

Genetic Algorithm and Simulated Annealing
methods on the Graph Partitioning Problem:
Implementation and Analysis

Philip Bradley

Department of Computer Science

Abstract

This report describes the problem of graph partitioning and presents 4 algorithms which may be applied to the problem, Local Search, the Kernighan-Lin algorithm, Simulated Annealing and a hybrid Genetic Algorithm. Experimental results on test graphs indicate the relative performance of all these methods. The effects of varying the parameters to Simulated Annealing and to the Genetic Algorithm are studied. A comparison is made between the performance of Simulated Annealing with single vertex migrations and with double vertex migrations. Methods of achieving balanced partitioning with single vertex migrations are explored.

Contents

1	Introduction	1
1.1	Problem and Notation	1
2	Local Optimisation Methods for GPP	3
2.1	Local Search	3
2.2	Local Search and GPP	3
2.3	The Kernighan-Lin Algorithm	4
2.4	The Kernighan-Lin Algorithm as a Local Improvement Heuristic	5
3	Global Optimisation Methods for GPP	7
3.1	Simulated Annealing	7
3.2	Simulated Annealing and GPP	7
3.3	An Overview of Genetic Algorithms	8
3.4	A Hybrid Genetic Algorithm for GPP	10
4	Design & Implementation	11
4.1	A Description of the Data Classes used in GPP	11
4.1.1	The methods associated with class Graph	12
4.1.2	The methods associated with class Solution	13
4.1.3	The methods associated with class Population	14
4.2	A Description of the Algorithmic classes used in GPP	15
4.2.1	Local Search	15
4.2.2	Simulated Annealing	16
4.2.3	Genetic Algorithm	16
4.2.4	The Kernighan-Lin Algorithm	17
4.3	Testing Strategy	17
4.4	Tools and Utilities	17

5	Results	18
5.1	Parameters of Simulated Annealing	18
5.1.1	The Imbalance Scaling (α)	19
5.1.2	The Cooling Rate	20
5.1.3	Initial Probability	20
5.1.4	Freeze Point	20
5.1.5	Run Length	21
5.2	Parameters of Genetic Algorithm	22
5.2.1	The Population Size	22
5.2.2	Maximum Iterations	23
5.2.3	Kernighan-Lin Iterations	23
5.3	Comparison of GA, SA, Kernighan-Lin, and Local Search	24
5.4	Simulated annealing with single & double migrations	26
6	Conclusion & Future Work	28
A	Source Code	31
A.1	BadVertexSwapException	31
A.2	BadPartitionException	32
A.3	CostStructure	33
A.4	GA	35
A.5	GeometricGraph	38
A.6	KerLin	47
A.7	LocalOpt	60
A.8	NoSuchEdgeException	63
A.9	NoVerticesAvailableException	64
A.10	Population	65
A.11	Queue	69
A.12	SA	72
A.13	Solution	83
A.14	CreateGraph	99

List of Figures

2.1	Local Search Algorithm	4
2.2	Local Improvement Algorithm for GA	6
3.1	Simulated Annealing Algorithm	8
3.2	Single point crossover	9
3.3	A typical genetic algorithm	9
3.4	The genetic algorithm for GPP	10
4.1	The object model for classes in GPP	12
5.1	The effect of α on the solution quality and running time.	19
5.2	The effect of <i>Cooling Rate</i> on the solution quality and running time.	20
5.3	The effect of <i>Initial Probability</i> on the solution quality and running time.	21
5.4	The effect of <i>Freeze Point</i> on the solution quality and running time.	21
5.5	The effect of <i>Run Length</i> on the solution quality and running time.	22
5.6	The effect of <i>Population Size</i> on the solution quality and running time.	23
5.7	The effect of <i>Maximum Iterations</i> on the solution quality and running time.	24
5.8	The effect of <i>Kernighan-Lin Iterations</i> on the solution quality and running time.	24
5.9	Comparison of single and double vertex migrations for different values of <i>Cooling Rate</i>	26
5.10	Comparison of single and double vertex migrations for different values of <i>Freeze Point</i>	27
5.11	Comparison of single and double vertex migrations for different values of <i>Initial Probability</i>	27
5.12	Comparison of single and double vertex migrations for different values of <i>Run Length</i>	27

