|V(C_i)| + |N(v)| - 2|N_i(v)| - 1) \\ &= |V(C_j)| - |V(C_i)| + 2(|N_i(v)| - |N_j(v)|). \end{aligned} \tag{2.5}$$

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 the wake of M, $\mathcal{W}(M)$, consists of any move to or from C_i or C_j. This includes, of course, any movement of v itself. Let M' be a move 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 2.5 and Theorem 1, 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 $u \sim v$.

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

Table 2.5: The effect of a move M on the change in naive cost associated with another move M'.

No other figures applicable to Equation 2.5 will change. From the equation, we can see that these points will cumulate as shown in Table 2.5. Suppose $u = v$, then. It is clear that the clustering's 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 $\mathcal{W}(M)$.

2.5.3 The Scaled Cost Function

The scaled cost function, unfortunately, cannot be maintained as quickly or easily. Far more work is required, both to initialize and to update. The wake of a move 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 2.2 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, for each vertex v. Let v be in C_i. Note that

$u \in C_i$?	$u \in C_j$?	$u \sim v$?	$\alpha'_u - \alpha_u$	$\beta'_u - \beta_u$	$\frac{\alpha'_u\beta_u - \alpha_u\beta'_u}{\beta'_u\beta_u}$
✓	×	✓	$+1$	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 2.6: The effect of a move M on α_u and β_u.

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 2.5.2, 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 and call the 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

$$\delta_s(M) = \frac{n-1}{3} \sum_{u \in C_i \cup C_j} \left(\frac{\alpha'_u}{\beta'_u} - \frac{\alpha_u}{\beta_u} \right) = \frac{n-1}{3} \sum_{u \in C_i \cup C_j} \left(\frac{\alpha'_u\beta_u - \alpha_u\beta'_u}{\beta'_u\beta_u} \right) \tag{2.6}$$

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 2.6. 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 in 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, when we make a move M we must update $\delta_s(M')$ for all moves $M' \in \mathcal{W}(M)$. 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 2.6, then, we get

$$\begin{aligned} \delta'_s(M') &= \frac{n-1}{3} \sum_{w \in C_k \cup C_l} \left(\frac{\alpha''_w}{\beta''_w} - \frac{\alpha'_w}{\beta'_w} \right) \\ &= \delta_s(M') + \frac{n-1}{3} \sum_{w \in C_k \cup C_l} \left(\frac{\alpha''_w}{\beta''_w} - \frac{2\alpha'_w}{\beta'_w} + \frac{\alpha_w}{\beta_w} \right). \end{aligned} \tag{2.7}$$

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.

2.6 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 easier than choosing a global move. In the case of shuffling diversification, we merely choose a vertex at random, then choose a permissible 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 randomly chosen 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 2.5.2, the size of $\mathcal{W}(M)$ for a move M is variable; since it depends on the clustering, particularly on the number of cross edges with an endpoint in C_i or C_j, the wake sizes will be small in a good clustering.

As seen in Table 2.5, we know exactly how to update the cost for a move in $\mathcal{W}(M)$ 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 we have the structures `MovesOf` and `MovesTo`, we can immediately access the set of nodes in `MoveList` corresponding to $\mathcal{W}(M)$. Hence we can update in the naive scheme in $\mathcal{O}(|\mathcal{W}(M)|)$ time, since there are no other data structures that require time consuming updates.

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 $\mathcal{W}(M)$ is expensive, and unfortunately must be done for every such M'.

To find the new value of $\delta_s(M')$ for a move in $\mathcal{W}(M)$, we simply recalculate it from scratch. This takes $\mathcal{O}(|C_k| + |C_l|)$ time for a move from C_k to C_l. There are $|\mathcal{W}(M)|$ moves to update, so updating in the scaled scheme takes about quadratic time in the average size of a cluster, as a very rough idea.

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.

Also consider the fact that the values in Table 2.6 are computations that are repeated many times. By preprocessing and storing these values for small values of α and β, we can make significant savings in time.

Chapter 3

Experimental Results

3.1 Test Data

For the purposes of analyzing the performance of the RNSC algorithm, we consider several types of graphs. In addition to considering some real-life protein-protein interaction (PPI) graphs, we consider several kinds of random graphs.

3.1.1 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 [1, 2, 6, 8, 12, 13, 14, 19].

Von Mering *et al.* [19] 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. Table 3.1 presents a summary of these graphs.

3.1.2 Random Graphs

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

Name	$\lvert V\rvert$	$\lvert E\rvert$	Density	Source
Y_{2k}	988	2455	5.035×10^{-3}	[19]
Y_{11k}	2401	11,000	3.818×10^{-3}	[19]
Y_{45k}	4687	45,000	4.098×10^{-3}	[19]
Y_{78k}	5321	78,390	5.538×10^{-3}	[19]

Table 3.1: Protein-protein interaction graphs for yeast, used as test data.

Erdös-Rényi graphs are the most common form of random graph. Given a set of n vertices, each pair of vertices is connected with probability p. We call this graph $G_E(n, p)$.

Scale-free random graphs are designed to be good models of things like the worldwide web and social interaction networks, in which a small number of vertices have very high degree and most have very low degree. More formally, the vertex degrees tend to follow a power-law distribution. We construct the scale-free random graph $G_S(n, k)$ by starting with an independent set of size k and iteratively adding vertices, which are then made adjacent to k other vertices with probability proportional to the current degree of the other vertices. These are more thoroughly described by Barabasi and Oltvai [2].

Geometric random graphs are made, in our case, by placing n vertices on an $l \times w$ grid and connecting two vertices if and only if the Euclidean distance between them is less than d. We denote such a graph $G_G(n, l, w, d)$. Pržulj *et al.* showed that certain types of geometric graphs are, in many ways, a better model of biological networks than scale-free random graphs [13].

3.2 Parameter Training and Statistical Results

Since the performance of RNSC depends heavily on the input parameters that are given, it is essential that we explore these dependencies. We will investigate the consequences of the choice of diversification parameters, tabu parameters, and search length.

3.2.1 Diversification and Tabu Parameters

With diversification and a tabu list, the RNSC has the capability to efficiently search many minima in the search space. In this section we will consider a standard tabu list, i.e. one with $\mathcal{T}_T = 1$, since it was shown in King's thesis that extending the tabu tolerance did not offer an advantage for the problem.

Figures 3.1 through 3.4 show the results of experimental runs using a variety of diversification parameters. For consistency, each run uses the same naive and scaled stopping tolerances and tabu length (5, 500, and 20 respectively). The graphs used are the PPI graph Y_{11k}, an Erdös-Rényi random graph, a scale-free random graph, and a geometric random graph, all of which have approximately the same number of vertices and edges.

We can see that $\mathcal{F}_D = n/50$ and $\mathcal{L}_D = \mathcal{F}_D/5$ yield fairly good results in terms of both score and time, but that no set of parameters is clearly the best for all classes of graphs. It is evident, however, that shuffling diversification is preferable to destructive diversification for all classes of graphs.

To examine the effect of the tabu list on the performance of the RNSC algorithm, we use the same graphs with $\mathcal{F}_D = 50$ and $\mathcal{L}_D = 10$, leaving all other parameters unchanged. The results are shown in Figures 3.5 through 3.8. Each type of graph has a fairly similar response to the changing tabu length, and RNSC clusters each quite well (and quite quickly) with a tabu length of around $n/100$.

