Multiobjective Graph Clustering with Variable Neighbourhood Descent

by

Igor Naverniouk

B.Sc., The University of British Columbia, 2003

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

Computer Science

THE UNIVERSITY OF BRITISH COLUMBIA

April 21, 2005

Abstract

Graph clustering is a well-known combinatorial problem that appears in many different incarnations. The task is to partition the vertex set of a graph in order to minimize a given cost function. Clustering has applications in VLSI design, protein-protein interaction networks, data mining and many other areas. In the context of multiobjective optimization, we have more than one cost function, and instead of finding a single optimal solution, we are interested in a set of Pareto-optimal solutions. We present a multiobjective variable neighbourhood descent algorithm for this problem and its results on a collection of synthetic and real world data. On data sets that have a known "correct" clustering, our algorithm consistently finds interesting unsupported solutions (those that can not be found by any linear single-objective restriction of the problem), demonstrating a clear advantage of the multiobjective approach. Additionally, the shape of the Pareto front generated by the algorithm can give clues for the areas of the cost function space that contain non-trivial solutions. We compare our method to a single-objective clustering algorithm (RNSC, [41]) and a multiobjective algorithm (MOCK, [27]). On all data sets, our algorithm requires substantially longer CPU time, but produces higher quality results.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Holger Hoos, for invaluable suggestions, ideas, advice and support. Thank you to Andrew King for the RNSC source code, to Julia Handl for the MOCK source code and to Raymond Ng, my second reader.

Chapter 1

Introduction

In this thesis, we present a new algorithm for multiobjective graph clustering. The algorithm is a variant of Multiobjective Variable Neighbourood Descent and generates, for a given graph, a set of clusterings that are Pareto-optimal with respect to a set of given cost functions.

1.1 Clustering

A *graph* is a pair $G = (V, E)$, where V is a finite set of *vertices* and $E \subseteq V^2$ is a set of *edges*. In an *undirected* graph, E is symmetric, *i.e.*if $(u, v) \in E$, then $(v, u) \in E$. In this case, the edge and its *dual* are usually referred to as a single edge. *Weighted* graphs associate a number (weight) with each edge. By convention, n denotes $|V|$ and m denotes $|E|$. A *complete* (undirected) graph is a graph that obeys $m = n^2$.

A (crisp) *clustering* of a graph is a partition $C = \{C_0, C_1, \ldots, C_k\}$ of V, *i.e.*

$$\forall i, j : C_i \cap C_j = \emptyset,$$

$$C_0 \cup C_1 \cup \ldots \cup C_k = V.$$

A cost function f assigns a real number to any given clustering of G. The (Single-Objective) Clustering problem is then to find a clustering that minimizes a given cost function on a given graph. An example of a cost function is the negative sum of edge weights between clusters. Other cost functions are discussed in Section 2.2.1. The elements being clustered are sometimes referred to as *nodes*, *points*, *sites* or *vertices*.

Clustering is usually defined in terms of weighted, undirected graphs, where weights correspond to either similarity scores or distances. Unweighted graphs can be viewed as a special case of weighted graphs, where each edge has a weight of one. Much of the literature on clustering deals with sets of k-dimensional vectors. Such a set can be viewed as a complete, weighted graph, where each vector is a vertex, and the weight of each edge is the distance between the two vectors it connects. Distance is usually Euclidean, but other choices are possible (*e.g.*correlation).

Applications of clustering range from multiple sequence alignment [10], to gene expression [2], to galaxy formation [56]. Clustering is one of the most widely used tools in data mining [38, 58] and natural language processing [53, 48]. Other applications include image registration, protein-protein interaction networks and VLSI circuit design. A Google Scholar search for “clustering” brings up 379,000 hits. There are many formalizations of the Clustering problem, most of which are NP-hard or even hard to approximate [20, 25, 11].

1.2 Multiobjective Optimization

Multiobjective Optimization Problems are minimization problems that work with a vector-valued cost function, F. The goal is to find the set S of Pareto-optimal solutions. Such a set has the following properties. For every solution $x \notin S$, there exists a solution $\hat{x} \in S$, such that no component of $F(\hat{x})$ is larger than the corresponding component of $F(x)$, and at least one component is smaller. Furthermore, S is the smallest such set.

There are both theoretical and practical reasons for considering the Clustering problem in a multiobjetive setting. In practice, there is a multitude of literature that uses clustering in one form or another, and each author has her own definition of a cost function and a "good" clustering (see Section 2.2.1). From the theoretical point of view, Kleinberg [42] proved that there is no single-objective clustering algorithm that works with an arbitrary cost function and produces results that have three very reasonable properties (scale-invariance, richness and consistency).

Definition 1.2.1. *The* Multiobjective Clustering Problem *is to compute the Pareto-optimal set S of clusterings (partitions of V) for a given graph $G = (V, E)$ with respect to a given cost function $F = (f_1, f_2, \ldots, f_k)$. This means that for each clustering $s \notin S$, there is a corresponding $\hat{s} \in S$ such that*

$$\forall i f_i(\hat{s}) \leq f_i(s),$$

$$\exists j f_j(\hat{s}) < f_j(s).$$

Furthermore, S is the smallest such set.

The multiobjective version of the Clustering problem is a generalization of the single-objective case and is, therefore, even more difficult. Most people are interested in computing approximations to the Pareto-optimal set, S. The algorithm presented in this thesis uses Stochastic Local Search to compute such an approximation.

1.3 Stochastic Local Search

Stochastic Local Search (SLS) is a class of algorithms (or meta-heuristics) that use randomness to search the set of all possible solutions to a combinatorial problem. The majority of effort in SLS research is spent on coming up with clever strategies for preventing such algorithms from getting stuck in local minima. A defining characteristic of SLS algorithms is their focus on making quick, local updates to an existing approximate solution in order to generate a better approximate solution. An update involves generating a neighbouring solution and modifying the cost function value(s) appropriately. Often, recomputing the value of a cost function for each new solution is too costly, and a better approach is to compute the change in value that is associated with the corresponding local update.

Most of these meta-heuristics maintain a set of approximate solutions (often, just one solution). At each step, we pick a solution from the set and locally modify it to obtain another valid approximate solution. Then, according to some criterion, we

decide whether to keep this new solution or discard it. This decision can be based on the solutions' cost function values, previous search history and/or randomness. Eventually, a termination condition causes the algorithm to stop and produce a result.

Different independent runs of a stochastic algorithm may produce different results. This is why these algorithms are usually executed multiple times, which also allows for a more robust performance analysis and comparison of results.

SLS algorithms are a popular approach to solving a variety of NP-hard problems. The best known algorithms for Travelling Salesman, 3-Satisfiability, Constraint Satisfaction and many other problems are SLS algorithms [1, 35]. Examples of SLS include Iterated Local Search, Dynamic Local Search, Simulated Annealing, Evolutionary Algorithms, Ant Colony Optimization, Memetic Algorithm and more. For some NP-hard problems, SLS provides the only known method of obtaining non-trivial results. There is a variety of SLS algorithms for Single-Objective Clustering [37, 23, 14, 39, 41].

The algorithm presented in this thesis is based on Variable Neighbourhood Descent. VND takes an approximate solution and modifies it, creating a sequence of successively better solutions. Eventually, it reaches a local minimum and uses a different search neighbourhood (local changes of a different sort) to find solutions that are not locally minimal. Having escaped the local minimum, it then continues as before, alternating between the neighbourhoods.

1.4 Ouline of Thesis

Chapter 2 describes previous work on single-objective clustering (Section 2.1), cost functions that have been used in the literature on single- and multi-objective clustering (Section 2.2) and related work on multiobjective optimization problems (Section 2.3).

In Chapter 3, we describe our algorithm (Multiobjective Variable Neighbourhood Descent). Section 3.1 presents a high-level overview of the most important procedures. Section 3.2 gives the implementation details for the non-trivial datastructures used.

In Chapter 4, we present our experimental results. Section 4.1 describes the F-measure – a mechanism for assigning a score to a clustering, given that the "correct" clustering for the data set is known. Section 4.2 presents our results for randomly generated unweighted graphs. We compare our algorithm to a state-of-the-art single objective algorithm (RNSC). Section 4.3 shows our results on a randomly generated vector set and a real-world data set, both with known expected clusterings. We compare our results to an evolutionary multiobjective algorithm (MOCK).

Chapter 5 contains a discussion of the results (Section 5.1), our conclusions and a number of improvements that can be made to our algorithm (Section 5.2).

Chapter 2

Related Work

The algorithm we present computes an approximate solution to the multiobjective clustering problem. This chapter describers some of the existing approaches for doing clustering, previous work on multiobjective optimization in general and related applications of stochastic local search.

2.1 Single-Objective Clustering

There are several variants of the clustering problem. *(Crisp) graph clustering* is the problem of partitioning the vertex set of a given graph in a way that minimizes a certain cost function. The cost function assigns a value to every partition (clustering) of the graph. Section 2.2 lists several cost functions that are used in the literature. When each vertex in the graph is a point in n-dimensional space, then clustering is sometimes referred to as *vector quantization*. Non-crisp clustering allows for overlapping clusters and unclustered vertices. The Maximum Clique problem can be considered a special case of non-crisp clustering.

2.1.1 Direct Methods

By "direct" methods, we mainly mean deterministic algorithms and randomized greedy algorithms (*e.g.*, k-means clustering [26]). Most classic methods for clustering come in one of two varieties – *agglomerative clustering* and *partitive clustering*.

Agglomerative methods initialize each element to belong to its own cluster. They then proceed to merge clusters until a certain terminating condition is met. Examples of the use of agglomerative clustering algorithms can be found in [3, 57, 44].

Partitive clustering starts with a single cluster containing all elements and proceeds by splitting clusters until a terminating condition is satisfied. Examples of the use of partitive clustering algorithms are [54, 52, 40].

A different single-objective clustering algorithm is k-means clustering [26]. It works for datasets that consist of points in n-dimensional space. This algorithm requires k as an input parameter and generates a clustering with k clusters in it. The most basic k-means implementation starts with a random k-clustering and improves it in a sequence of modifying iterations. In each iteration, we first compute the centroids of each existing cluster. We then assign each data point to the cluster defined by the centroid that is closest to that data point (in terms of Euclidean distance). The algorithm terminates when no change has been made in the last iteration. The k-means

algorithm essentially performs hill-climbing – the simplest example of stochastic local search.

The k-means algorithm is the best-known member of the class of representation-based algorithms, where each cluster is represented by a data point – either a member of the original data set or a new point constructed during the execution of the algorithm (*e.g.*, the means in k-means). Other representation-based algorithms include Fuzzy-c-means [6] and Expectation Maximization [13].

Sultan *et al.*[52] combine k-means clustering with the partitive clustering framework and Kohonen's self-organizing maps [43] to produce an algorithm for point (or vector) clustering and visualization called *Binary Tree-Structured Vector Quantization* (BTSVQ). The algorithm starts with a single cluster containing every data point and recursively splits it into two using k-means clustering with $k = 2$. Recursion terminates once a variance condition on the child clusters is satisfied. The result is a binary tree that has potential clusters at the leaf nodes. The next stage uses self-organizing maps to visualize the contents of each node (subtree). A human observer can then decide which subtrees correspond to true clusters.

2.1.2 Stochastic Local Search

Stochastic Local Search (SLS) is a class of algorithms that make use of randomized choices in generating or selecting candidate solutions for a given combinatorial problem instance [35]. This section describes some of the SLS approaches to clustering.