Chapter 1

Introduction

The Graph Partitioning Problem (GPP) arises in many areas including task scheduling, VLSI design, and computer communications networks design. Since the problem has been proved to be NP complete (see Johnson et al. [1]), a number of heuristics which find good, but not necessarily optimal solutions have been proposed. This report describes a number of such approaches including local search, the Kernighan-Lin algorithm, simulated annealing, and a hybrid genetic algorithm. The project has the following goals:

- Design and Implementation of simulated annealing and a genetic algorithm for finding a *near optimal* solution to the graph partitioning problem.
- Investigation of simulated annealing for the graph partitioning problem according to different initial configurations.
- Investigation of the hybrid genetic algorithm for the graph partitioning problem according to different initial configurations.
- Comparative study of the performance of the genetic algorithm and simulated annealing applied to a range of graphs.
- Investigation of the value of modifications to both algorithms, proposed within this report, on the performance of the algorithms.

1.1 Problem and Notation

Let $G = (V, E)$ be an undirected weighted graph. Let w_i be the weight of the vertex v_i and w_{ij} be the weight of the edge (v_i, v_j) . A k -way partitioning $P = (V_1 \cdots V_k)$ is given by $V = \cup_{i=0} V_i$ and $V_i \cap V_j = \phi$ for $i \neq j$. An edge (v_i, v_j) is said to be external to V_i if and only if one vertex is in V_i , and internal if both v_i and v_j are in V_i .

Given a partition P , we define the cost of P , $C(P)$ to be:

$$C(P) = E_p + \alpha \sum_{i=0}^k (V_i - \bar{V})^2 \quad (1.1)$$

where E_p is the sum of the weights of the external edges.

The second term refers to the *imbalance* of the resulting solution. This term is called the penalty function and the imbalance factor (α), is called the penalty parameter. The latter is used to attach a measure of importance to the balance of the solution, as this may vary depending on the problem to which the technique is being applied. The problem is to minimise $C(P)$.

This formulation also allows for balanced partitioning. In a case where an algorithm produces only balanced partitions, the cost function simplifies to $C(P) = E_p$ since $\forall V_i, |V_i| = |\bar{V}|$

Chapter 2

Local Optimisation Methods for GPP

2.1 Local Search

Local search is a technique for finding a solution to a combinatorial optimisation problem. A local search algorithm for a specific problem requires the following two concepts to be defined:

Cost It must be possible to map a solution onto a numeric value reflecting the quality of that solution.

Neighbourhood A function *neighbourhood* for deriving a new solution from a given solution by means of a *local* perturbation method must be defined. The Neighbourhood of a solution S , denoted $\mathcal{N}(S)$ is the set of solutions that can be derived from that solution, by a single application of the neighbour function.

A solution S is locally optimum if

$$\forall S' \in \mathcal{N}(S), \text{cost}(S') \geq \text{cost}(S)$$

An arbitrary solution may be made locally optimum by application of the procedure shown in figure 2.1.

The search is restricted to those solutions which lie within $\mathcal{N}(S)$ where S is the initial solution. Thus, although the returned value is locally optimum, it is not guaranteed to be globally optimum, that is, there may be another locally optimum solution of lower cost.

2.2 Local Search and GPP

Local search can be applied naturally to the graph partitioning problem. For clarity, the partitioning is assumed to be 2-way. Given a graph $G = (V, E)$ with partition

```

while (There is an untested neighbour of S) do
  Solution S = new Solution()
  S' = neighbour(S)
  if (cost(S') ≤ cost(S))
    then
      S = S'
    fi
end while
return(S)

```

Figure 2.1: Local Search Algorithm

$P = (V_1, V_2)$, then the cost $C(P)$ is defined as:

$$C(P) = E_p + \alpha(V_1 - V_2)^2 \quad (2.1)$$

One possible way to define the neighbourhood of a partition P , $\mathcal{N}(P)$, is all those partitions which differ from P by the position of one vertex only. Thus if $P = (V_1, V_2)$ and $v \in V_1$, then $(V_1 - v, V_2 + v)$ is in the neighbourhood of P .