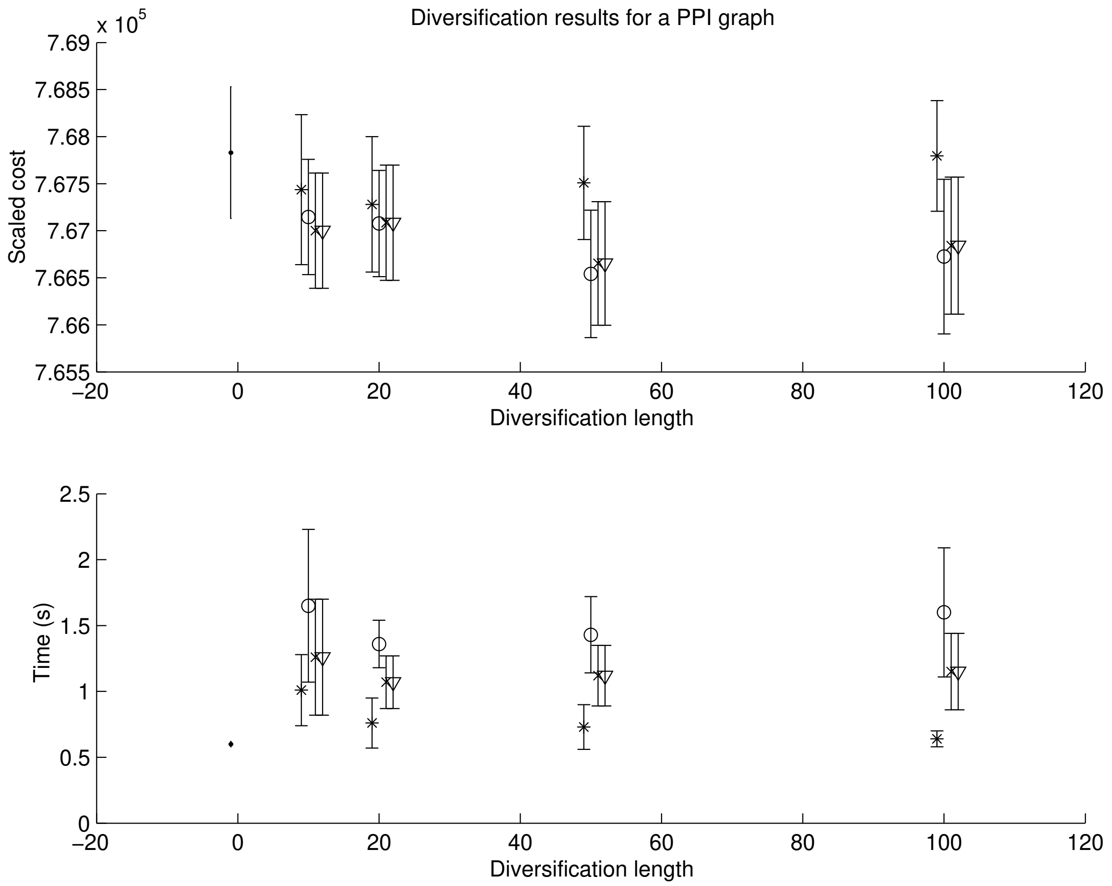

Figure 3.1: Diversification results for the PPI graph Y_{11k}. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{F}_D \in \{10, 20, 50, 100, \infty\}$, with destructive diversification marked with $*$, no diversification marked with $\bullet$, and shuffling diversification with $\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$, $\lfloor \mathcal{F}_D/5 \rfloor$, and $\lfloor \mathcal{F}_D/10 \rfloor$ marked with $\circ$, $\times$, and ∇ respectively.

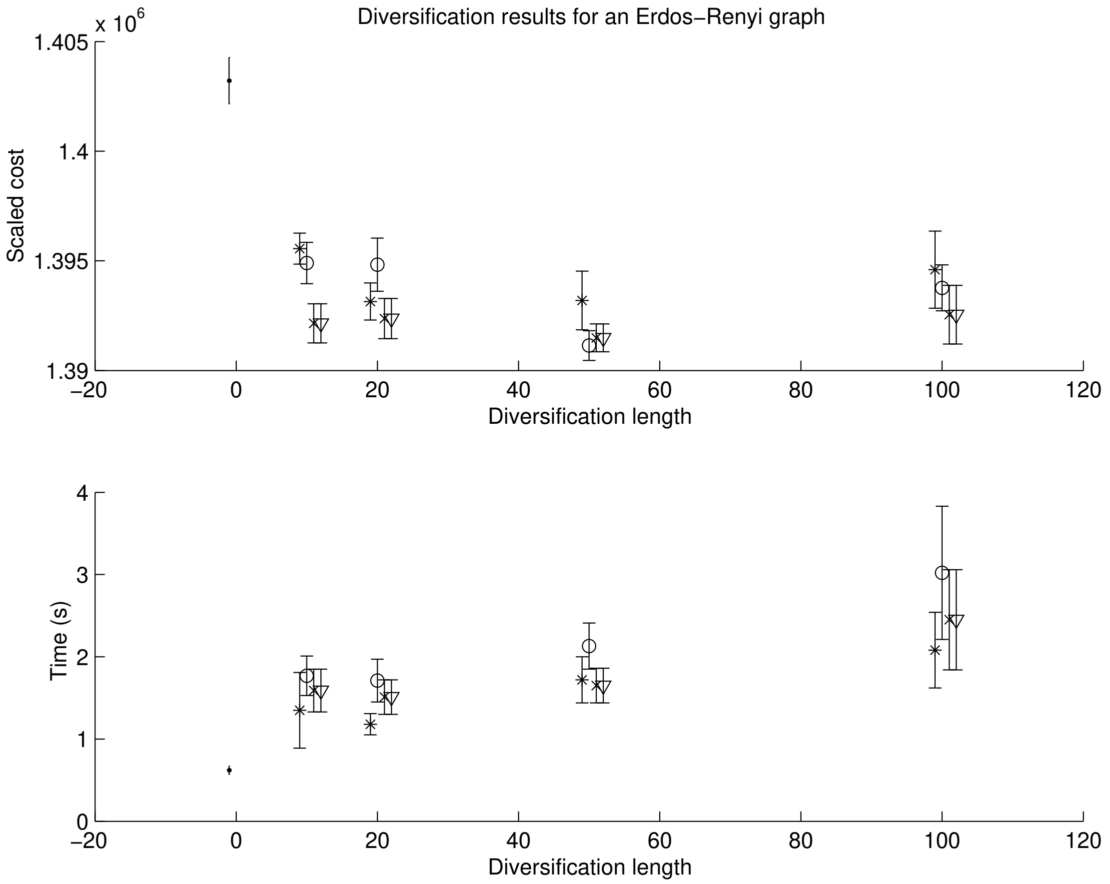

Figure 3.2: Diversification results for a random Erdös-Rényi graph on 2400 vertices and 10,997 edges. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{F}_D \in \{10, 20, 50, 100, \infty\}$, with destructive diversification marked with $*$, no diversification marked with $\bullet$, and shuffling diversification with $\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$, $\lfloor \mathcal{F}_D/5 \rfloor$, and $\lfloor \mathcal{F}_D/10 \rfloor$ marked with $\circ$, $\times$, and ∇ respectively.

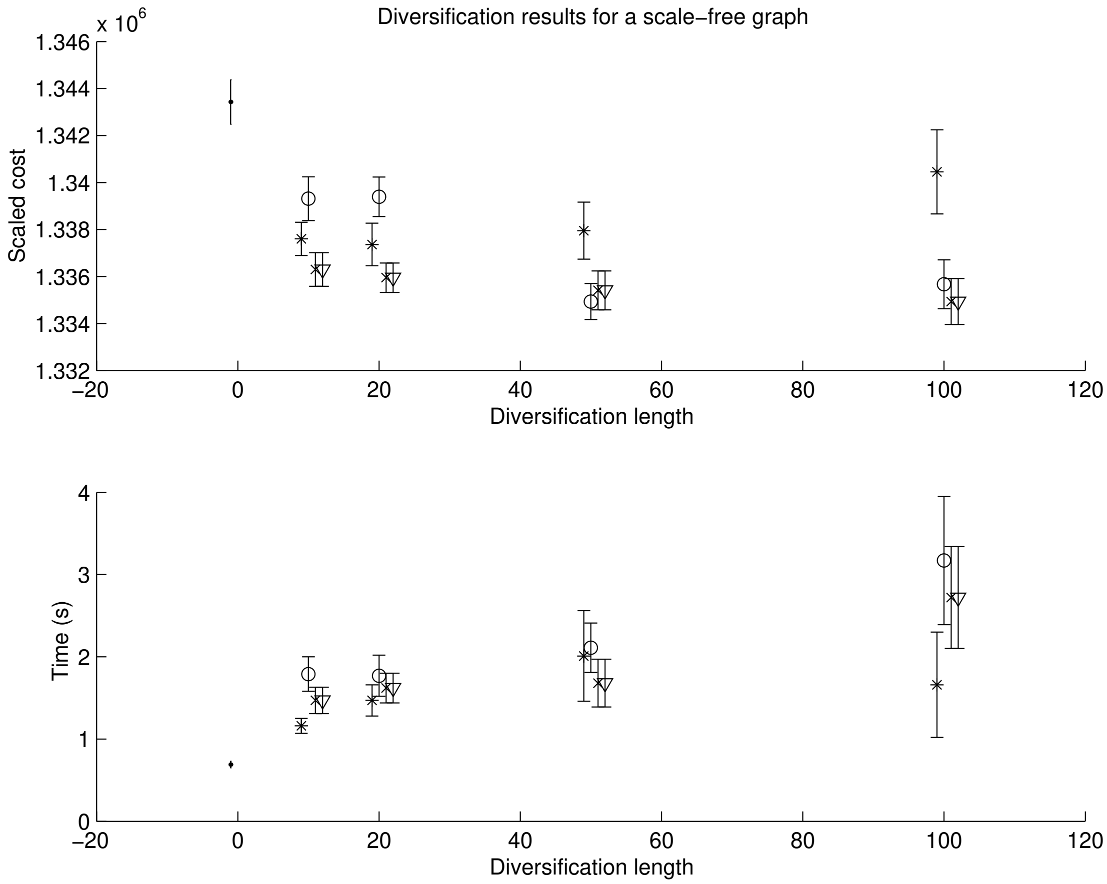

Figure 3.3: Diversification results for a random scale-free graph on 2400 vertices and 11,000 edges. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{F}_D \in \{10, 20, 50, 100, \infty\}$, with destructive diversification marked with $*$, no diversification marked with $\bullet$, and shuffling diversification with $\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$, $\lfloor \mathcal{F}_D/5 \rfloor$, and $\lfloor \mathcal{F}_D/10 \rfloor$ marked with $\circ$, $\times$, and ∇ respectively.