King [41] describes a single-objective *Restricted Neighbourhood Search Clustering* (RNSC) algorithm that uses two different cost function in succession. The algorithm is based on *tabu search* [24]. The first function (naive cost) is fast to compute and is used to quickly generate a near-optimal clustering. In the next stage, a slower, but more accurate function (scaled cost) is used to converge to a more optimal solution. The cost functions are described in detail in Section 2.2.1. King's algorithm uses a search neighbourhood consisting of moves, each of which takes a vertex to a different cluster. A tabu list stores the last few moves and prohibits returning to a previous configuration. After seeing a number of non-improving moves, the algorithm runs a diversification step that splits and recombines clusters in order to escape from a local minimum.

Handl and Knowles [27] use two competing cost functions for clustering (see Section 2.2.1) and apply a multiobjective evolutionary algorithm [34, 50]. The algorithm maintains a population of non-dominated solutions and creates new solutions by using a single mutation operator and no crossover. The mutation operator is applied to each gene with equal probability and results in the move of a vertex and its g nearest neighbours to a different cluster. g is an external parameter. The algorithm was tested on a number of synthetic and real life datasets and produced better clusterings than the same algorithm used to optimize just one objective function. Moreover, the multiobjective algorithm was found to be more robust than k-means clustering and average-link agglomerative clustering.

2.2 Cost Functions for Clustering

A big obstacle in clustering is the fact that the problem is poorly defined. The optimal clustering entirely depends on the choice of the cost function used to evaluate the goodness of a clustering, and the choice of a cost function is to some extent subjective and depends on the particular clustering goal.

There are two competing criteria that define a "good" clustering – high intra-cluster homogeneity and low inter-cluster connectivity. Intuitively, if graph edges represent relationships between vertices, then we want many edges within clusters and few edges between clusters. However, if we define the cost function to be the number of inter-cluster edges, then the problem of minimizing it is trivial – pick the clustering that contains a single cluster. When defining a cost function, it is easy to fall into this trap and get a trivial minimizer.

This problem can be largely avoided by fixing the number of clusters, as is done in the k-means [26] and k-modes [36] clustering algorithms. If the number of clusters is fixed, then one only needs to define a pairwise dissimilarity metric on the data points and make the cost function be the sum of the dissimilarities between pairs of points that share a cluster. In a review paper on vector clustering, Estivill-Castro [19] describes several popular cost functions (maximum likelihood, least squared error, least absolute error), all of which rely heavily on the assumption of either a fixed number of clusters or a particular random distribution of the data points.

On the other hand, Estivill-Castro argues that, "*propenents of [clustering] methods [often] leave undefined what are 'good clusters'. They delegate this responsibility to the user by making the algorithms depend on arguments supplied by the user (so called parameters of the algorithm). As a consequence, the results vary widely with changes to these user-supplied arguments.*" [19]. This is certainly an indication of how difficult it is to define a robust cost function.

2.2.1 Cost Functions Used in the Literature

As was noted in the previous section, fixing the number of clusters greatly simplifies the task of defining a cost function. Only some measure of inter-cluster distance or intra-cluster similarity is required. However, if the number of clusters is unknown and allowed to vary, the task becomes more difficult.

Edachery *et al.*[17] recast the clustering problem as the problem of finding the minimum number of distance-k cliques that cover the graph, where a distance-k clique is a subgraph of diameter k. In this form, the cost function is simply the number of clusters (distance-k cliques). Unfortunately, the algorithm requires the clique diameter k as a parameter, which is, arguably, as difficult for a human to estimate from the input data as the number of clusters is.

Handl and Knowles [27] apply a multiobjective evolutionary algorithm (MOCK) to the problem of clustering a set of d-dimensional vectors, where edge weights are distances between the vectors (points). They use two competing cost functions – *deviation*

(D) and *connectivity* (E) defined for a given clustering P as

$$centroid(C) = \frac{1}{|C|}\sum_{v \in C} v,$$

$$D := \sum_{C \in P}\sum_{u \in C} dist(u, centroid(C)),$$

$$E := -\frac{1}{|V|}\sum_{u \in V}\frac{1}{h}\sum_{j=1}^{h} neigh(u, nn_u(j)),$$

where $dist(u, v)$ is the weight of the edge (u, v), h is a user-defined parameter and $nn_u(j)$ is the j'th nearest neighbour of u according to the $dist()$ function. $neigh(u, v)$ is 1 if u and v are in the same cluster and 0 otherwise. MOCK only works with vector sets, not with general weighted graphs.

The two distance functions used in [27] are Euclidean distance and Cosine similarity of the normalized data. To compute the Cosine similarity, the data were first normalized to have a mean of 0 and a standard deviation of 1 in each dimension. Then for any two vectors u and v,

$$CS(u, v) = \frac{u \cdot v}{\|u\|\|v\|}.$$

King [41] solves the clustering problem on unweighted, undirected graphs $G = (U, V)$ using a stochastic local search algorithm. There are two cost functions used: the naive cost function f and the scaled cost function g. In any given clustering, let $c(v)$ denote the unique identifier of the cluster to which v is assigned. Then for any clustering C, $f(C) = f_1(C) + f_2(C)$, where

$$f_1(C) = \sum_{(u,v) \in E} \mathbb{I}(c(u) \neq c(v)),$$

$$f_2(C) = \sum_{(u,v) \in V \times V} \mathbb{I}(c(u) = c(v))\mathbb{I}((u, v) \notin E).$$

Here, $\mathbb{I}(P)$ is the indicator function that is equal to 1 when P is true and 0 when P is false. f_1 counts the number of edges between different clusters, and f_2 counts the number of edges that are missing inside each cluster. Ideally, a clustering would be a set of disconnected cliques, in which case both f_1 and f_2 would be zero. The fact that f_1 and f_2 are added together to define the naive clustering cost function is somewhat subjective and will be addressed in Section 2.3.

The second cost function used in [41] is $g(C) = g_1(C) + g_2(C)$, where

$$g_1(C) = \sum_{u \in V}\sum_{(u,v) \in E} \frac{\mathbb{I}(c(u) \neq c(v))}{|C_u \cup N(u)|},$$

$$g_2(C) = \sum_{u \in V}\sum_{v \in C_u} \frac{\mathbb{I}((u, v) \notin E)}{|C_u \cup N(u)|}.$$

Here, $C_u \subseteq V$ is the cluster containing u, and $N(u) = \{v \in V : (u, v) \in E\}$ is the set of neighbours of u in G. The scaled costs are less sensitive to large clusters in sparse graphs than the naive cost functions, but they require more computational resources to maintain [41]. Note that g_2 does not penalize a large, space cluster as much as f_2 does. Once again, it is unclear that simply adding g_1 and g_2 is the best way of combining these two functions into a single clustering cost function, and this issue will be addressed in Section 2.3.

Note that both f_1 and g_1 measure inter-cluster connectivity (which should be minimized in an optimal clustering), while f_2 and g_2 measure intra-cluster difference (which should also be minimized). Both f and g have the advantage of not requiring the number of clusters to be fixed. Together, they do not favour trivial clusterings where all nodes are assigned to the same cluster, or each node is assigned to its own cluster. Note also that none of f_1, f_2, g_1 or g_2 separately have this important property.

Both the naive and scaled costs are constant multiples of those introduced by van Dongen [15, 16], who also describes criteria for evaluating the quality of clusterings on a weighted graph. Van Dongen's idea is to consider edge weights to be proportional to transition probabilities between vertices. He views the graph as a flow network and proposes a clustering algorithm that simulates this flow. The key observation is that there should be little flow between clusters and much flow within clusters. Note that this use of edge weights as similarity measures is opposite to the usual notion of edge "length".

Given a row u of the adjacency matrix, let $\pi = cu$, where c is a scalar such that $\sum \pi_i = 1$. Then the *mass centre* (of order 2) of u is defined as

$$ctr(u) = \sum_{i=1}^{n} \pi_i^2.$$

Given any clustering where the vertex u is assigned to the cluster C_u, van Dongen then defines the weighted coverage measure associated with u as

$$Cov(u) = 1 - \frac{|C_u| - \frac{1}{ctr(u)}(\sum_{v \in C_u} \pi_v - \sum_{v \notin C_u} \pi_v)}{n}. \tag{2.1}$$

The total weighted coverage measure for a clustering is $\sum_{u \in V} Cov(u)$.

He then argues that the quantity $1/ctr(u)$ is the size of the "ideal" clustering for u. This is easily seen in the case where each transition vector u is homogeneous (including an equal probability of a self-loop) because this case is similar to an unweighted graph.

Note once again the two sums in the numerator, one measuring intra-cluster connectivity and the other – inter-cluster separation. The constants 1 and n are only there to ensure that $0 \leq Cov(u) \leq n$ for all u. They can be dropped if we are using $Cov(u)$ as a cost function in a clustering algorithm.

2.3 Multiobjective Optimization

Multiobjective Optimization is a well studied class of problems (see Section 1.2 for a definition). Clustering is a naturally multiobjective problem because a "good" clustering maximizes both inter-cluster separation and intra-cluster connectivity [15, 16, 41].

There are many different clustering algorithms, each one with its own cost function and its own preferred definition of an acceptable cluster [19]. This led to an approach of treating the different algorithms as multiple cost functions and combining them into a single multiobjective clustering framework of Law *et al.*[45].

The following definitions are often used when discussing Multiobjective Optimizaition Problems (MOP's).

Definition 2.3.1. *Given two n-dimensional vectors u and v, $u \leq v$ is defined to be true iff $\forall i \in \{1, \ldots, n\} u_i \leq v_i$ and $\exists j \in \{1, \ldots, n\} u_j < v_j$.*

Definition 2.3.2. *A solution x is* efficient *(also* non-dominated *or* Pareto-optimal*) in the feasible set X with respect to the (vector-valued) cost function f, iff there is no $x' \in X$ such that $f(x') \leq f(x)$.*

Note that the $\leq$ relation is irreflexive, which might seem counter-intuitive when trying to associate it with its usual meaning of "less than or equal to". For the special case of 1-dimensional vectors, this relation reduces to $<$. However, this notation is standard.

Most of the MOP's studied in the literature are multiobjective extensions of single-objective combinatorial optimization problems (*e.g.*, traveling salesman, knapsack, set covering, scheduling and minimum spanning tree) [18]. Most of these are NP-hard even in the single-objective case; thus, the majority of recent literature on multiobjective optimization focuses on heuristics and stochastic local search (SLS) methods [18].

2.3.1 Single-objective Restrictions

Some of the heuristic methods for MOP's are based on taking weighted sums of the cost function components and reducing the problem to a set of single-objective cases. However, in general, multiobjective optimization problems can have *unsupported* solutions.

Definition 2.3.3. *A solution $\hat{x}$ to a MOP with a p-dimensional cost function, f, is* supported *if there exists a p-dimensional weight vector λ such that $\hat{x}$ is an optimal solution of the corresponding single-objective problem with the cost function*

$$g(x) = \sum_{k=1}^{p} \lambda_k f_k(x).$$

Otherwise, $\hat{x}$ is called unsupported.

The existence of unsupported solutions is easy to demonstrate in the case of two cost functions. Consider the following geometric interpretation of the set of efficient solutions. Associate the x- and y-axes with the two cost functions and plot the set E of efficient solutions as a set of points in the xy-plane. Then the fact that no member of E is dominated implies that for each plotted point $p = (p_x, p_y)$, the quadrant $[p_x, \infty) \times [p_y, \infty)$ is empty of points.

Take any two points, P and Q and draw a line through them. In general, it is possible to have another point R be in the bounding box of P and Q and above the line. In this case, R would correspond to an unsupported solution because no linear combination $\alpha f + \beta g$ of the two cost functions would be minimized at R. Of course, we have to assume that both α and β are non-negative, which is true for any reasonable cost function since we are talking about a minimization problem [18].

2.3.2 Multiobjective Stochastic Local Search

Because of unsupported solutions, solving any number of single-objective instances of a problem is not necessarily sufficient for finding all of the efficient solutions to the multiobjective problem. Another approach involves adapting existing stochastic local search (SLS) algorithms to handle the multiobjective case.