Balanced partitioning may be accomplished if the initial partition is balanced and the definition of $\mathcal{N}(P)$ is modified such that it denotes all those solutions obtainable by switching two vertices rather than just one vertex. Thus, if P is a partition of G , as above, and $u \in V_1, v \in V_2$ then, $\{V_1 + v - u, V_2 + u - v\} \in \mathcal{N}(P)$.

2.3 The Kernighan-Lin Algorithm

The Kernighan-Lin algorithm, is a local optimisation algorithm for finding a 2-way partitioning of a graph. Given an initial solution, the algorithm improves on it by swapping equal sized subsets of vertices between the partitions. This process is repeated until no further improvement can be found.

Let $u \in V_1$ and $v \in V_2$. Let g_v denote the reduction in cut size when the vertex v is moved from its current partition to the opposite partition and let $g(u, v)$ denote the reduction in cut size (or *gain*) when vertices u and v are swapped. Let w_{ij} denote the weight of the edge (i, j) . Then, according to [2].

$$g(u, v) = g_u + g_v - 2 \times \delta(u, v) \quad (2.2)$$

where

$$\delta(u, v) = \begin{cases} w_{ij} & (u, v) \in E \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

The pair (u, v) which maximises $g(u, v)$ is selected and eliminated from further selection on that iteration. A sequence of pairs $(u_1, v_1), \dots, (u_{\frac{n}{2}-1}, v_{\frac{n}{2}-1})$ is created.

The algorithm then chooses a pair (X, Y) , $X = u_1, \dots, u_m$, $Y = v_1, \dots, v_m$ where $m \leq \frac{n}{2} - 1$, such that $\sum_{i=1}^k g(u_i, v_i)$ is maximised.

The algorithm exchanges the sets of vertices X , and Y . This is one iteration of the Kernighan-Lin algorithm. The full Kernighan-Lin algorithm repeats this process on the derived partition until no further improvement is possible.

The algorithm described above, is a graph bisection algorithm, that is, it is designed for splitting the graph into 2 disjoint sets of vertices. The algorithm may also be used to perform k -way partitioning of graphs. To do this, the graph is first partitioned into k random disjoint sets. The Kernighan-Lin algorithm is then applied to pairs of sets in order to achieve pairwise local optimality. This phase must be applied a number of times since when two sets are made locally optimal, their local optimality with respect to other sets may be disrupted. Note also, that even global pairwise optimality is not necessarily global optimality. A more detailed description of the process is given in [3].

2.4 The Kernighan-Lin Algorithm as a Local Improvement Heuristic

The hybrid genetic algorithm (see section 3.4) proposed by Bui and Moon [2], uses a local improvement heuristic based on the Kernighan-Lin algorithm. In order to speed up the local optimisation phase, they propose a number of modifications. Firstly, the proposed algorithm makes only one pass. Additionally, while the Kernighan-Lin algorithm swaps the amount of vertices which maximises the reduction in cost, the proposed algorithm limits the number of vertices which may be exchanged. This limit is a parameter set by the user. The proposed algorithm for 2 way partitioning is shown in figure 2.2. In order to perform k -way partitioning, this phase must be repeated for each pair of partitions.

```

Compute  $g_u, g_v$  for each  $u \in V_1, v \in V_2$ 
Make two gain lists of  $g_u$  and  $g_v$ 
 $Q_u = null; Q_v = null$ 
for  $i = 1$  to  $MaxExchangeSize$ 
do
  Choose  $u_i \in V_1 - Q_u, v_i \in V_2 - Q_v$  such that
   $g(u_i, v_i)$  is maximal.
   $Q_u = Q_u \cup \{u_i\}; Q_v = Q_v \cup \{v_i\}$ 
  for each  $u \in \{V_1 - Q_u\}$  adjacent to  $u_i$  or  $v_i$ 
  do
     $g_u = g_u + 2 \times \delta(u, u_i) - 2 \times \delta(u, v_i)$ 
    if  $g_u$  changed, adjust gain list of side  $V_u$ 
  end for
  for each  $v \in \{V_2 - Q_v\}$  adjacent to  $u_i$  or  $v_i$ 
  do
     $g_v = g_v + 2 \times \delta(v, v_i) - 2 \times \delta(v, u_i)$ 
    if  $g_v$  changed, adjust gain list of side  $V_2$ 
  end for
end for
Choose  $k \in \{1, \dots, MaxExchangeSize\}$  to maximise  $\sum_{i=1}^k g(u_i, v_i)$ 
Swap subsets  $\{u_1, \dots, u_k\}$  and  $\{v_1, \dots, v_k\}$ 