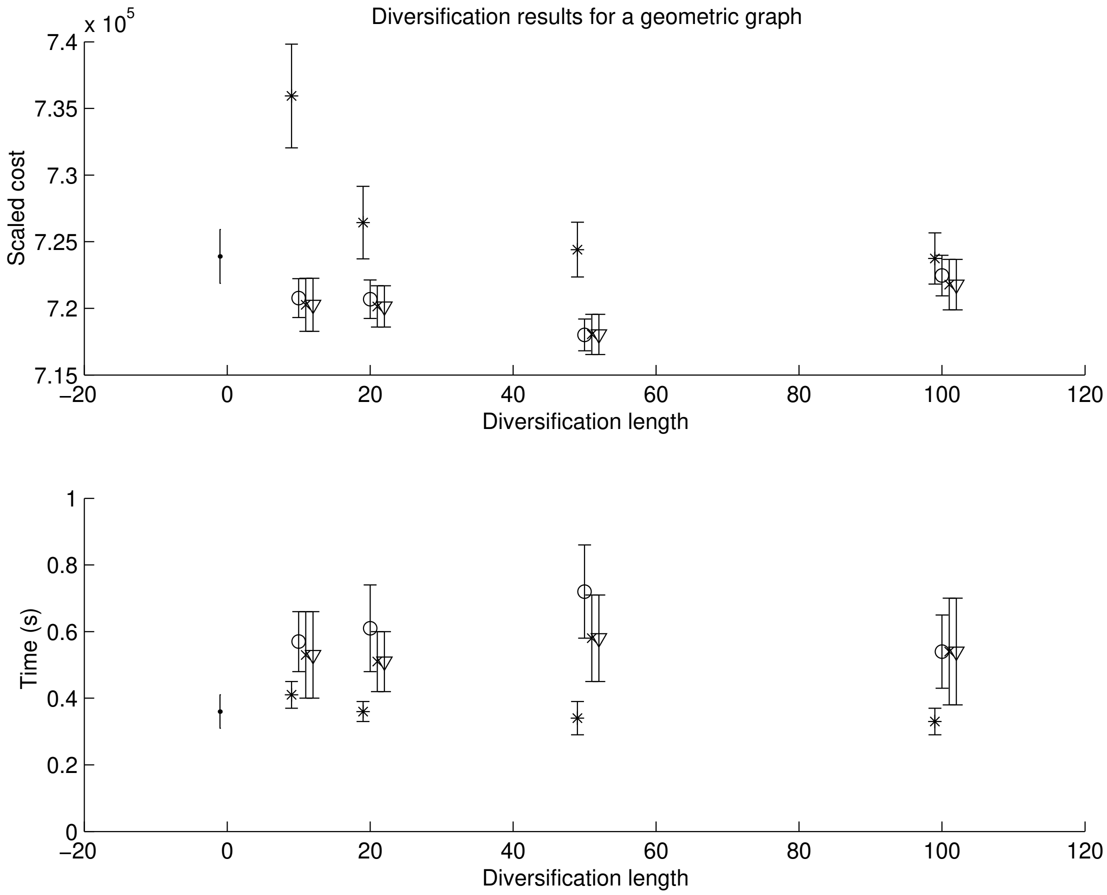

Figure 3.4: Diversification results for a random geometric graph on 2399 vertices and 11,000 edges. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{F}_D \in \{10, 20, 50, 100, \infty\}$, with destructive diversification marked with $*$, no diversification marked with $\bullet$, and shuffling diversification with $\mathcal{L}_D = \lfloor \mathcal{F}_D/3 \rfloor$, $\lfloor \mathcal{F}_D/5 \rfloor$, and $\lfloor \mathcal{F}_D/10 \rfloor$ marked with $\circ$, $\times$, and ∇ respectively.

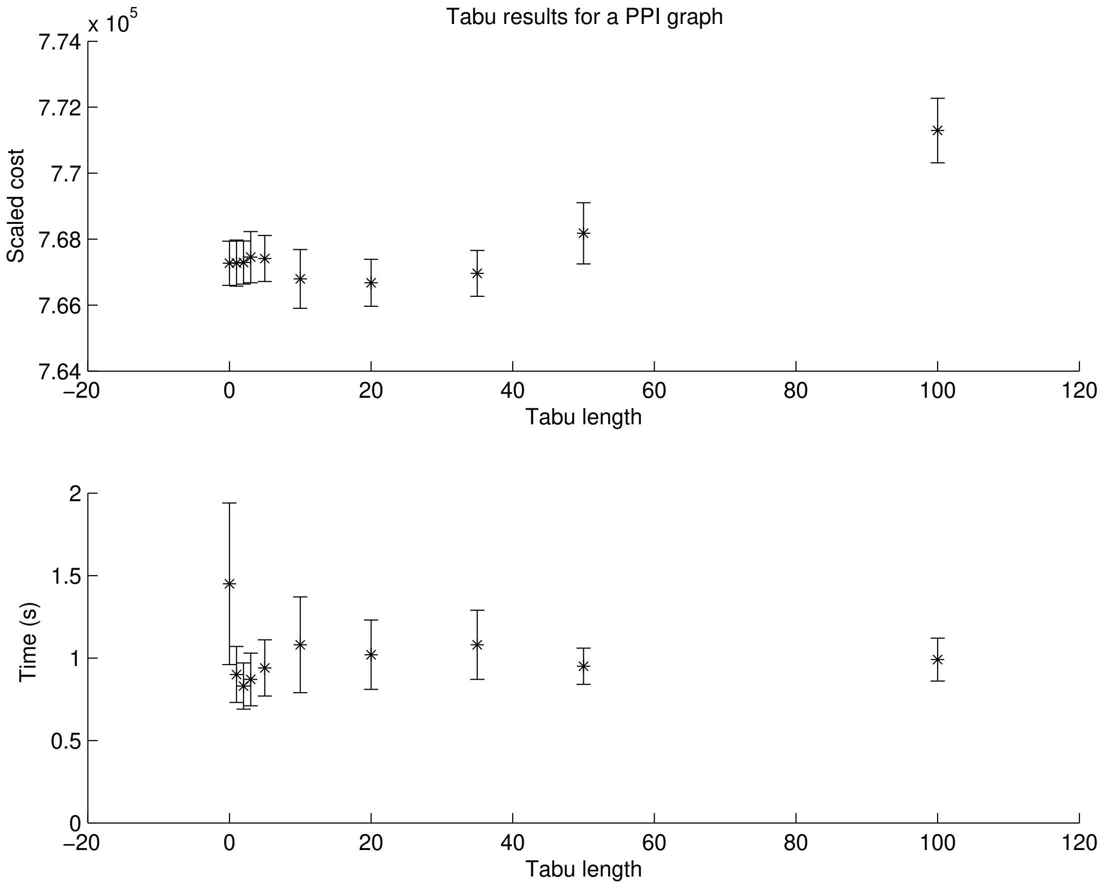

Figure 3.5: Tabu results for the PPI graph Y_{11k}. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{L}_T \in \{0, 1, 2, 3, 5, 10, 20, 50, 100\}$.

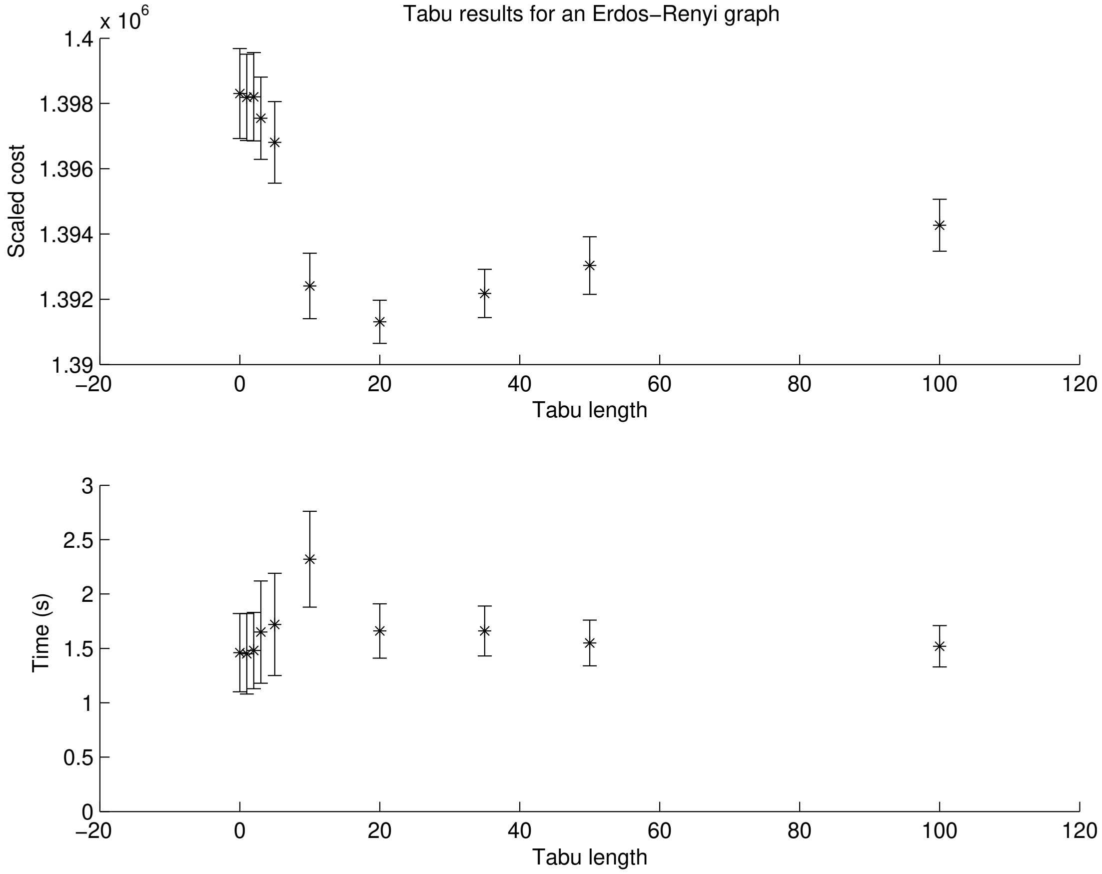

Figure 3.6: Tabu results for an Erdös-Rényi random graph on 2400 vertices and 10,997 edges. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{L}_T \in \{0, 1, 2, 3, 5, 10, 20, 50, 100\}$.