In a survey paper, Ehrgott and Gandibleux [18] describe a multitude of such algorithms including multiobjective versions of genetic algorithms, simulated annealing, tabu search, ant colony optimization, artificial neural networks, greedy randomized adaptive search and others. Most of these algorithms rely on maintaining a set of potentially efficient solutions (PE-set or *Pareto front*). This set is usually initialized at the beginning of the algorithm using random solutions or solutions generated by single-objective optimization. During the course of the algorithm, new solutions are added to the set, and dominated solutions are removed from it.

After the algorithm terminates, the PE-set represents the algorithm's best guess at the true set of efficient solutions. Just like in single-objective SLS, all such algorithms may get stuck in local minima and produce a sub-optimal set of solutions in the end. It is also worth noting that any multiobjective local search algorithm is most likely doomed to be slower than its single-objective version because it has to maintain a set of solutions at any time, while most single-objective algorithms only maintain and update a single solution.

One of the most popular classes of multiobjective algorithms is the class of Multiobjective Evolutionary Algorithms (MOEA). Evolutionary algorithms seem naturally suited for multiobjective optimization because they maintain a population of solutions [9]. Handl and Knowles' MOCK [27, 28] is an example of a MOEA for clustering.

The algorithm for multiobjective clustering that is described in this thesis is a multiobjective version of Variable Neighbourhood Descent (VND). Single-objective VND [35] is a popular SLS algorithm that has been applied to a variety of problems, such as Vehicle Routing [12], Undirected Capacitated Arc Routing [33] and Scheduling [5].

Chapter 3

Algorithm

This section describes our algorithm for the multiobjective clustering problem. Section 3.1 presents a high-level view of the most important routines. Section 3.2 talks about the data structures and important implementation details.

The pseudocode style used in this section is that of Cormen *et al.*[8]. The triangle ($\triangleright$) is used to denote comments. The left arrow ($\leftarrow$) denotes variable assignment.

3.1 Overview

Our algorithm is an application of Multiobjective Variable Neighbourhood Descent. It maintains a set of potentially efficient (non-dominated) solutions (PE-set) and manipulates this set by performing a VND run on a randomly chosen element of the PE-set to the PE-set. After each insertion, all the dominated solutions are removed from the set. When the algorithm terminates, the output is the contents of the PE-set.

3.1.1 Variable Neighbourhood Descent

At the core, our algorithm is a variant of Variable Neighbourhood Descent (VND) [35] in a multiobjective framework. The algorithm maintains two data structures – the graph (represented as an adjacency list and matrix) and the PE-set (a set of solutions). Each solution is a clustering with an associated vector of cost function values.

Top level algorithm

The PE-set is initialized to contain at least one solution. This is done with either a greedy or a random clustering. Section 3.1.3 discusses initialization. Then a number of VND runs are performed on randomly picked members of the PE-set. The members are picked uniformly at random. Each VND run attempts to add all of the solutions it visits to the PE-set. Some might be rejected because they are dominated by existing members of the set. This process continues until the termination condition described below is met.

Maintaining a set of non-dominated solutions

Addition of solutions to the PE-set works in the following way:

PE-INSERT(x)

```
1  for each y ∈ PE
2      do if y ≤ x
3          then return false
   ▷ Insert y into the set
4  for each y ∈ PE
5      do if x ≤ y
6          then ▷ Remove y from PE
7  return true
```

The $\leq$ relation on the solutions is introduced in Definition 2.3.1. More efficient ways of implementing PE-set insertion are discussed in the Future Work section.

Solution Neighbourhoods

A *neighbourhood* in SLS is a function that maps a solution s to a set of solutions that can be obtained from s by performing a local modification [35]. Our algorithm uses three different neighbourhoods, N_1, N_2 and N_3.

The algorithm relies on an efficient implementation of moves in the neighbourhood N_1. This includes both creating a modified clustering and updating the cost functions. Therefore, we chose N_1 to be very simple. It is the neighbourhood obtained by moving a vertex, v from its cluster to a neighbouring cluster (a cluster that contains at least one vertex connected to v by an edge). King [41] uses the same neighbourhood in his single-objective tabu-based clustering algorithm. Handl and Knowles [27] use a slightly more complicated one, where several of v's nearest neighbours are also moved to the same cluster.

The neighbourhoods N_2 and N_3 are a mechanism for escaping from local minima. We chose N_2 to be the neighbourhood obtained by splitting an existing cluster along its diameter. This is done by running the Floyd-Warshall [22, 55, 8] algorithm on the cluster to compute the diameter. Then we find two endpoint vertices of a diameter, du and dv. If there is a tie, we pick the largest numbered du. If there is still a tie, we pick the largest numbered dv (these are arbitrary choices). Finally, we create a new cluster and move to it all the vertices that are closer to dv than to du in the sense of the shortest path. Vertices for which du and dv are equidistant are moved with probability 0.5.

N_3 is obtained by joining together two existing clusters. N_1 and N_2 moves are more expensive to compute than the N_1 moves and cause larger structural changes to the clustering. The three neighbourhoods are used during a VND run.

Performing a VND run

At each step of the multiobjective VND algorithm, the algorithm picks a random solution s from the PE-set and performs the following VND run on it.

```
VND(s)
1  A: while TRUE
2       do PE-INSERT(s)
3          for i ← 1 to 3
4              do for each t ∈ N_i(s)
5                     do if t ≤ s
6                           then s ← t
7                                continue A
8          break
```

N_1, N_2 and N_3 are the three solution neighbourhoods described above. The VND procedure simply performs iterative first improvement (or hill climbing) in the first neighbourhood, N_1. Once a local minimum (*w.r.t.*N_1) is reached, VND switches to the neighbourhood N_2. When no improvement can be made there either, it tries N_3. If a better solution t is found in either N_2 or N_3 (one that dominates s), then we substitute t for s and start again in the neighbourhood N_1.

This approach of using three neighbourhoods is a standard application of Variable Neighbourhood Descent [47, 32, 35]. On the other hand, it is not VND in the strict sense because decision whether any given move is an improvement or not depends on the whole PE-set, not only on the solution whose neighbourhood we are considering. VND is closely related to Iterated Local Search (ILS). In fact, if we allowed worsening moves in neighbourhoods N_2 and N_3, the algorithm could be considered an ILS. See the Future Work section for a further discussion about worsening moves.

The Termination Criterion

The algorithm ensures that the set of potentially efficient solutions that is eventually returned when it terminates is a true local optimum with respect to all 3 neighbourhoods. This means that if $s \in PE$ and $t \in N_1(s) \cup N_2(s) \cup N_3(s)$, then t is dominated by some member of PE ($\exists r \in PE r \leq t$). This is guaranteed by the algorithm because this condition is the termination criterion and is explicitly checked in the following way.

Each step of the algorithm involves picking a random member of the PE-set (uniformly at random) and performing a VND run on it. If this run results in no changes made to the PE-set (no new non-dominated solutions found), then the run is considered unsuccessful. If we make a certain number, T, (see below) of unsuccessful runs in a row, then instead of picking out random members of the PE-set, the algorithm will switch to the less efficient strategy of scanning the whole PE-set and performing VND runs on each of the solutions. If at least one of them is successful, we revert back to the random strategy. Otherwise, we have checked the neighbourhoods of each of the elements of the PE-set, and a local optimum is found. At this point, the algorithm terminates.

This is the core of our multiobjective VND algorithm.

```
MOVND()
 1  INIT-PE()
 2  noChangeFor ← 0
 3  while noChangeFor < T
 4        do s ← PICK-RANDOM-ELEMENT(PE)
 5           if VND(s) is successful
 6             then noChangeFor ← 0
 7             else noChangeFor ← noChangeFor +1
 8           if noChangeFor = T
 9             then for each s ∈ PE
10                        do if VND(s) is successful
11                              then noChangeFor ← 0
12                                   break
```

The value of T is set to be half of the size of the PE-set. It is a parameter that affects the running time of the algorithm. An alternative, more efficient strategy for ensuring termination at a local minimum is discussed in the Future Work section. The INIT-PE procedure is described in Section 3.1.3.

3.1.2 Cost Functions

We tested our algorithm on two types of data – unweighted graphs and vector sets, both of which can be viewed as a special case of weighted graphs. Theoretically, our algorithm can work with an unlimited number of cost functions, but using more that two usually makes the PE-set unreasonably large and inefficient to work with. (See Section 3.1.3 for a technique aimed at keeping the PE-set small.)

Unweighted Graphs

For unweighted graphs, we used a pair of cost functions derived from King's naive cost [41], which was, in turn, taken from van Dongen [15]. The cost function is a sum of two quantities, one measuring inter-cluster connectivity and the other measuring intra-cluster separation. Here, $\mathbb{I}(P)$ is the indicator function that is equal to 1 when P is true and 0 when P is false.

$$f(C) = \sum_{(u,v)\in E} \mathbb{I}(c(u) \neq c(v)) + \sum_{(u,v)\in V\times V} \mathbb{I}(c(u) = c(v))\mathbb{I}((u,v) \notin E).$$

The first term is simply the number of edges connecting two clusters. The second term is the minimum number of edges that need to be added in order to make each cluster a clique. King [41] himself argues that it is unclear whether simply adding the two values is the best way of combining them into a single cost function. He suggests using a linear (or a convex) combination instead.

We have the option of avoiding this question altogether and separating the naive cost function into two functions, which is what our algorithm does. The two cost

functions used for unweighted graphs are $f1$ and $f2$, defined as:

$$f_1(C) = \sum_{(u,v)\in E} \mathbb{I}(c(u) \neq c(v)),$$

$$f_2(C) = \sum_{(u,v)\in V\times V} \mathbb{I}(c(u) = c(v))\mathbb{I}((u,v) \notin E).$$

We use three local search neighbourhoods. The first one involves moving a vertex v to a neighbouring cluster. "Neighbouring" here means a cluster to which v is connected by an edge. Updating f_1 and f_2 here means looking at the neighbours of v, which can be done in time proportional to the degree of v using the adjacency list. When v is moved from its cluster, C_v, to another cluster, D, the changes in f_1 and f_2 are

$$\Delta f_1(v, D) = \sum_{(u,v)\in E} \left(\mathbb{I}(u \in C_v) - \mathbb{I}(u \in D)\right),$$

$$\Delta f_2(v, D) = |D| - |C_v| + 1 + \sum_{(u,v)\in E} \left(\mathbb{I}(u \in C_v) - \mathbb{I}(u \in D)\right),$$

Moves in the other two neighbourhoods (merging two clusters and splitting a cluster along its diameter) are not as frequent as the moves in N_1 because VND gives priority to N_1 moves. During the moves in N_2 and N_3, the two cost functions are simply recomputed. This is sub-optimal because it is possible to compute the local change in cost functions faster than recomputing all of f_1 and f_2. However, the gain in performance would be negligible because the vast majority of local improvement moves take place in the first neighbourhood. (Over 85% of moves were made in N_1 during a test run on the synthetic data set Square1(64).)

Vector Sets

For vector sets, the naive cost functions are inappropriate for two reasons. First of all, the naive cost functions only work for unweighted graphs, and it would be a waste of information to disregard distances between data points or use a user-supplied threshold. Secondly, without thresholding, a vector set is best modelled by a complete graph. This means that there is only one optimal clustering – that one consisting of a single cluster. It is a trivial solution in this case.

Fortunately, van Dongen [15] describes a generalization of the naive cost function to weighted graphs. See Section 2.2.1 for a description of his weighted coverage measure (Equation (2.1)).