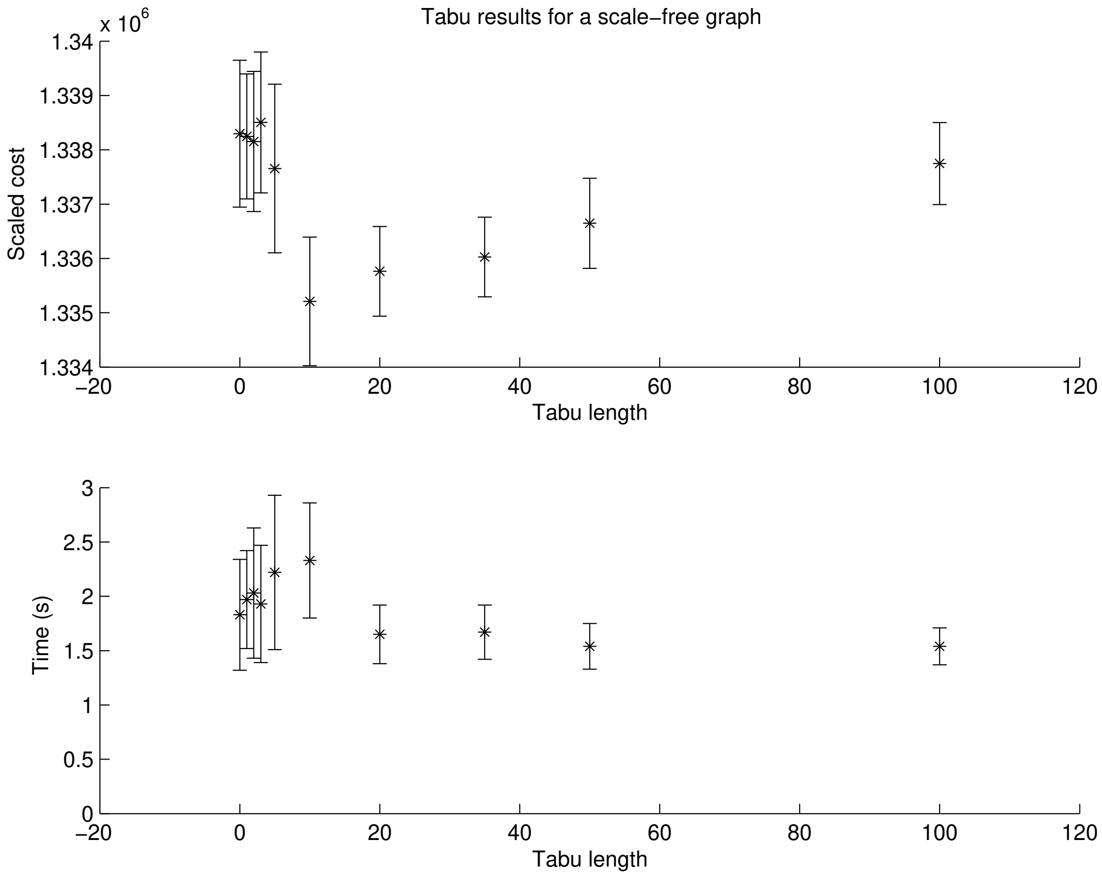

Figure 3.7: Tabu results for a random scale-free graph on 2400 vertices and 11,000 edges. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{L}_T \in \{0, 1, 2, 3, 5, 10, 20, 50, 100\}$.

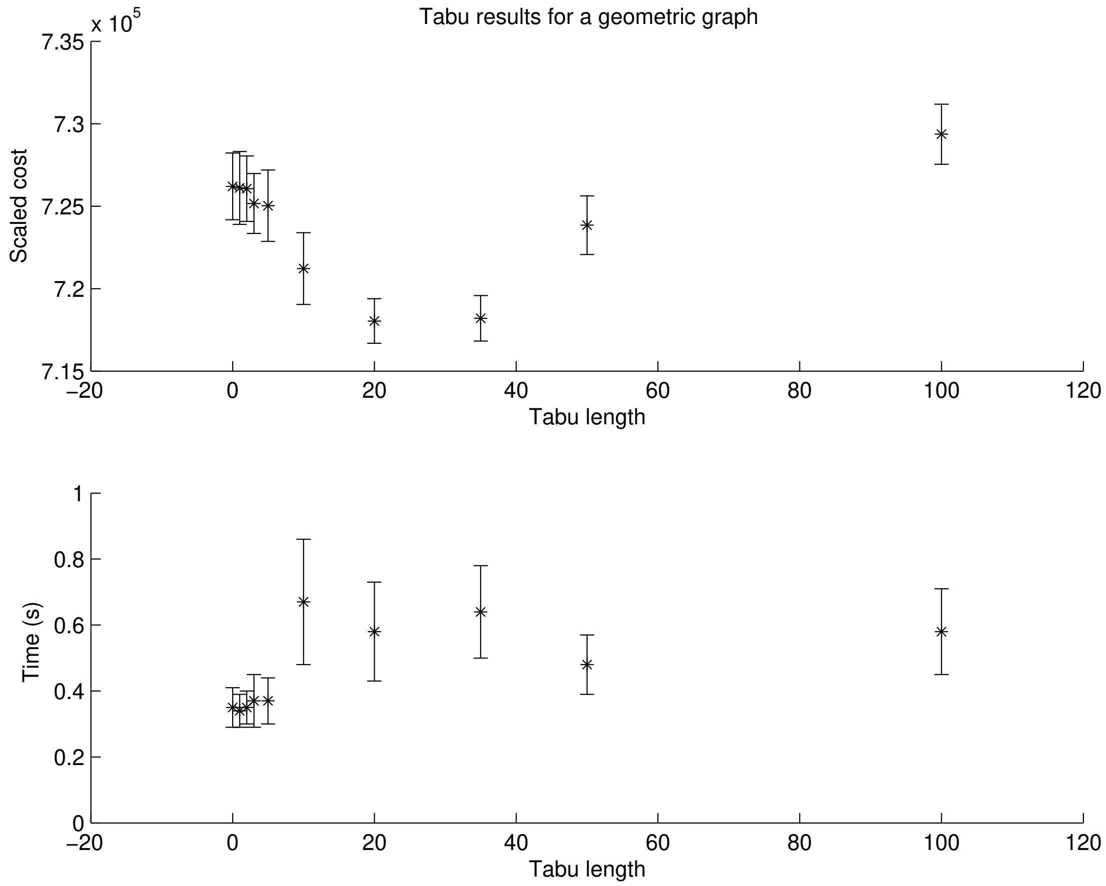

Figure 3.8: Tabu results for a random geometric graph on 2399 vertices and 11,000 edges. The mean scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{L}_T \in \{0, 1, 2, 3, 5, 10, 20, 50, 100\}$.

3.2.2 Search Length

Obviously if we increase the length of the search, we are likely to improve the cost of the final clustering and increase the time taken by the RNSC algorithm. Hence we are interested in seeing the results of an increased stopping tolerance. To this end, we use the four graphs that we used in Section 3.2.1 with $\mathcal{F}_D = 50$, $\mathcal{L}_D = 10$, and $\mathcal{T}_T = 20$. As before, we use a naive stopping tolerance of 5, but this makes little difference. Experimental results are seen in Figures 3.9 through 3.12.

The results follow intuition. For all graph types, the final scaled cost decreases as the stopping tolerance is increased and the standard deviation of the cost decreases similarly. The time taken per experiment does the opposite: the mean, minimum, and standard deviation of the time increase more or less linearly with the scaled stopping tolerance. Not surprisingly, we are presented with a choice between speed and quality when we choose the naive stopping tolerance.

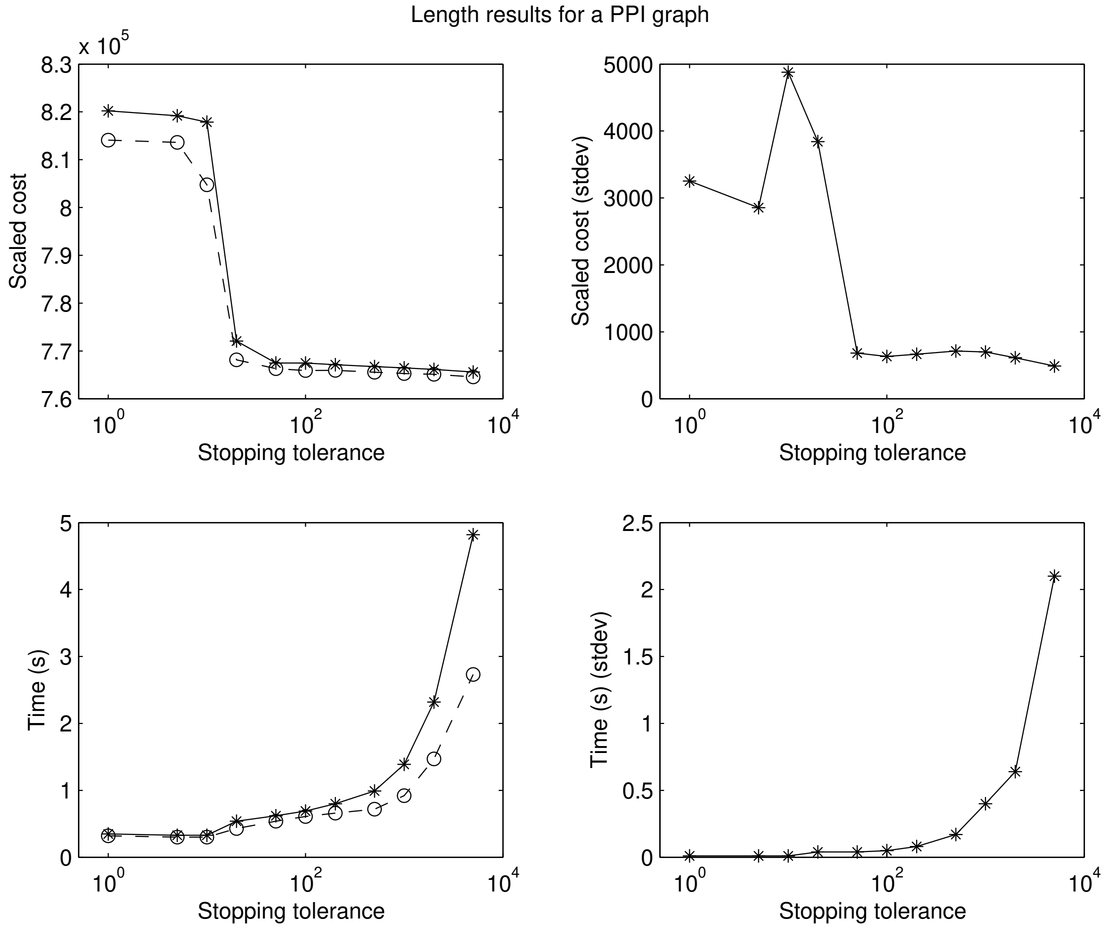

Figure 3.9: Stopping tolerance results for the PPI graph Y_{11k}. The mean ($*$) and minimum ($\circ$) scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{T}_s \in \{1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$.

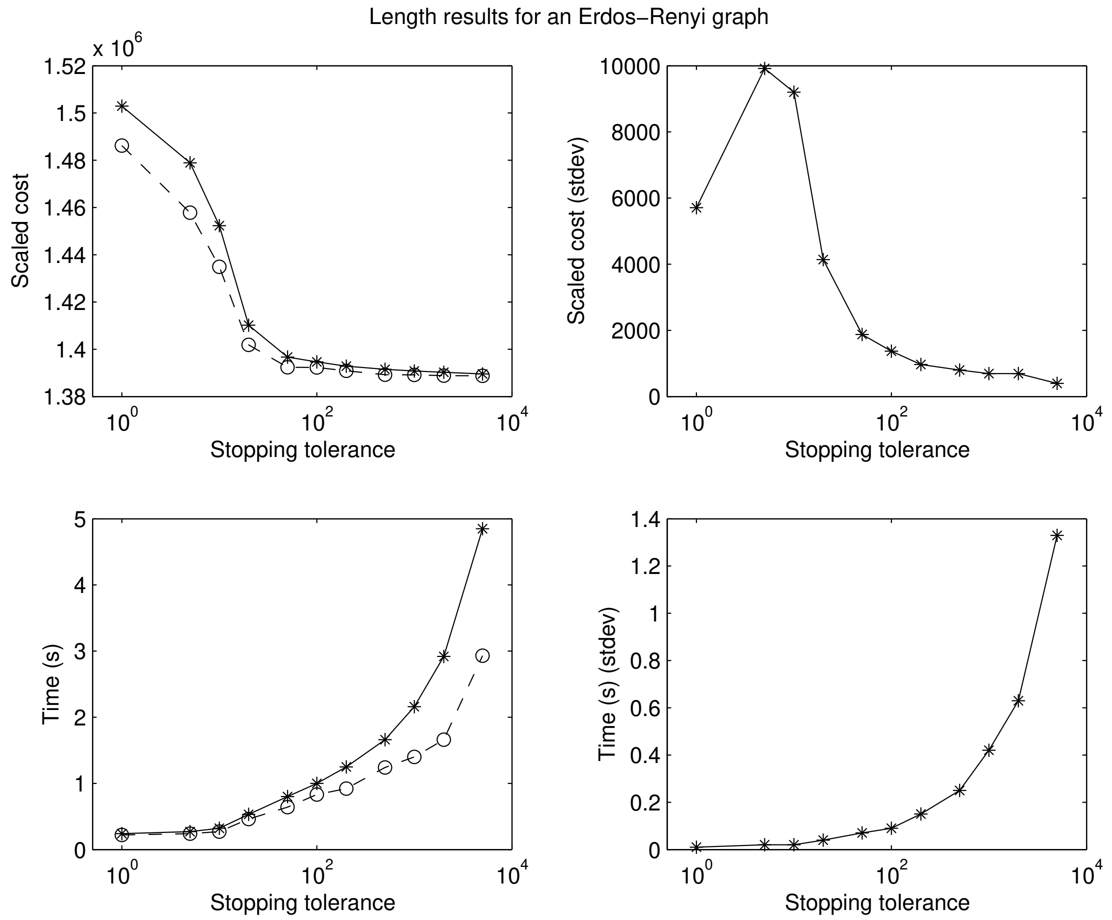

Figure 3.10: Stopping tolerance results for an Erdös-Rényi random graph on 2400 vertices and 10,997 edges. The mean ($*$) and minimum ($\circ$) scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{T}_s \in \{1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$.

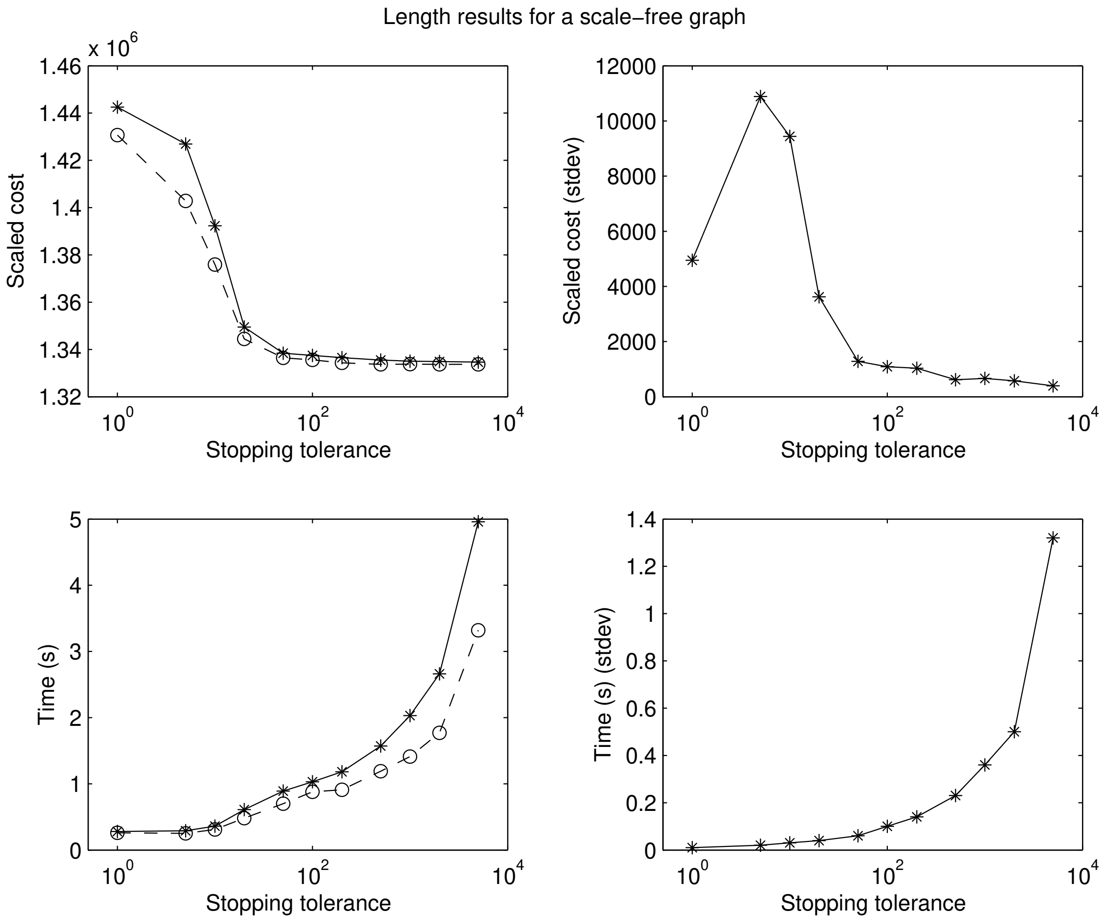

Figure 3.11: Stopping tolerance results for a random scale-free graph on 2400 vertices and 11,000 edges. The mean ($*$) and minimum ($\circ$) scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{T}_s \in \{1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$.