First of all, the constants 1 and n are irrelevant to the minimization problem. Furthermore, the numerator can be split into two competing criteria:

$$h_1(C) = \sum_{u\in V} \left(|C_u| - \frac{\sum_{v\in C_u} \pi_v}{ctr(u)} \right),$$

$$h_2(C) = \sum_{u\in V} \left(\frac{1}{ctr(u)} \sum_{v\notin C_u} \pi_v \right).$$

Note that in the case when all edges have the same weight, this definition of h_1 and h_2 reduces to the naive functions f_1 and f_2. After simplifying the sums, we get

$$h_1(C) = \sum_{u \in V} |C_u| + \sum_{(u,v) \in E} \mathbb{I}(C_u = C_v)\Pi_{uv},$$

$$h_2(C) = \sum_{(u,v) \in E} \mathbb{I}(C_u \neq C_v)\Pi_{uv},$$

where

$$\Pi_{uv} = \frac{\pi_{uv}}{ctr(u)} \frac{\pi_{vu}}{ctr(v)}.$$

It is important to note that van Dongen's coverage measure works only if the edge weights correspond to transition probabilities. In other words, a higher weight of the edge (u, v) corresponds to a higher probability of u and v being in the same cluster. The usual definition of an edge weight is that of a distance between u and v. In the case of a vector set, edges correspond to Euclidean distances. To convert them to probabilities, we used both inverse distances and inverse square distances. See Results for a discussion.

Updating h_1 and h_2 just before a vertex v is moved from its cluster C_v to another cluster, D, is very similar to updating the naive costs.

$$\Delta h_1(v, D) = 2\left(|D| - |C_u| + 1\right) + \sum_{(u,v) \in E} \left(\mathbb{I}(C_u = C_v) - \mathbb{I}(C_u = D)\right) \Pi_{uv},$$

$$\Delta h_2(v, D) = \sum_{(u,v) \in E} \left(\mathbb{I}(C_u = C_v) - \mathbb{I}(C_u = D)\right) \Pi_{uv}.$$

The Π values can be pre-computed, and the updates performed in time linear in the number of vertices. For neighbourhoods N_2 and N_3, the function values are recomputed from scratch after each move.

3.1.3 Depth Runs

Ideally, we wouuld like to keep the size of the PE-set small because the algorithm's running time depends heavily on the time required to update the set. Therefore, it could be useful to generate an initial PE-set that contains a small number of fairly good solutions that are likely to dominate a large number of other solutions. This set should be both small (at most a few hundred solutions) and diverse (able to dominate a large portion of the cost function space).

We solve this problem by starting the algortihm with a sequence of "depth runs". Each depth run is a simple first improvement hill climbing run in the neighbourhood N_1 (moving a vertex to a neighbouring cluster). These depth runs are relatively fast and terminate at a local minimum. Each one is initialized with a random clustering and is performed independently of the other runs and the current PE-set.

The first depth run is always initialized to a greedy clustering. The greedy clustering is constructed by taking the smallest numbered vertex and building a cluster

consisting of it and its neighbours in the graph. Then we take the next smallest numbered vertex and its neighbours to build the second cluster, *etc.*. All subsequent depth runs start from random clusterings. The greedy clustering is often trivial (especially in the case of a complete graph); therefore, we never use fewer than 2 depth runs to avoid getting meaningless results.

Alternative approaches to doing depth runs include optimizing just one objective function or a linear combination of the objective functions using a single-objective algorithm. The former approach was used by Paquete and Stützle [49] in their two-phase algorithm for the biobjective travelling salesman problem. The latter method is discussed in more detail in the Future Work section.

3.1.4 Distributed Computation

The core of our multiobjective local search algorithm is the PE-set. Each step begins by taking a random element of the set. Then we perform certain operations on the element. Finally, we add one or more new elements to the set. This operation is highly parallelizable. As long as the PE-set is protected from concurrent modification, there is nothing stopping us from picking two or more solutions simultaneously and exploring their neighbourhoods in parallel.

The only drawback to this is that the PE-set itself becomes a bottleneck. A more scalable approach is to maintain multiple copies of the PE-set and periodically synchronize them by performing a union and subsequent elimination of dominant solutions. Our implementation is capable of spawning a number of worker processes, each one maintaining its own PE-set. Once every W seconds (wall clock time), each worker process connects to the server, sends its copy of the PE-set and receives the server's copy. Both the worker and the server then perform the union and dominant solution elimination.

The constant W is a parameter that depends on the number of worker processes and the expected size of the PE-set. For now, it is determined by the user, but ideally, it should be changed adaptively in order to be kept as small as possible without causing a bottleneck at the server. Low values of W are preferable because this way, the PE-set copies are kept as synchronized as possible between the worker threads, which reduces the chances of a process exploring the neighbourhood of a dominated solution.

We ran the distributed version of the algorithm on a Sun N1 Grid Engine 6 consisting of 20 dual x86 2GHz Linux machines. For most distributed experiments, we used 20 parallel worker processes, plus a central PE-set server responsible for collecting and redistributing solution sets. See Results for a more detailed description of the hardware.

3.2 Implementation and Data Structures

The algorithm was implemented using C++ (g++ 3.3 on SuSE Linux) and is portable to other flavours of UNIX. The code makes use of no external libraries other than the Standard Template Library. This section describes the data structures used for the non-trivial parts of the algorithm.

3.2.1 Input Data

Thee are two different types of input data – unweighted graphs and vector sets. An unweighted graph is stored in three different forms simultaneously: as an adjacency matrix, an adjacency list and an edge list. These structures require a fair amount of memory, but are nonetheless small compared to the size of the PE-set.

The adjacency matrix allows $O(1)$-time checks for whether an edge exists or not and can store the weight of an edge in a weighted graph (for future extensions to the algorithm).

The adjacency list allows $O(vertexDegree)$-time traversal of the neighbour set of a vertex. This is required when updating certain cost functions, for instance the naive cost of [41].

The edge list allows $O(|E|)$-time traversals of all edges. Technically, the same time complexity can be achieved by scanning the adjacency list of each vertex in order. However, this way is slightly more efficient and elegant. The cost in memory is not a big concern in comparison to the PE-set (see Section 3.2.2).

The second kind of input data is a set of n k-dimensional vectors (points). Such a set is modelled by a complete graph on n vertices. Each edge has a weight equal to the Euclidean distance between the corresponding pair of vectors. The cost functions used for this type of data are the van Dongen weighted functions, so we need to precompute the mass center of each vertes, as well as the π matrix. In fact, since the π values are always used in the form of Π_{uv} (see Section 3.1.2), we simply store the symmetric matrix Π, where

$$\Pi_{uv} = \frac{\pi_{uv}}{ctr(u)} \frac{\pi_{vu}}{ctr(v)}.$$

3.2.2 PE-set

The PE-set is a set of non-dominated solutions found so far. More formally, if $s \in PE$, then there exists no $t \in PE$ such that $t \leq s$, where the $\leq$ relation on the solutions (*i.e.*their cost function vectors) is defined in Definition 2.3.1.

A solution is represented by an ordered pair (C, f), where C is a clustering and f is a vector of cost function values. When comparing two solutions, s and t, one must be careful to deal with the case when the f-values of s and t are equal. In this case, if C-values are also equal, then the solutions are the same and one of them can be ignored. Otherwise, neither solution is dominated by the other one, and both need to be kept in the PE-set.

The data structure for the set itself is implemented in the simplest possible way – as an array of solutions. The only operation allowed on the PE-set is PE-INSERT(x). If x is dominated by an element of the PE-set, then PE-INSERT(x) does nothing. Otherwise, all the solutions dominated by x are removed from the set, and x itself is inserted. This is implemented by a simple linear scan through the array. In the vast majority of the cases, x is indeed dominated, and no modification needs to be done. Otherwise, another linear-time scan is performed, and all of the dominated solutions are removed.

In total, PE-INSERT(x) requires a number of solution comparisons that is linear in the size of the PE-set in the worst case. This is far from optimal. Roughly 80% of

the CPU time is spent in PE-INSERT(x). See the Future Work section for a way to improve this data structure so that PE-INSERT(x) runs in amortized logarithmic time.

Chapter 4

Experimental Results

We tested the algorithm on two different types of input data – vector sets and unweighted graphs. The former are data sets with known "correct" clusterings and were used to evaluate the quality of produced results. The latter are random graphs that are hard to cluster and were used to evaluate the algorithm's performance characteristics.

All parallel experiments were performed on a Sun N1 Grid Engine Linux cluster consisting of 20 dual 2 GHz Intel XEON machines with 512 kB of cache, 4 GB of RAM and 5 GB of swap space, running SuSE Linux with kernel 2.6.4-52-smp. All reported time values are wall clock times, not CPU times and may have been affected by other processes running at the same time.

All one-machine experiments were performed on a 1 GHz Pentium III with 256 kB of cache, 256 MB of RAM and 512 MB of swap space, running SuSE Linux with kernel 2.6.5-7.151-default. The reported times in this case are CPU times.

4.1 F-measure

The F-measure [51] is a function used often in the clustering literature to compare the similarity between two clusterings. More precisely, it is used to compare the quality of a clustering with respect to a known correct clustering for a given instance of the clustering problem.

Let $C = (C_1, C_2, \ldots, C_x)$ be a given clustering and $D = (D_1, D_2, \ldots, D_y)$ be the correct clustering. Then we call each of the D_i sets *classes*. The F-measure of a given cluster C_i with respect to a class D_j is then

$$F(C_i, D_j) = \frac{2|C_i \cup D_j|}{|C_i| + |D_j|}.$$

Let n be the total number of elements in all of the classes of D. Then the F-measure for the whole clustering C with respect to D is defined as

$$F(C, D) = \sum_j \frac{|D_j|}{n} \max_i F(C_i, D_j).$$

The value of the F-measure is always between 0 and 1 with 1 possible only in the case of a perfect match. Intuitively, it weights each of the classes according to its size and picks the cluster that best matches that class. The F-measure is used by Handl and Knowles [27] to report the results of a multiobjective evolutionary clustering algorithm (see Section 4.3.3).

Varying the number of depth runs on sf25.gra.

number of	CPU time			PE-set size		
depth runs	Min.	Avg.	Max.	Min.	Avg.	Max.
2	1.443 s	3.168 s	6.271 s	108	117.9	147
25	1.527 s	2.649 s	6.211 s	107	117.8	144
100	1.961 s	2.972 s	4.931 s	106	119.5	144
1000	8.989 s	9.827 s	11.634 s	109	120.3	141

Table 4.1: Dependence of the CPU time and the number of computed solutions on the number of depth runs for the sf25.gra input. 100 independent runs were performed in each case.

4.2 Unweighted Graphs

For unweighted graphs, we used two slightly different cost functions. These were derived from the van Dongen naive cost function [15] and are the same as those used during the first stage of King's single-objective RNSC algorithm [41] (see Section 2.2.1).

4.2.1 25-Vertex Scale-Free Graphs

A scale-free graph is an unweighted, undirected graph whose vertex degrees obey a power law. Such a graph is constructed by starting with k vertices and 0 edges and adding the other $n - k$ vertices iteratively. Each new vertex, v, is connected with k existing vertices uniformly at random. The probability of connecting v to u is proportional to the current degree of u. Scale-free graphs are believed to be good models of certain types of biological networks and other types of naturally occurring graphs [41].

To evaluate the effect of depth runs on the performance of the algorithm, we used a small scale-free graph (sf25.gra in King's RNSC data set). The smallest number of depth runs that we used was 2 (see Section 3.1.3 for the explanation). Table 4.1 shows that a small number of depth runs seems to improve the running time.

With this in mind, we ran the algorithm on 50 randomly generated 25-vertex scale-free graphs using 5 independent runs, initialized with 25 depth runs each. Figure 4.1 shows the distribution of the average size of the resulting PE-set versus the average CPU time (in seconds, using one machine) for each of the 50 random graphs. The figure clearly shows a positive correlation, which is not surprising because the algorithm's running time critically depends on the size of the PE-set. It also shows quite a large variation in the average CPU times and PE-set sizes.