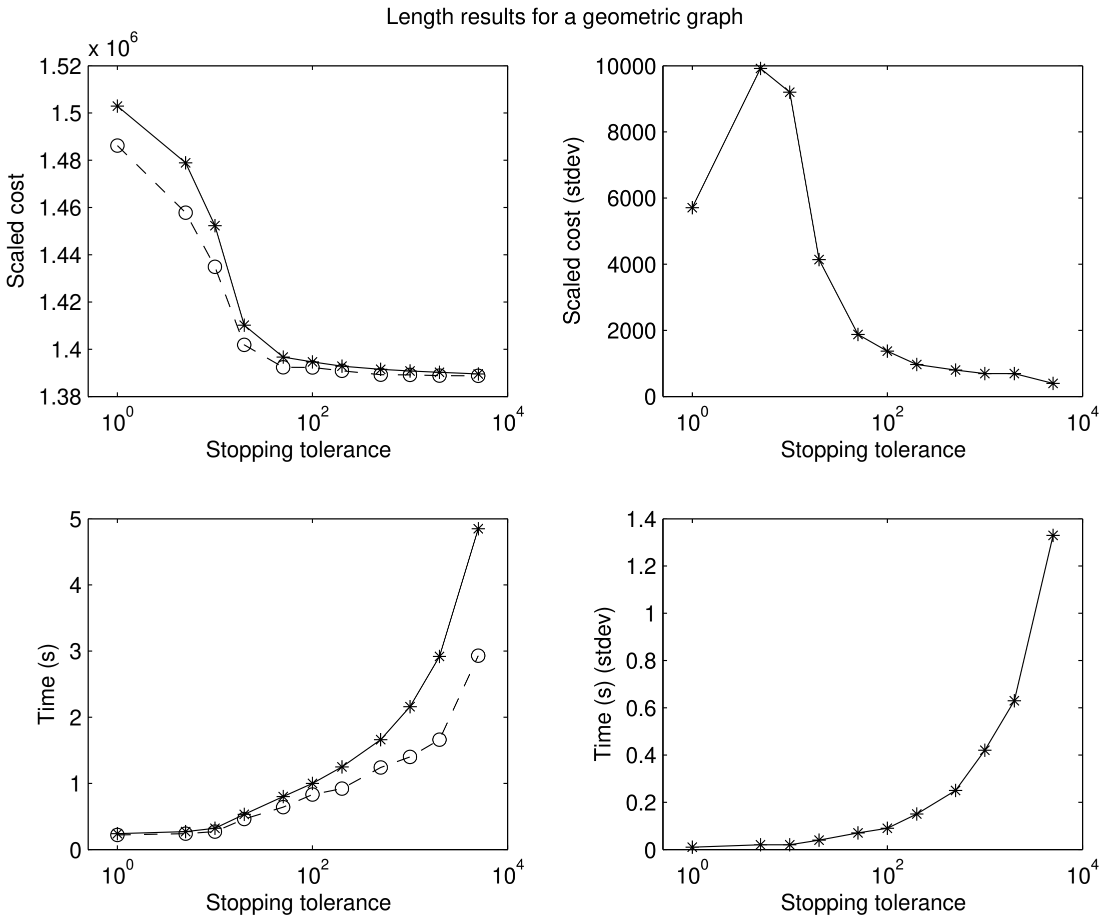

Figure 3.12: Stopping tolerance results for a random geometric graph on 2399 edges and 11,000 vertices. The mean ($*$) and minimum ($\circ$) scores and times are given for 50 experiments, along with their standard deviations. Trials were run with $\mathcal{T}_s \in \{1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$.

3.3 Wake sizes

We are interested in some general idea of the wake sizes during a typical run of the RNSC algorithm. Figures 3.13 through 3.16 show mean wake sizes over 50 experiments each on a PPI graph, an Erdös-Rényi graph, a scale-free graph, and a geometric graph, all on about 2400 vertices and 11,000 edges. These are run with 10 shuffling moves every 50, and with up to $|V|$ clusters. While there is a huge amount of variation among the graph types, each experiment begins with relatively small wake sizes. This is because initially, the clustering is determined by throwing the vertices into $|V|$ bins uniformly, hence the initial clustering consists, generally, of tiny clusters. The diversification explains most of the noise in the results.

There is an interesting point to be seen here: The dramatic difference in the wake sizes between the PPI graph and the scale-free graph lend supoport to the findings of Pržulj *et al.*, i.e. that scale-free graphs make, in many ways, poor models of PPI networks [13]. Note that we cannot use the data on the geometric graph in this case, since the type of geometric graph that we are using (two-dimensional, uniform distribution) is obviously a poor model for PPI networks.

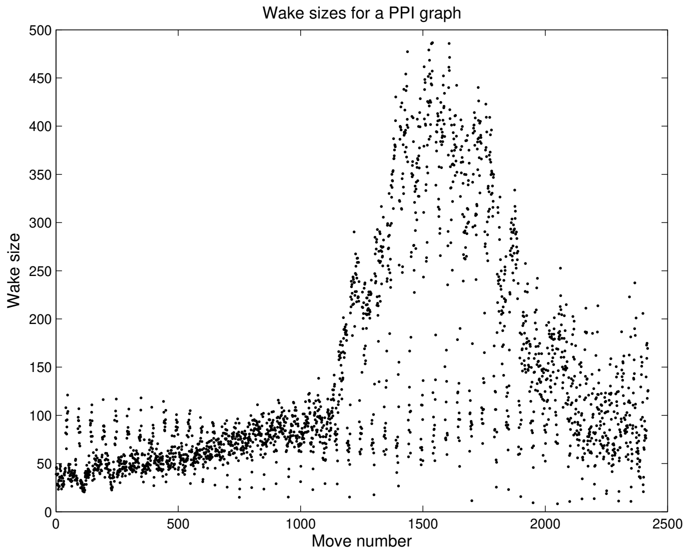

Figure 3.13: Mean wake sizes for 50 experiments on the PPI graph Y_{11k}.

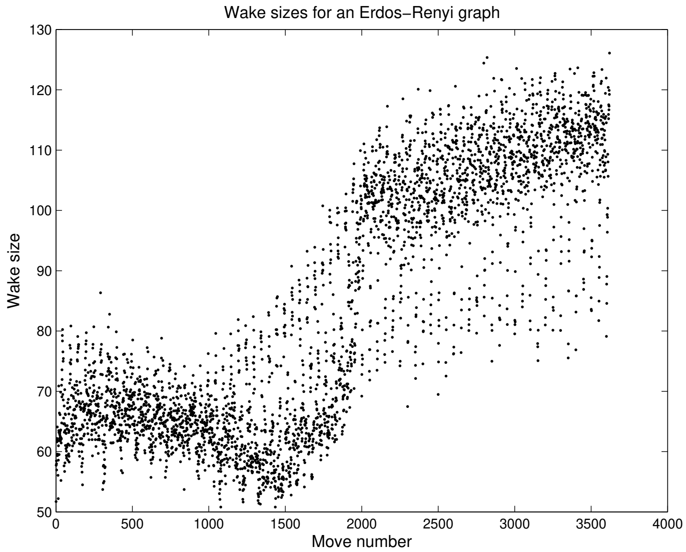

Figure 3.14: Mean wake sizes for 50 experiments on an Erdös-Rényi random graph on 2400 vertices and 10,997 edges.

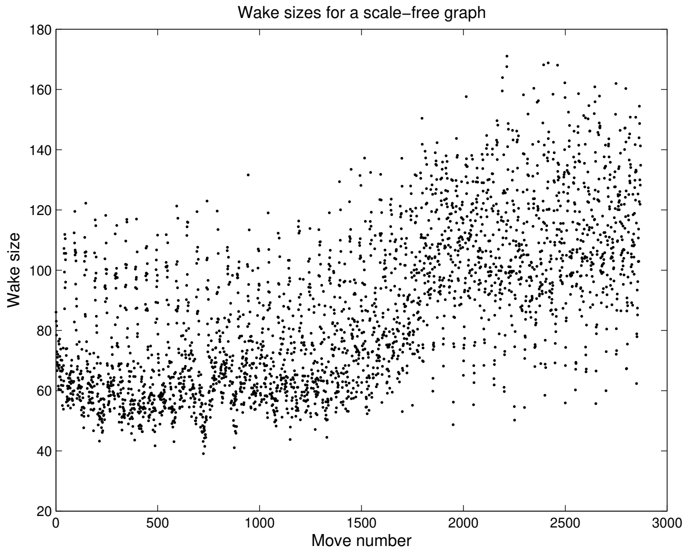

Figure 3.15: Mean wake sizes for 50 experiments on a random scale-free graph on 2400 vertices and 11,000 edges.

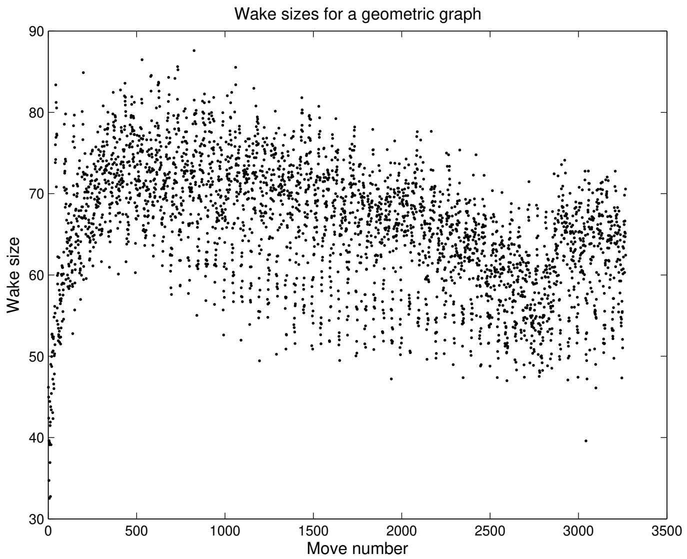

Figure 3.16: Mean wake sizes for 50 experiments on a random geometric graph on 2399 edges and 11,000 vertices.

3.4 Performance Results

We would like to benchmark the RNSC algorithm against a comparable clustering algorithm in order to judge its performance, both in terms of quality and efficiency. We use van Dongen's MCL (Markov cluster) algorithm as comparison. MCL is a deterministic graph clustering algorithm that uses matrix expansion and contraction to find clusterings as equilibria in simulated flow, and has been used with great effectiveness in the area of protein network analysis [3, 17, 18]. Specifically, we use the November 9, 2004 distribution, `mcl-04-314`, available at `www.micans.org/mcl/`.

We are mainly interested in three things: clustering quality, time requirements, and memory requirements. The memory requirements for MCL and RNSC are comparable. For the PPI graph Y_{78k}, MCL used around 26MB of memory, while RNSC used around 25MB. For a random scale-free graph on 10,000 vertices and 100,000 edges, MCL and RNSC used 13MB and 34MB of memory, respectively.

3.4.1 Random Graphs

To evaluate the scores and times of the two algorithms, we use, in addition to our four PPI networks, 15 random graphs of each type: Erdös-Rényi, scale-free, and geometric. These graphs consist of three each of size 500, 1000, 2000, 5000, and 10,000, each with a different density. The instances of size 500, 1000, and 2000 have density $10^{-1.5}$, 10^{-2}, and $10^{-2.5}$, while the larger graphs have density $10^{-2.5}$, 10^{-3}, and $10^{-3.5}$. The results for random graphs are shown in Figures 3.17 through 3.19.

The MCL algorithm was run six times on each graph, with inflation value $I \in \{1.5, 1.7, 2.0, 2.5, 3.0, 4.0\}$. The inflation value affects the granularity of the clustering. The lower the inflation value, the larger the clusters. Of these six runs, the one resulting in the clustering with the lowest scaled score is used. RNSC was run 20 times on each graph with $\mathcal{T}_n = 5$, $\mathcal{T}_s = n/25$, $\mathcal{F}_D = n/50$, $\mathcal{L}_D = n/250$, and $\mathcal{L}_T = n/100$.

Unsurprisingly, MCL does not produce a lower score for any graph; RNSC is built around the performance criterion. However, the results are reasonably competitive. Recall, though, that small relative differences in score can represent rather large differences in clustering quality and the scores are shown on a logarithmic scale. RNSC ran faster

	MCL			RNSC	
Graph	Time (s)	Score	Best I	Mean time (s)	Mean score
Erdös-Rényi	63.64	1,562,932	1.7	1.16	1,394,831
Scale-free	8.88	1,481,527	3.0	1.16	1,337,513
Geometric	0.46	848,953	2.5	0.41	722,298
Y_{2k}	0.58	138,075	2.0	0.13	85,931
Y_{11k}	1.71	989,674	3.0	0.95	767,378
Y_{45k}	59.04	4,841,999	2.5	9.21	4,068,704
Y_{78k}	106.38	6,378,818	2.5	33.14	5,265,080

Table 3.2: Performance data for random and PPI graphs. The random graphs are all roughly the same size as Y_{11k}.

than MCL on most graphs, but not all. Results for the three random graphs used in parameter training are shown in Table 3.2, using 50 RNSC experiments for each graph with the same parameter values as we use here.

3.4.2 Protein-Protein Interaction Networks

Table 3.2 shows the results of experiments on the four PPI graphs using the same parameter values as the previous section, with 50 RNSC experiments run for each graph. Here it is more evident that the RNSC performs better than MCL according to our performance criteria – the results are very impressive. However, RNSC lacks a parameter to adjust the granularity of a clustering. This is mentioned in the next chapter. It should be noted that MCL runs much faster with higher values of I. For example, MCL processed Y_{2k} in 0.08 seconds with $I = 4.0$, but achieved a score of 145,072.

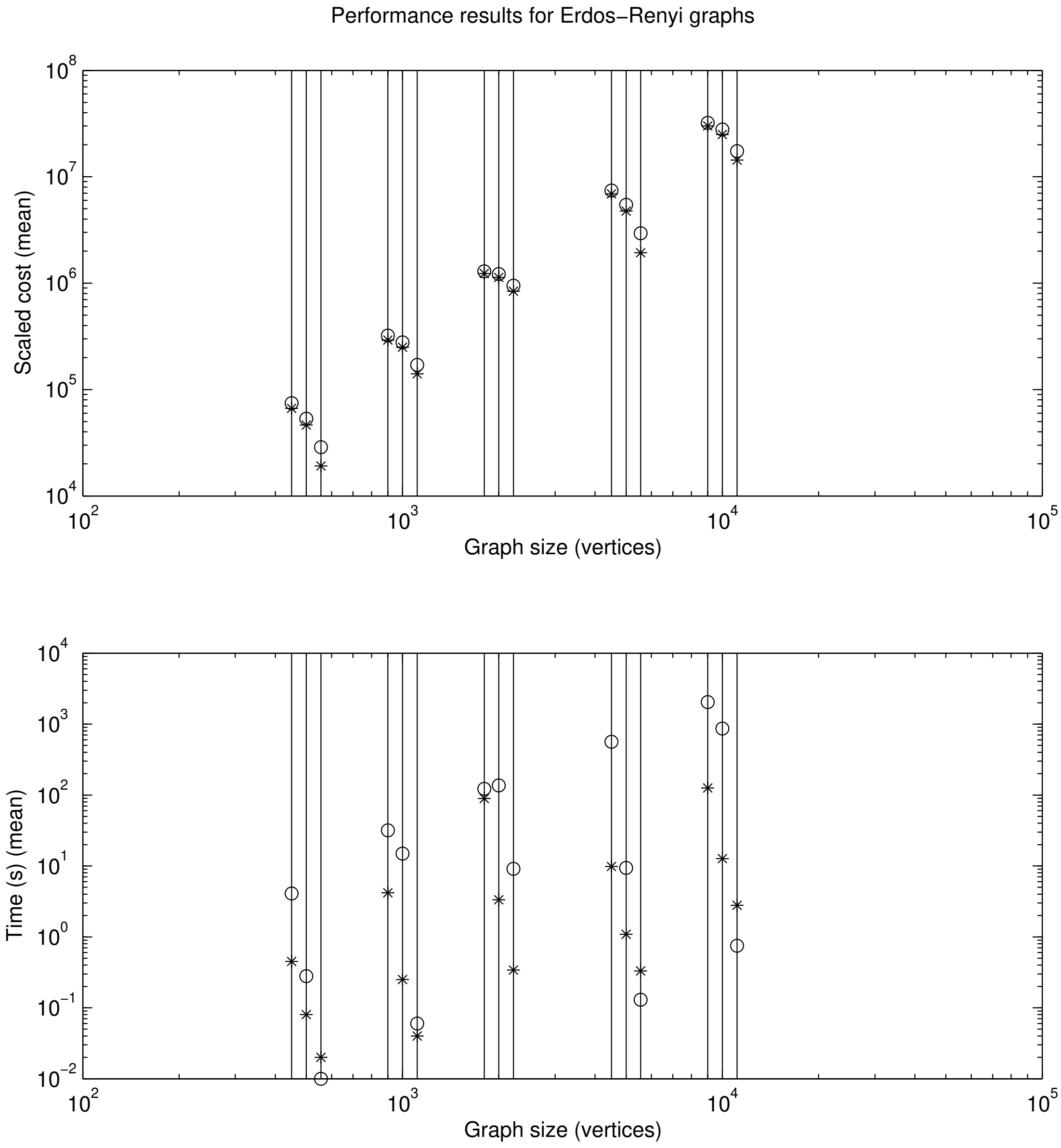

Figure 3.17: Performance results for Erdös-Rényi random graphs. Results for MCL are denoted by circles and results for RNSC (mean, 20 runs) are denoted by asterisks, and results for graphs of the same size are shown grouped in descending order of density. The precision of all measurements is 10^{-2}.