4.2.2 100-Vertex Scale-Free Graph

Figure 4.2 shows the results of running our multiobjective algorithm on a 100-vertex scale-free graph (sf100.gra from King's RNSC data set). There are 18176 Pareto optimal clusterings. Also shown are the 1018 different clusterings produced using 10000 independent runs of RNSC (single-objective algorithm) with the naive cost function only. Our algorithm took 30 hours to run on a single machine, while RNSC required

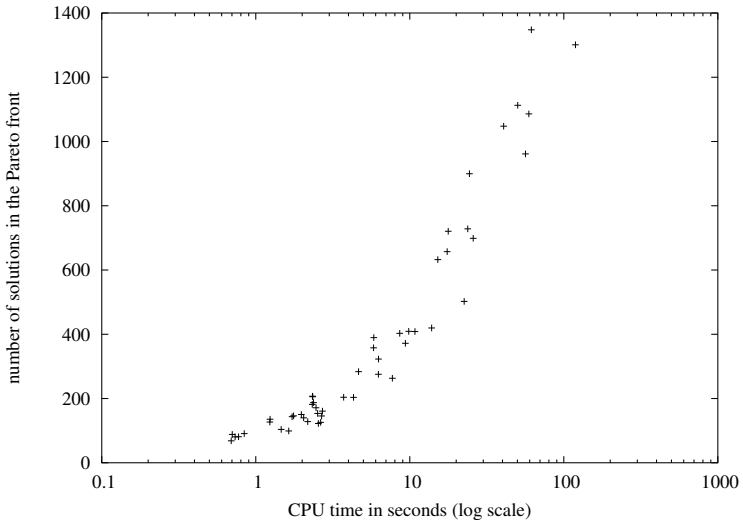

Figure 4.1: Size of the Pareto front for 50 random 25-vertex scale-free graphs with 5 independent runs, 25 depth runs each.

10 minutes. RNSC was executed with the naive stopping toleance set to 15 and the scaled stopping tolerance set to zero. No other parameters were changed from their default values. The value of 15 was chosen higher than the default to give the algorithm an opportunity to use more time in order to find a better solution.

4.3 Vector Sets

A vector set is a collection of n k-dimensional real vectors. It can be modelled by a weighted complete graph on n nodes, where each edge (u, v) has a weight that depends only on u and v. For these data sets, we used two cost functions based on the van Dongen weighted coverage measure. For general graphs, the weighted, scaled coverage measure would be a better choice, but in the case of complete graphs, the two are identical [15].

The weighted coverage measure works on weighted graphs where the graph is modelled as a stochastic flow network, and each edge's weight is a transition probability. To generate such a graph from a vector set, we chose to make each edge weight proportional to the inverse of the squared Euclidean distance between the two corresponding vertices. This makes sense if the vertices are considered to be embedded in the plane with a uniformly random distribution because in this case the number of vertices that are a distance r away from a given vertex u is proportional to r^2. Other ways of assigning transition probabilities are possible (correlation, nearest neighbours, *etc.*).

The two vector sets that we used are the Square1(100) data set and the Iris data set from the UCI Machine Learning Database [21].

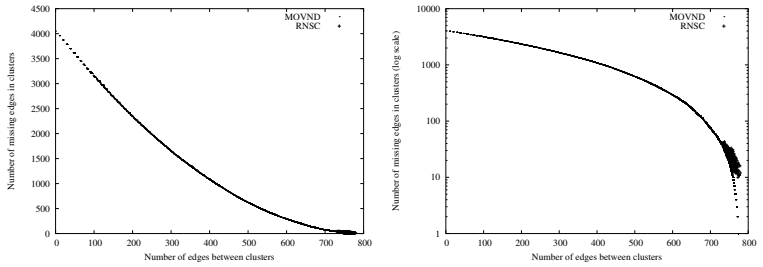

Figure 4.2: Comparison of the Pareto front to a single-objective algorithm on a scale-free graph with 100 vertices. The figure on the right shows the same plot, but with the y-axis in log scale.

4.3.1 Square1(100)

The first vector set we considered is a smaller version of the widely used Square1 dataset [46, 27, 31]. This one consists of 100 2-dimensional points grouped into 4 25-point clusters. Each cluster has a 2D Gaussian distribution with means $(\pm 1, \pm 1)$ and standard deviation of 0.3, forming 4 globular clusters with centers at the corners of a 2x2 square (see Figure 4.3). We generated this data set locally.

Figure 4.4 shows the clusterings that were found after running 1 process with 100 depth runs for 4 hours and 40 minutes until termination. The curve represents the 1179 Pareto optimal solutions found during this run of the algorithm. The 3 noted solutions are those containing exactly 4 clusters and similar to the correct clustering (after human inspection). They are unsupported and are not likely to be found by any algorithm that optimizes linear combinations of the two cost functions. One of those dots is precisely the correct clustering (the one with F-measure equal to 1.0).

4.3.2 Square1(64)

In order to be able to run multiple experiments, we built an even smaller version of the Square1 data set, consisting of only 64 points in 4 clusters, with the same mean and standard deviation as in the Square1(100) data set. Five independent runs of MOVND are described in Table 4.2. All 5 runs found the correct (F-measure 1.0) clustering. Table 4.3 shows the same results for 5 independent runs of MOCK. MOCK finishes much faster, but finds much fewer solutions. Note that the variability in F-measure values and Pareto front sizes produced by MOCK is much greater than that of our algorithm. All 5 runs of MOCK found the correct (F-measure 1.0) clustering.

4.3.3 Iris

This dataset contains 150 4-dimensional vectors with 3 known clusters of size 50 each. Each of the 4 components of the vectors corresponds to a geometric property of the Iris flower. We used two cost functions based on the van Dongen weighted coverage

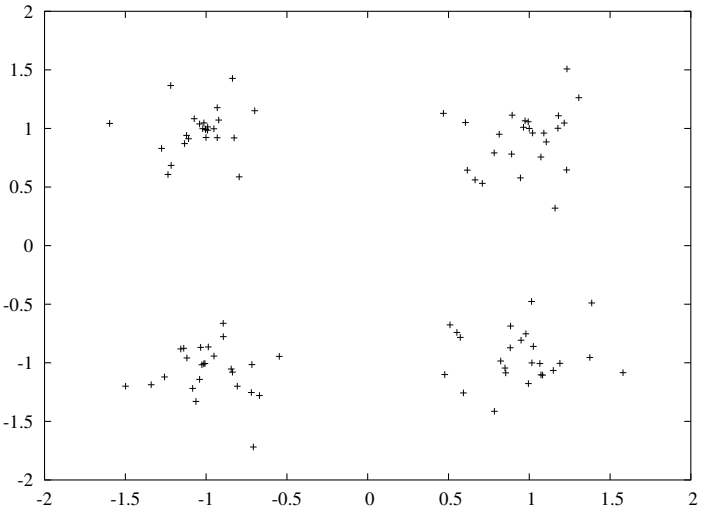

Figure 4.3: Square1(100) data set.

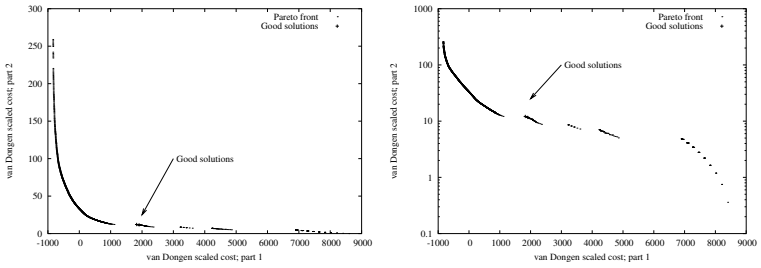

Figure 4.4: Pareto front for the Square1(100) data set with transition probabilities inversely proportional to squared Euclidean distance. The arrow shows the correct clustering and two solutions similar to it. The plot on the right displays the y-axis in log scale, which clearly shows that the good solutions are unsupported.

Independent runs of MOVND on Square1(64).

CPU time	F-measure quantiles q25	q50	q75	QVC	Size of the Pareto front
13m51.587s	0.75132	0.83598	0.90509	9.28333	298
9m2.162s	0.75016	0.83598	0.90509	9.35992	298
8m21.262s	0.75016	0.83598	0.90509	9.35992	300
8m45.138s	0.75132	0.84583	0.90509	9.28333	293
6m55.745s	0.75132	0.84091	0.90509	9.28333	297

Table 4.2: CPU running times, F-measure quantiles and Pareto front sizes for 5 independent runs of MOVND on the Square1(64) data set. QVC is the Quartile Variation Coefficient ($100\frac{q75-q25}{q75+q25}$).

Independent runs of MOCK on Square1(64).

CPU time	F-measure quantiles q25	q50	q75	QVC	Size of the Pareto front
13.277s	0.76921	0.82784	0.91117	8.44809	52
13.392s	0.85503	0.89074	0.93704	4.57627	36
11.374s	0.79227	0.84911	0.91419	7.14462	49
12.308s	0.75657	0.83921	0.90171	8.75244	50
11.548s	0.74619	0.83132	0.90436	9.58287	54

Table 4.3: CPU running times, F-measure quantiles and Pareto front sizes for 5 independent runs of MOVND on the Square1(64) data set.

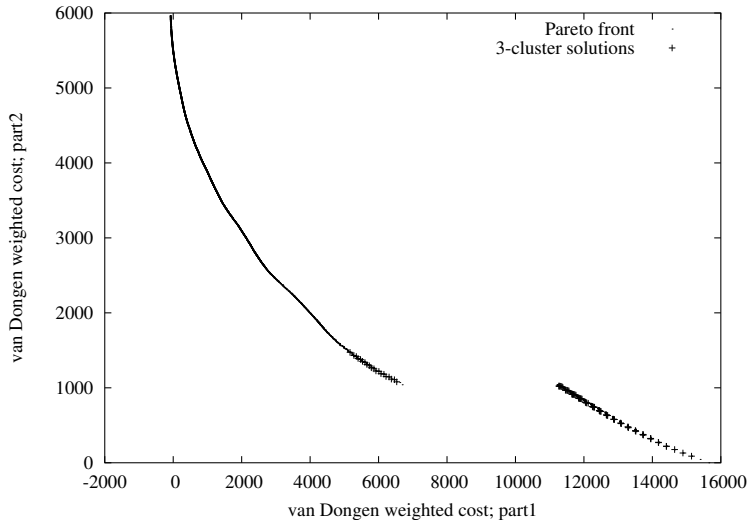

Figure 4.5: Iris with transition probabilities inversely proportional to Euclidean distance. The x- and y-axes are the cost functions h_1 and h_2 respectively (see Section 3.1.2). The curve represents all of the 3854 Pareto optimal solutions. The marked solutions are those with exactly 3 clusters.

measure (see Section 3.1.2). Since those functions work on a graph where edges correspond to transition probabilities, there is the question of generating such a graph from a set of 4-dimensional vectors. We used two methods. First, we made the probability inversely proportional to the Euclidean distance between the 4D points. This approach yielded poor results (see Figure 4.5). The curve shows all of the 3854 Pareto optimal solutions generated after about 60 hours of computation with 20 parallel processes, each one performing 50 depth runs and a subsequent multiobjective VND. The circles correspond to all of the clusterings that contained exactly 3 clusters. After a quick human inspection, all of those were found to be very different from the provided "correct" clustering.

The second approach was to make the transition probabilities inversely proportional to the square of the Euclidean distance. In this case (see Figure 4.6), the algorithm took 27 hours to run until termination and produced 2715 solutions (curve), 35 of them consisting of 3 clusters, 4 of which were similar to the expected clustering provided with the Iris dataset. The PE-set was obtained by running 20 parallel processes, each one performing 20 depth runs with subsequent multiobjective VND. Note in particular that the interesting solutions are all unsupported. Not one of them can be a minimizer of any linear function of h_1 and h_2 (x and y). Therefore, they would not be found by any algorithm that minimized only a linear combination of the two cost functions (unless the algorithm were lucky to terminate at just the right local minimum).

The good solutions displayed in Figure 4.6 are in fact not the best clusterings found

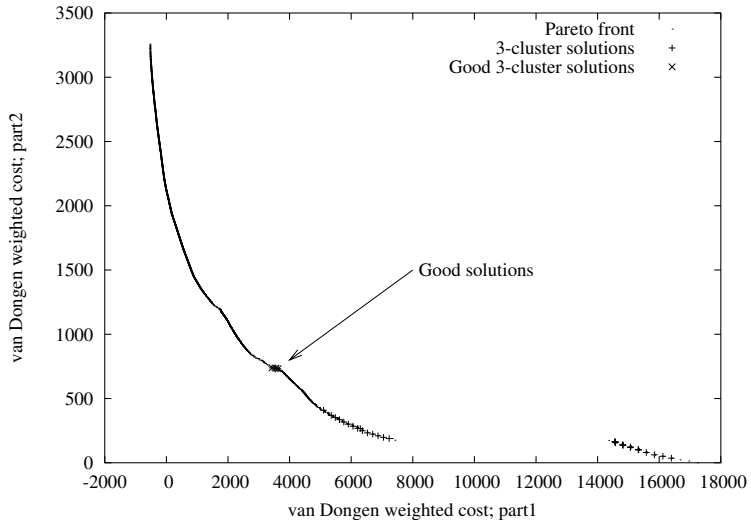

Figure 4.6: Iris results with transition probabilities inversely proportional to squared Euclidean distance. Note the marked unsupported solutions.

by our algorithm. They are certainly the best out of the ones that have exactly 3 clusters, but if we use the F-measure (see Section 4.1) to evaluate the quality of each clustering, we get a different, but conceptually very similar picture. Figure 4.7 again shows the computed Pareto front with the 10 "best" solutions marked. They are the best in terms of the highest value of the F-measure (it is 0.909676 for each of the marked clusterings, with a value of 1.0 corresponding to the correct clustering). These solutions have 6, 7, 8 or 10 clusters in them with 3 large clusters that closely match the 3 correct ones. Once again, note that these solutions are unsupported – they are found on a concave (up) portion of the Pareto front.

We ran the Handl and Knowles MOCK algorithm [27] on the Iris data. The algorithm terminated in under one minute and produced 98 solutions. The highest F-measure attained by a solution was 0.843172. Figure 4.8 shows a cumulative distribution of F-measure values of our algorithm and MOCK. Firstly, our algorithm produced substantially more solutions. Secondly, a large number of our solutions had higher F-measure values. The former could be attributed to the choice of cost functions and is certainly influenced by the differences in running time. It was impossible to change MOCK's parameters in order to increase its running time without a thorough understanding of the source code. We used the default external parameters, except for the maximum number of clusters, which was set equal to the size of the data set.

In hopes of making a closer comparison between MOVND and MOCK, we ran our algorithm on the Iris data set for 90 seconds (comparable to the running time of MOCK). We used the minimum of 2 depth runs, the first was initialized to the greedy clustering and finished immediately. The second depth run took 60 seconds to finish.

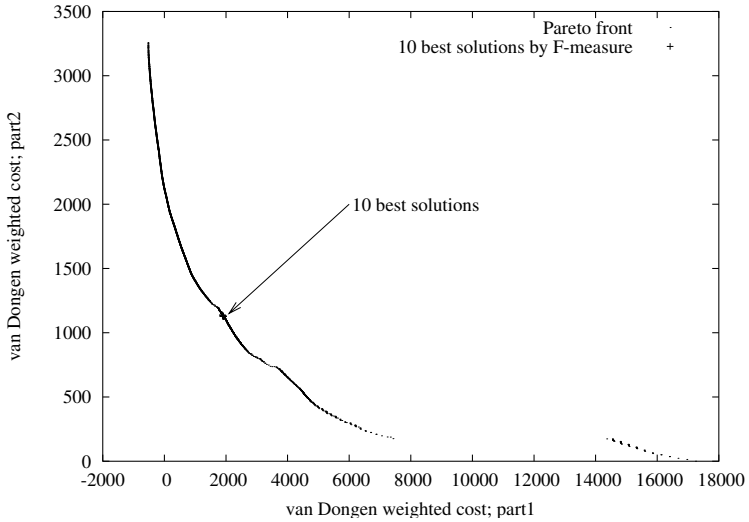

Figure 4.7: 10 best Iris results in terms of the F-measure.

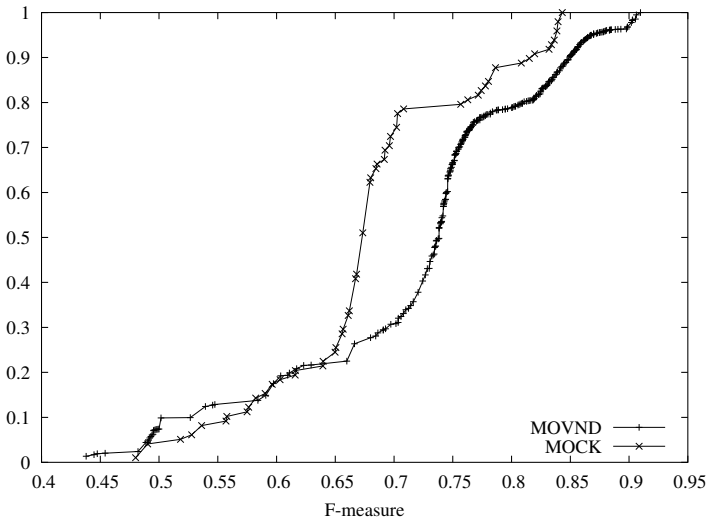

Figure 4.8: Cumulative distributions of F-measure values in the Pareto fronts for the Iris data set for a single run of MOCK and our algorithm (MOVND). Our algorithm has clearly found a number of solutions with higher F-measure values.

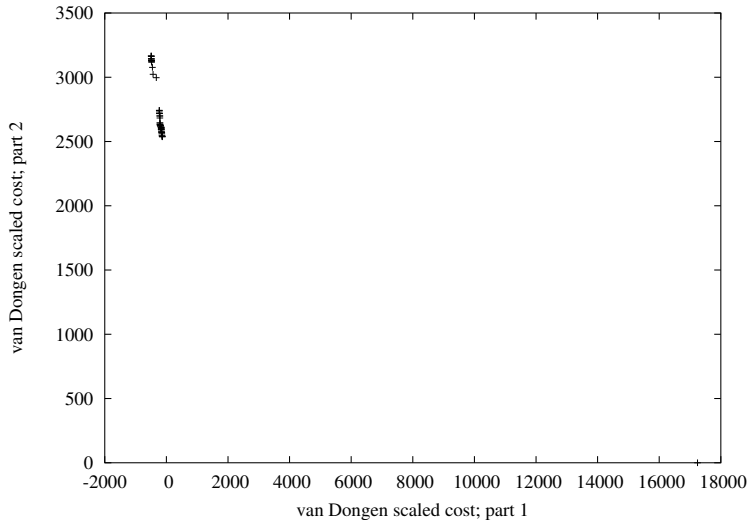

Figure 4.9: The Pareto front generated after 90 seconds of MOVND running on the Iris data set. Two depth runs were used.

The remaining 30 seconds were spent exploring the neighbourhood of the solution produced by the single non-trivial depth run. Figure 4.9 shows the pareto front with the 1-cluster solution on the bottom-right and a small number (47) of solutions on the top-left. The best of those solutions has an F-measure of 0.536440, which is worse than most of the solutions found by MOCK in the same CPU time.

Chapter 5

Discussion and Conclusions

We have described a Multiobjective Variable Neighbourhood Descent algorithm for the clustering problem and presented our results for a collection of data sets, some of which had random structure and others had known correct clusterings. On the structured data sets, the algorithm found interesting unsupported solutions, which is a clear advantage over single-objective clustering. On the random inputs, the algorithm required a substantial amount of running time, but in the end, produced a very wide Pareto front, covering a much larger portion of the search space than a set of independent runs of a single-objective algorithm did.

5.1 Discussion

In this section, we discuss the results and argue for the usefulness of our approach. After that, we describe several improvements that can be made to our algorithm.

5.1.1 Comparison to Single-Objective Clustering

The major advantage of multiobjective clustering over single-objective clustering can be clearly seen in the results. For both the Iris and Square1(100) data sets, our algorithm found unsupported solutions that are of interest and are not the minimizers of any linear combination of the two cost functions. Hence, no single-objective algorithm that looks for such minimizers will be able to find those solutions (except by accident).

The second advantage is conceptual. The Clustering problem is by its nature multiobjective. Defining the problem using a single cost function is a very difficult task, and many people avoid it by fixing the number of clusters (see Section 2.2). If the ”correct” number of clusters is unknown, then in most of the literature, the cost function is defined as the sum or product of two competing quality measures [19]. To avoid non-uniformity, sometimes those functions are somehow normalized or converted to similar ”units”. All of these difficulties can be avoided by using a multiobjective algorithm.

The main disadvantage of the multiobjective approach is its running time. Our algorithm has to maintain a large set of solutions, while a single-objective algorithm can usually keep just one solution. On the other hand, first of all, our algorithm's can be improved substantially (see Section 5.2). Secondly, the algorithm is highly parallelizable.

We can take advantage of the relative efficiency of single-objective algorithms by solving single-objective restrictions of the problem as a part in the multiobjective algorithm. As our results show, the single-objective RNSC algorithm is superiour to ours on

large graphs (2000 vertices). Section 5.2.3 talks about a way of exploiting the efficiency of single-objective algorithms without sacrificing the multiobjective advantages.

5.1.2 Comparison to Other Multiobjective Algorithms

Our algorithm is not the first one for multiobjective clustering. Handl and Knowles [27] applied a multiobjective evolutionary algorithm (MOCK) to solve the clustering problem. They used a different pair of cost functions (see Section 2.2.1) and also compared their approach to single-objective algorithms (k-means and average-link agglomerative clustering). Their algorithm found clusterings with a higher value of the F-measure than either of the single-objective algorithms.

They have results for the Iris dataset and report an F-measure of 0.840924 for the best clustering found by MOCK [27]. Our algorithm found 2715 Pareto optimal clusterings, 351 of which had an F-measure higher than 0.840924. We obtained an improved version of MOCK from the authors and performed a number of experiments on the Iris data set. Figure 4.8 shows that our algorithm produced a large number of solutions that were better (in terms of the F-measure) than the best solution found by MOCK. However, our running time was substantially longer.

In general, it is difficult to compare two multiobjective algorithms because it involves comparing two Pareto fronts, which is a non-trivial task. In the case of our algorithm *vs.*MOCK, performing a fair comparison would involve making substantial modifications to the MOCK source code. According to [30], the latest version of MOCK, which we used to run the experiments, uses a fixed number of generations (500) and a maximum external population size (1000) in the evolutionary algorithm. MOCK was designed to compete with single-objective clustering algorithms. The implementation also contains a post-processing stage that selects the best solutions based on the properties of the Pareto front. These selected solutions are shown to be better in most cases than those produced by k-means and agglomerative clustering (on data sets with known expected clusterings). All three algorithms have comparable running times. When executing on the Iris data set, our algorithm's PE-set size grew substantially beyond 1000, and the algorithm needed thousands of iterations of the VND() procedure to converge to a multiobjective local minimum. It would be interesting to modify the internal parameters of MOCK to allow for a comparable running time.