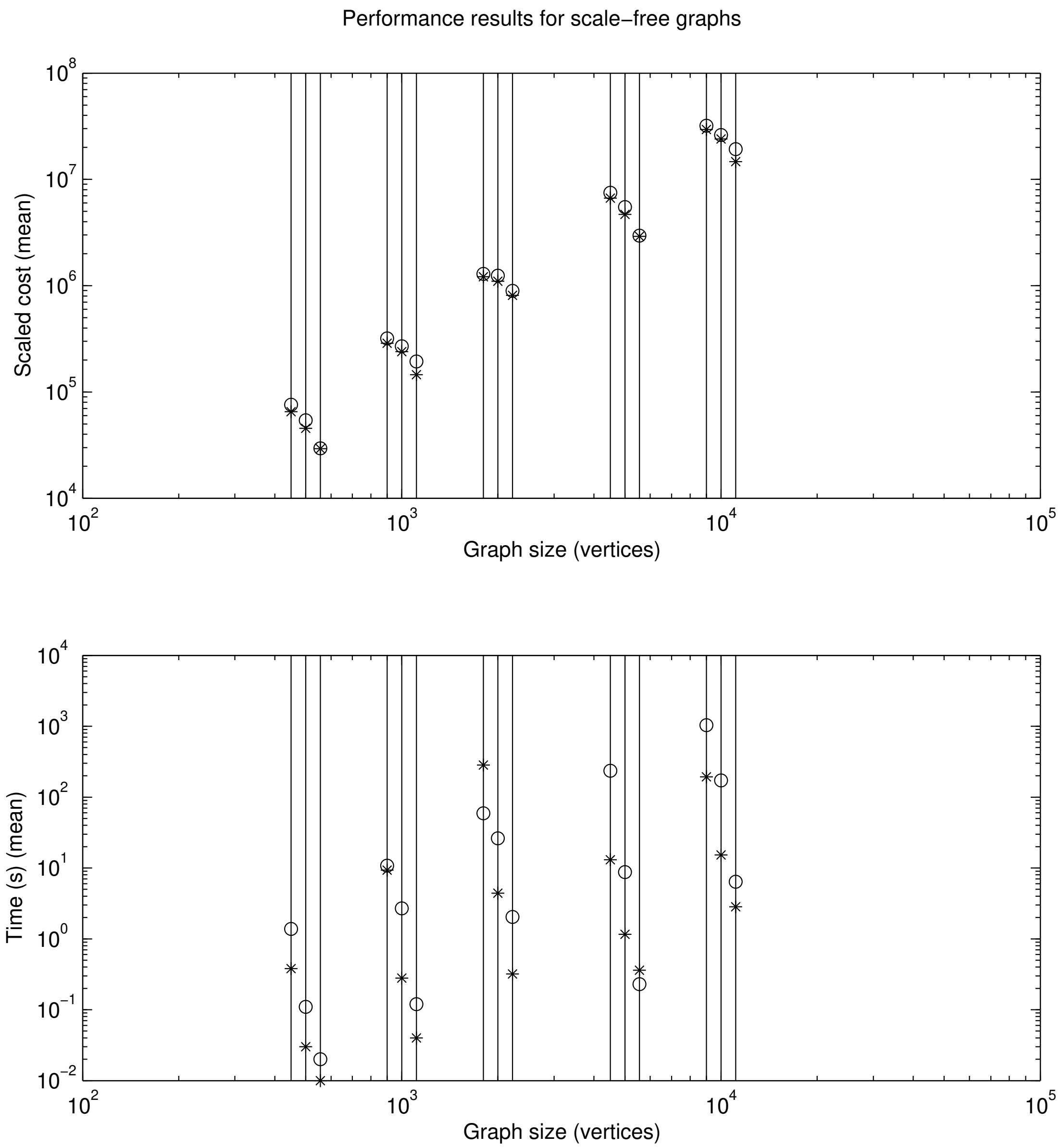

Figure 3.18: Performance results for random scale-free graphs. Results for MCL are denoted by circles and results for RNSC (mean, 20 runs) are denoted by asterisks, and results for graphs of the same size are shown grouped in descending order of density. The precision of all measurements is 10^{-2}.

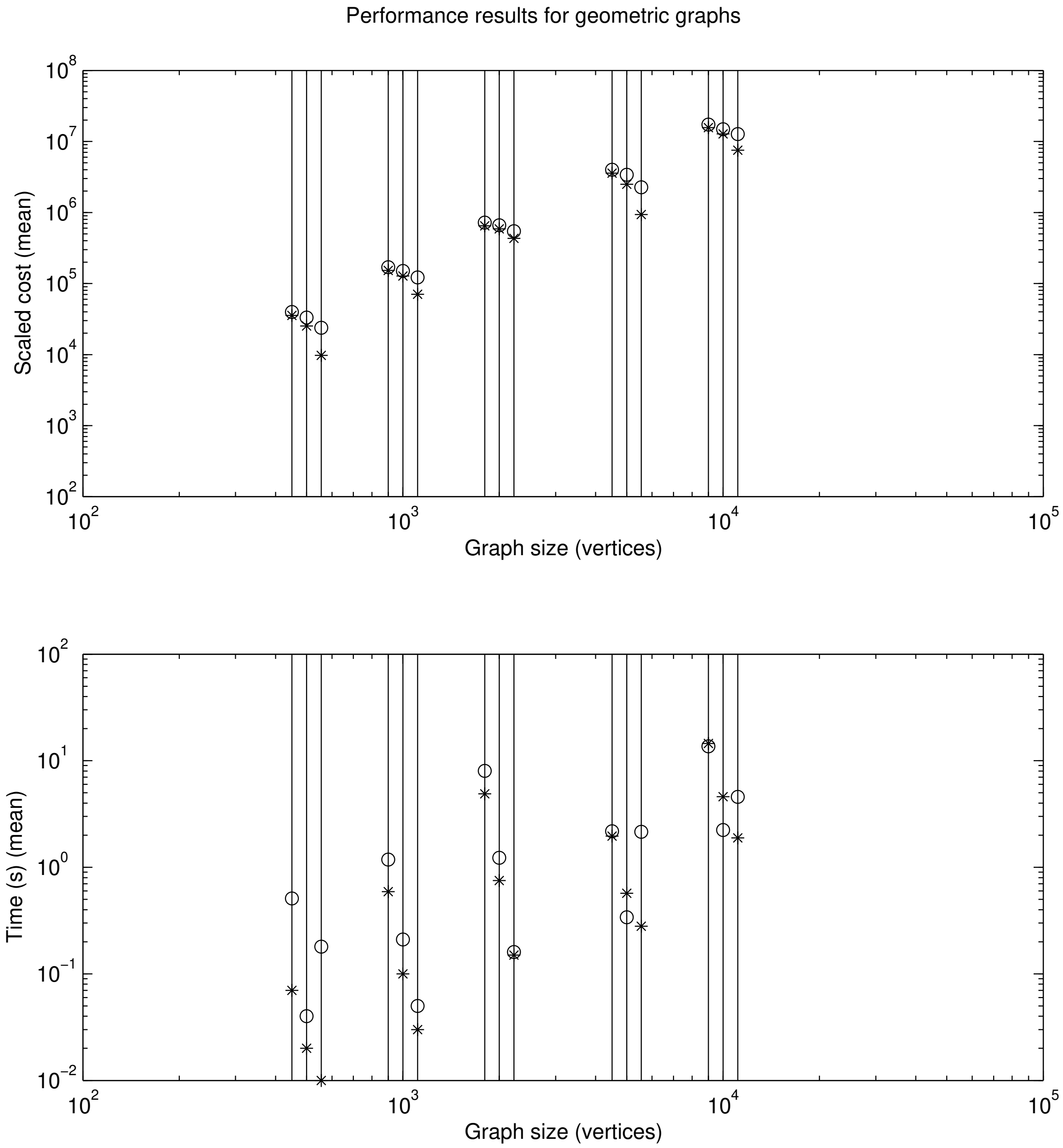

Figure 3.19: Performance results for random geometric graphs. Results for MCL are denoted by circles and results for RNSC (mean, 20 runs) are denoted by asterisks, and results for graphs of the same size are shown grouped in descending order of density. The precision of all measurements is 10^{-2}.

Chapter 4

Conclusions and Future Work

We have presented the Restricted Neighbourhood Search Clustering Algorithm, a powerful and efficient method for clustering graphs. We use simple principles of stochastic local search algorithms and two separate performance criteria, naive and scaled, to ensure good performance. In terms of our primary criterion, the scaled cost function, RNSC outperforms the MCL algorithm and runs faster in most cases.

However, RNSC has a couple of drawbacks, both of which MCL, in contrast, is capable of handling. The first shortcoming of RNSC is its inability to generate clusterings of varying granularity. The second is its inability to handle weighted graphs. It seems as though both of these problems can be dealt with easily.

What affects the granularity of a clustering, in our case, is the relative cost incurred by missing edges within clusters versus present edges between clusters. If we make cross-edges very costly, the clusters will tend to be larger to the point at which the problem approaches that of finding small edge cuts in the graph. If we make cross-edges very cheap compared to missing edges within clusters, the clusters will tend to be dense and small. While this kind of scaling may be troublesome in the naive cost function, it should be very easy in the scaled cost function.

To handle weighted graphs, we must affect the obvious changes necessary in the cost function. This should be very easy but adds the difficulty that we must consider all edges separately. Again, this will likely affect the performance of the naive cost function more than that of the scaled cost function.

Appendix A

Glossary of Notation

Graph Data (for graph G)

V	Vertex set
E	Edge set
n	Order $= \|V\|$
m	Edge order $= \|E\|$
$\mathcal{C}[G]$	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 a 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 $\mathcal{C}$
$\#^0_{\mathrm{in}}(G, \mathcal{C}, v)$	Number of good edges missing from v
$\#^1_{\mathrm{out}}(G, \mathcal{C}, v)$	Number of bad edges incident to v
$N_i(v)$	Open neighbourhood in C_i of a vertex 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
$\mathcal{N}_C$	Maximum 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] G. Bader and C. Hogue. An autormated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(2), 2003.

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

[3] 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.

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

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

[6] 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.

[7] A. D. King. Graph clustering with restricted neighbourhood search. Master's thesis, University of Toronto, 2004.

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

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

[10] 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.

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

[12] 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.

[13] N. Pržulj, D. G. Corneil, and I. Jurisica. Modeling interactome: Scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

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

[15] 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.

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

[17] S. M. van Dongen. Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, Centrum voor Wiskunde en Informatica, May 2000.

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

[19] 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.