The fact that our algorithm and MOCK optimize completely different cost functions is another obstacle to making a fair comparison. We believe that our cost functions (those based on the van Dongen coverage measures) are better than the ones in [27] because they require no external parameters. However, substantially more comparison testing is required before such a claim can be verified in practice with any degree of certainty. In any case, it would be interesting to implement their cost functions in our algorithm and run both to see a comparison in terms of running time and memory usage.

5.1.3 Examining the Pareto Front

In both the Iris and Square1(100) data sets, the interesting solutions were found near the kinks in the otherwise smooth Pareto front (Figures 4.6 and 4.3). This suggests

that perhaps those are the places where one should look for interesting solutions, in general. One could speculate that the smooth, convex portions of the Pareto front are unimportant because they correspond to sequences of solutions that all incur a large local penalty in one of the cost functions. On the other hand, portions of the front that are concave (up) or have sudden jumps deserve more attention because something unusual is happening there.

Branke *et al.*[7] talk about "knees" in multiobjective optimization – solutions that corresponds to irregular sections of the Pareto front. More precisely, they argue that interesting solutions are those that have the property that making a small change in one cost function leads to a large change in the cost objective function.

The idea of examining the Pareto front in order to automatically select the best clusterings is not new. Handl and Knowles [29, 28] use a similar idea in MOCK (a multiobjective evolutionary algorithm). Although their approach is application-specific, experimental results show that it improves the quality of the computed clusterings.

One could go further and make a claim that if the Pareto front for a given problem instance looks "smooth" and convex, then the input is too random and has no good clusterings. This observation seems to be supported by our results for the randomly generated graphs. If a number of human observers were given the task of clustering a random scale-free graph, for instance, they would probably come up with different answers. It would be very interesting to investigate this properly, with human subjects, but due to time constraints, we did not pursue this direction.

5.1.4 Advantages of Depth Runs

Depth runs (see Section 3.1.3) serve the purpose of initializing the PE-set to a diverse set of fairly good solutions that are likely to dominate most of the solutions that will be found during the execution of the algorithm. This is important because the speed of the algorithm depends on the size of the PE-set, and we would like to keep this set relatively small. The danger, however, is that too many solutions will be rejected, and the multiobjective VND will hit a local minimum very quickly.

The other purpose of the depth runs is to initialize the PE-set with a diverse family of solutions in hopes of covering as much of the search space as possible. Our tests on a small (25-vertex) scale-free graph show that a moderate number of depth runs improved the running time of the algorithm (see Section 4.2.1) without adversely affecting the solution quality. Section 5.2.3 describes an alternate approach that can replace depth runs.

5.1.5 Running on Larger Random Graphs

To evaluate the scalability of the algorithm, we ran it on a weighted graph built by thresholding a 500x500 correlation matrix that came from an gene expression experiment on an Affymetrix microarray. In this case, we replaced the first cost function (the number of edges between clusters) by its weighted version (the sum of edge weights between clusters). The second function was kept the same (the number of missing edges in each cluster). This way, the first function was real-valued and the second one was integer valued.

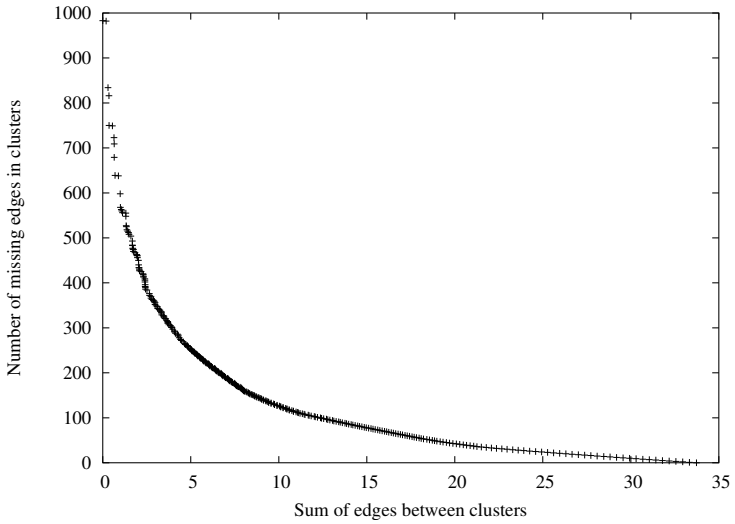

Figure 5.1: Pareto front for a 500-vertex weighted graph built by thresholding a correlation matrix.

After 38.5 hours running as a single process, with 1000 depth runs, the algorithm terminated with the 357 results shown in Figure 5.1. In this experiment, we used only the first two neighbourhoods (N_1 and N_2). N_3 had not been implemented yet. This is the largest graph that the algorithm could handle, and it was difficult to cluster, which is indicated by the smooth Pareto front.

Another type of graph we have considered is a 2000-vertex graph (11.gra from King's RNSC data set). The graph proved to be virtually impossible to cluster with the current version of our algorithm. After several days of running 5 parallel processes, it found only the few results shown in Figure 5.2. A single run of the RNSC algorithm immediately produced an undominated clustering. Hopefully, adding some of the improvements discussed in Future Work will make our implementation more scalable. It is also unclear how scalable Handl and Knowles' MOCK algorithm is. Alghough it can handle much larger input instances, our algorithm produces better results on the Iris data set.

5.2 Future Work

This section describes some of the improvements that can and should be made to the algorithm and its implementation in order to achieve the best performance. They are listed in decreasing order of importance, according to our best judgement.

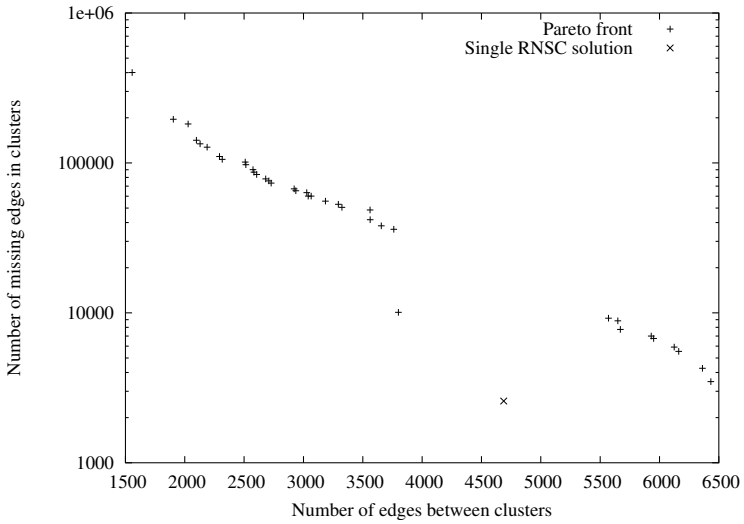

Figure 5.2: Results for a 2000-vertex graph compared to a single run of RNSC. The y-axis is shown in log scale.

5.2.1 Speeding up Updates of the PE-set

Majority of the time in the algorithm is spent updating the set of potentially efficient solutions (PE-set). The datastructure used to represent the PE-set is trivial and inefficient. A much better implementation would use a range tree, a BSP-tree or a kd-tree [4], all of which would allow sub-linear time queries and insertions.

If we restrict attention to only two cost functions, then a simple balanced binary search tree would allow $O(\log n)$ amortized time queries and insertions. The tree of solutions can be kept sorted by the values of the first cost function (non-decreasing). The non-dominance condition on the PE-set would then guarantee that the solutions are also sorted by the values of the second cost function (non-increasing). Checking whether a given solution is dominated by the set is then a simple matter of binary searching the tree. Inserting a solution and removing all existing solutions that are dominated by it can be done in $O(\log n + r)$ worst-case time, or $O(\log n)$ amortized time, where r is the number of solutions to be removed from the set during one insertion.

5.2.2 Adding 'seen' Flags to Explored Solutions

One very useful idea can be borrowed from "don't look" bits and Tabu search [35]. In the current implementation, the algorithm proceeds by picking a random solution from the PE-set and exploring its neighbourhood. The next randomly picked solution could happen to be one of those that have been picked previously. This constitutes a waste of computation time.

A simple fix is to add a 'seen' flag to each solution in the PE-set to ensure its neighbourhood is never explored twice. These flags need to be taken care of in the distributed context as well. If worker process number 1 has explored solution s, then it needs to relay the fact that s's 'seen' flag is now true to all of the other worker processes through the server. This would slightly increase the bandwidth requirements, and due to synchronization issues, using 'seen' flags alone would not completely eliminate wasted computation. Nevertheless, this change is very likely to improve the algorithm's performance.

5.2.3 Needle Runs

Our algorithm relies on a sequence of depth runs (see Section 3.1.3) to initialize the PE-set. The advantage of depth runs is their efficiency. They do not use the PE-set and try to converge to a local minimum as quickly as possible. A similar idea can be used during the execution of the algorithm. We could use single-objective restrictions of the multiobjective problem in order to "poke a hole" in the Pareto optimal frontier and quickly find a new potentially efficient solution. This idea was used before by Paquete and Stützle [49] for the multiobjective Traveling Salesman problem.

Consider a Pareto optimal set of solutions (an instance of the PE-set). Now pick any solution $s \in PE$. Consider a convex combination of the objective functions $f_1, f_2, \ldots, f_k$:

$$f = \sum_{i=1}^{k} w_i f_i, \sum w_i = 1.$$

The function f defines a single-objective restriction of the problem. Now we can use a faster single-objective optimization algorithm to get a locally optimal solution $\hat{s}$ that minimizes f. Geometrically, imagine the PE set as a set of points in a k-dimensional objective space. Then optimizing a single-objective restriction, looking for $\hat{s}$, corresponds to starting at point s and looking for better solutions in the direction defined by the weight vector $w = (w_1, w_2, \ldots, w_k)$. In essence, we are trying to poke a hole through our existing Pareto optimal frontier and get a new non-dominated solution.

These "needle runs" have the advantage of being computationally inexpensive, compared to the multiobjective optimization algorithm. On the other hand, they are unlikely to find unsupported solutions (see Definition 2.3.3). This is why we cannot rely exclusively on the needle runs to generate the PE-set, although they are likely to improve the running time of the algorithm and the quality of the solutions.

5.2.4 Using Component-Based Cost Functions

To cluster a set of k-dimensional vectors, we first treat each one as a point and compute pairwise Euclidean distances. Other distance metrics can be used instead, such as correlation or Cosine measure [27]. The distances are then used to compute the transition probabilities for the stochastic flow simulation network that is used to define the van Dongen cost function. However, real world data sets may have vectors for which the Euclidean distance has little meaning. In the Iris data set, the vectors are 4-dimensional, and the 4 values correspond to some geometric characteristics of the flower's shape.

It would be in keeping with the core idea of multiobjective optimization to define 4 separate cost functions, one for each component of the vectors. Perhaps, we could even use 2 cost functions per component, for a total of 8. The problem essentially reduces to defining a good pair of cost functions for clustering a set of real numbers. This is a difficult task in itself, and our initial attempt at defining a pair of simple functions failed.

A possible downside of this approach is the fact that the running time will be adversely affected by the large number of cost functions. What is worse, the size of the PE-set will probably grow substantially. In our preliminary experiments, adding an extra cost function has caused the PE-set to increase in size to the point of becoming unmanageable.

5.2.5 Allowing Worsening Diversification Moves

Our Multiobjective Variable Neighbourhood Descent algorithm, as described above, simply performs first-improvement hill climbing while switching between three solution neighbourhoods. In addition, we keep a set of potentially efficient solutions found so far that affects whether any new solution is considered an improvement or not. In theory, this approach might get stuck in local minima fairly easily.

A potentially beneficial idea to investigate is the possibility of allowing worsening diversification steps in the neighbourhoods N_2 and N_3. Some care needs to be taken to ensure that these sub-optimal solutions are kept separate from the PE-set in order to maintain the mutual non-dominance invariant. MOCK [27] uses a similar idea by keeping two solution populations – external (the Pareto front) and internal (dominated). It would be interesting to see how extending the algorithm into a multiobjective iterated local search would affect the algorithm's overall performance.

5.2.6 Optimizing Parallel Computing

In the current implementation, a number of worker processes generate new solutions and periodically synchronize their copies of the PE-set with a server process. The timeout for synchronization is a constant, W, which is a severely limiting condition. Ideally, W should be kept as low as possible to ensure little or no duplicated work among the worker processes. However, setting W too low would create a bottleneck at the server process. The value of W should depend on the size of the PE-set and on the overall real-time performance of the system. If the workers find themselves waiting for the server, then W should be increased automatically. Otherwise, it should be decreased.

A more sophisticated strategy for maintaining several copies of the PE-set is also possible. For instance, a hierarchy imposed on the worker nodes could help reduce the bottleneck that appears at the server process. Each node could be made to communicate with its parent and at most two children, thus distributing the networking costs at the price of decreased synchronization of the PE-set copies.

Bibliography

[1] E. H. L. Aarts, J. K. Lenstra, ”Local Search in Combinatorial Optimization,” Wiley-Interscience, ISBN 0471948225, 1997.

[2] U. Alon, N. Barkai, D. A. Notterman, K. Gish, S. Ybarra, D. Mack and A. J. Levine, ”Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays,” In Proceedings of *National Academy of Science*, USA. 96(12):6745-6750, 1999.

[3] D. Beeferman and A. Berger, ”Agglomerative clustering of a search engine query log,” In Proceedings of *ACM SIGKDD International Conference*, pages 407-415, 2000.

[4] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 1997.

[5] M. den Besten and T. Stützle, ”Neighborhoods Revisited: An Experimental Investigation into the Effectiveness of Variable Neighborhood Descent for Scheduling,” *MIC’2001 – 4th Metaheuristics International Conference*, pp. 545-549, 2001.

[6] J. Bezdek, ”Pattern recognition with fuzzy objective function algorithms,” *Plenum Press*, New York, 1981.

[7] J. Branke, K. Deb, H. Dierolf and M. Osswald, ”Finding Knees in Multi-objective Optimization,” In Proceedings of *The Eighth International Conference on Parallel Problem Solving from Nature*, pp. 722-731, Springer, Heidelberg, 2004.

[8] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Massachusetts, 1990.

[9] C. A. Coello, ”A comprehensive survey of evolutionary-based multiobjective optimization techniques,” *Knowledge and Information Systems*, 1:269-308, 1999.

[10] F. Corpet, ”Multiple sequence alignment with hierarchical clustering,” *Nucleic Acids Research*, v. 16(22):10881-10890, 1988.

[11] P. Crescenzi and V. Kann, ”A compendium of NP optimization problems,” Manuscript, available at http://www.nada.kth.se/theory/problemlist.html, 1997.

[12] J. Crispim and J. Brandão, ”Reactive Tabu Search and Variable Neighbourhood Descent Applied to the Vehicle Routing Problem with Backhauls,” *MIC’2001 – 4th Metaheuristics International Conference*, pages 631-636, 2001.

[13] A. Dempster, N. Laird and D. Rubin, ”Maximum likelihood from incomplete data via the EM algorithm,” *Journal of the Royal Statistical Society B*, 39:1-38, 1977.

[14] I. S. Dhillon, Y. Guan and J. Kogan, ”Iterative Clustering of High Dimensional Text Data Augmented by Local Search,” In Proceedings of *The 2002 IEEE International Conference on Data Mining*, 2002.

[15] S. van Dongen, ”Performance Criteria for Graph Clustering and Markov Cluster Experiments,” *Information Systems*, Center for Mathematics and Computer Science (CWI), Amsterdam, report INS-R0012 ISSN 1386-3681, ftp://ftp.cwi.nl/pub/CWIreports/INS/INS-R0012.pdf 2000.

[16] S. Van Dongen, ”Graph clustering by flow simulation,” PhD Thesis, University of Utrecht, The Netherlands, 2000.

[17] J. Edachery, A. Sen and F. J. Brandenbur, ”Graph Clustering Using Distance-k Cliques,” In Proceedings of *The 7th International Symposium on Graph Drawing*, GD ’99, (Jan Kratochvíl, editor), volume 1731 of Lecture notes in Computer Science, Springer, pages 98-106, 1999.

[18] M. Ehrgott and X. Gandibleux, ”Approximative Solution Methods for Multiobjective Combinatorial Optimization,” *Top*, 12(1):1-90, 2004.

[19] V. Estivill-Castro, ”Why so many clustering algorithms: a position paper,” *ACM SIGKDD Explorations Newsletter*, 4(1):65-75, 2002.

[20] T. Feder and D. Greene, ”Optimal algorithms for approximate clustering,” In Proceedings of *The 20th Annual ACM Symposium on Theory of Computing*, Chicago, Ill., May 2-4, ACM, New York, pages 434-444, 1988.

[21] R. A. Fisher, ”The use of multiple measurments in taxonomic problems,” *Annals of Eugenics*, 7:179-188, 1936.

[22] R. W. Floyd, ”Algorithm 97: Shortest Path,” C.ACM, 5(6):345, 1963.

[23] P. Fränti and J. Kivijärvi, ”Randomised Local Search Algorithm for the Clustering Problem,” *Pattern Analysis and Applications*, 3(4):358-369, 2000.

[24] F. Glover, E. Taillard and D. de Werra, ”A user’s guide to tabu search,” *Annals of Operations Research*, 41(1,4):3-28, 1993.

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

[26] J. A. Hartigan and M. A. Wong, ”A k-means clustering algorithm,” *Applied Statistics* 28(100), 1979.

[27] J. Handl and J. Knowles, ”Evolutionary Multiobjective Clustering,” In Proceedings of *The Eighth International Conference on Parallel Problem Solving from Nature*, (PPSN VIII), pages 1081-1091, 2004.

[28] J. Handl and J. Knowles, ”Multiobjective clustering with automatic determination of the number of clusters,” Technical Report TR-COMPSYSBIO-2004-02, UMIST, Manchester, UK. Under submission, 2004.

[29] J. Handl and J. Knowles, ”Exploiting the trade-off – the benefits of multiple objectives in data clustering,” *Third International Conference on Evolutionary Multi-Criterion Optimization*, EMO, 2005.

[30] J. Handl and J. Knowles, ”Improving the scalability of multiobjective clustering,” Technical Report TR-COMPSYSBIO-2005-03. UMIST, Manchester, UK, 2005.

[31] J. Handl, J. Knowles and M. Dorigo, ”Strategies for the Increased Robustness of Ant-Based Clustering,” Postproceedings of *The First International Workshop on Engineering Self-Organising Applications (ESOA 2003)*, 2003.

[32] P. Hansen and N. Mladenović, ”An introduction to variable neighborhood search,” In S. Voss, S. Martello, I. H. Osman and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 422-458. Kluwer Academic Publishers, Boston, MA, USA, 1999.

[33] A. Hertz and M. Mittaz, ”A Variable Neighborhood Descent Algorithm for the Undirected Capacitated Arc Routing Problem,” *Transportation Science*, 35(4):425-434, 2001.

[34] J. H. Holland, ”Adaptation in natural and artificial systems,” The University of Michigan Press, Ann Arbor, 1975.

[35] H. Hoos and T. Stützle, ”Stochastic Local Search: Foundations and Applications,” Morgan Kaufmann Publishers, ISBN:1-55860-872-9, 2005.

[36] Z. Huang, ”A Fast Clustering Algorithm to Cluster Very Large Categorical Data Sets in Data Mining,” *Research Issues on Data Mining and Knowledge Discovery*, 1997.

[37] S.-W. Hur and J. Lillis, ”Relaxation and Clustering in a Local Search Framework: Application to Linear Placement,” *VLSI Design* 14(2):143-154, 2002.

[38] A. K. Jain, M. N. Murty and P. J. Flynn, ”Data clustering: A review,” *ACM Computing Surveys*, 31:264-323, 1999.

[39] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman and A. Y. Wu, ”A Local Search Approximation Algorithm for k-Means Clustering,” In Proceedings of *The 18th Annual ACM Symposium on Computational Geometry*, pages 10-18, 2002.

[40] M. Y. Kiang, ”Extending the Kohonen self-organizing map networks for clustering analysis,” *Computational Statistics and Data Analysis*, 38:161-180, 2001.

[41] A. D. King, ”Graph Clustering with Restricted Neighbourhood Search,” MSc Thesis, University of Toronto, Graduate Department of Computer Science, 2004.

[42] J. Kleinberg, ”An impossibility theorem for clustering,” In Proceedings of *The 15th Conference on Neural Information Processing Systems*, Vancouver, Canada, 2002.

[43] T. Kohonen, ”Self-organizing maps”, *Springer-Verlag*, New York, USA, 1997.

[44] T. Kurita, ”An efficient agglomerative clustering algorithm using a heap,” *Pattern Recognition*, 24(3):205-209, 1991.

[45] M. H. C. Law, A. P. Topchy and A. K. Jain, ”Multiobjective Data Clustering,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2004.

[46] D. Merkle, M. Middendorf and A. Scheidler, ”Decentralized Packet Clustering in Networks,” In Proceedings of *The 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[47] N. Mladenović and P. Hansen, ”Variable neighborhood search,” *Computers and Operations Research*, 24(11):1097-1100, 1997.

[48] P. Pantel and D. Lin, ”Discovering word senses from text,” In Proceedings of *The ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 613-619, 2002.

[49] L. Paquete and T. Stützle, ”A Two-Phase Local Search for the Biobjective Traveling Salesman Problem,” In Proceedings of *Evolutionary Multi-Criterion Optimization: Second International Conference, EMO 2003*, Faro, Portugal, pages 479-493, April 8-11, 2003.

[50] H.-P. Schwefel, ”Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie,” *Interdisciplinary systems research*, 26, Birkhauser, Basel, 1977.

[51] M. Steinbach, G. Karypis and V. Kumar, ”A Comparison of Document Clustering Techniques,” Univesity of Minnesota, dept. of Computer Science and Engineering, Technical Report #00-034, 2000.

[52] M. Sultan, D. A. Wigle, C. A. Cumbaa, M. Maziarz, J. Glasgow, M. S. Tsao and I. Jurisica, ”Binary tree-structured vector quantization approach to clustering and visualizing microarray data,” *Bioinformatics*, 18(1):S111-S119, 2002.

[53] A. Ushioda and J. Kawasaki, ”Hierarchical clustering of words and application to NLP tasks,” In E. Ejerhed and I. Dagan (Eds.), *Fourth Workshop on Very Large Corpora* pages 28-41, Somerset, New Jersey: Association for Computational Linguistics, 1996.

[54] J. Vesanto and E. Alhoniemi, ”Clustering of the Self-Organizing Map,” *IEEE Transactions on Neural Networks*, 11(3):586-600, 2000.

[55] S. Warshall, ”A Theorem on Boolean Matrices,” *Jounal of the ACM*, 9(1):11-12, 1963.

[56] S. D. M. White, C. S. Frenk, ”Galaxy formation through hierarchical clustering,” *Astrophysical Journal*, Part 1 (ISSN 0004-637X), 379:52-79, 1991.

[57] Y. Yaari, ”Segmentation of expository texts by hierarchical agglomerative clustering,” In Proceedings of *RANLP'97*, Bulgaria, 1997.

[58] T. Zhang, R. Ramakrishnan and M. Livny, ”BIRCH: an efficient data clustering method for very large databases,” In Proceedings of *ACM SIGMOD*, 25(2):103-114, 1996.