```

Figure 2.2: Local Improvement Algorithm for GA

Chapter 3

Global Optimisation Methods for GPP

3.1 Simulated Annealing

Simulated Annealing is an approach to finding a solution to a combinatorial optimisation problem. The algorithm works by modifying the current solution. If the modified solution is of lower cost then it is accepted, if it is of higher cost then it is accepted with a certain probability. This probability depends on two factors, the extent of the increase in cost and, a parameter called the *temperature*. In this report, accepting a solution of higher cost is referred to as an *uphill move*. Uphill moves are more likely to be accepted at high temperatures than at low temperatures thus the temperature of the simulation is analogous to the *energy* of a physical system.¹ The temperature is decreased periodically and thus, with it, the probability that uphill moves are accepted. The basic algorithm for simulated annealing is shown in figure 3.1.

3.2 Simulated Annealing and GPP

The simulated annealing algorithm proposed by Johnson et al. [1], uses a search technique similar to the local search algorithm described in section 2.1, that is, given a solution S , it explores $\mathcal{N}(S)$. The local search algorithm is guaranteed to stop at the first local minimum encountered, and so, the resultant solution depends on the initial solution. By modifying $\mathcal{N}(S)$ to include solutions of higher cost, which are accepted probabilistically, simulated annealing attempts to avoid the problem of premature convergence to a local minimum. The algorithm can explore a greater range of solutions and so, it should produce a better solution than the local search algorithm.

Simulated annealing may be used to apply single vertex migrations or double vertex

¹The simulated annealing algorithm is based on an analogy with the process of physical annealing, that is, slow cooling, to produce large regular crystals.

```

T = InitialTemperature
Solution S = new Solution()
while (not frozen()) do
  do L times
    S' = neighbour(S)
    Δ = cost(S') - cost(S)
    if (Δ ≤ 0) (downhill move)
      S = S'
    if (Δ > 0) (uphill move)
      S = S' with probability exp -Δ/T
  od
  T = rT (0 < r < 1)
od
return(S)

```

Figure 3.1: Simulated Annealing Algorithm

migrations to GPP (see section 1.1). If double vertex migrations are applied then, the distribution of vertices in each partition does not change as a result of applying the algorithm. Thus an evenly balanced initial partition will result in an evenly balanced final partition. If single vertex migrations are applied, then the vertices are redistributed in order to optimise the cost of the final partitioning. This may result in a final solution in which the vertices are unevenly distributed. The degree to which the final solution may be unbalanced is regulated by the penalty parameter, (see eqn 1.1). Setting the penalty parameter to a low value is likely to lead to an unbalanced solution, whereas setting it to a high value is likely to lead to a balanced solution. However, setting the penalty parameter high restricts the search capability of the algorithm and can lead to solutions of high cost.

One potential means of using the single vertex migration algorithm to produce high quality, balanced partitions is to apply two phases of optimisation to the partition. In the first phase of optimisation, the algorithm is applied with the penalty parameter set at a low value. This produces a low cost but, potentially unbalanced solution. The algorithm is subsequently applied with the penalty parameter set to a very high value. This results in a balanced partitioning and does not diminish the searching ability of the algorithm. An alternative second phase may consist of a redistribution of vertices based on the cost of the vertices to be migrated.

One solution that we propose, is to increase the penalty parameter as the algorithm progresses, thus allowing the algorithm to explore a large portion of the search space initially. As the algorithm progresses, unbalanced solutions are penalised to a greater extent, thus it is expected that the algorithm will produce a good, balanced solution.

3.3 An Overview of Genetic Algorithms

Genetic Algorithms (GAs) are adaptive methods which may be used to solve search and optimisation problems. They are based on the genetic process of biological or-

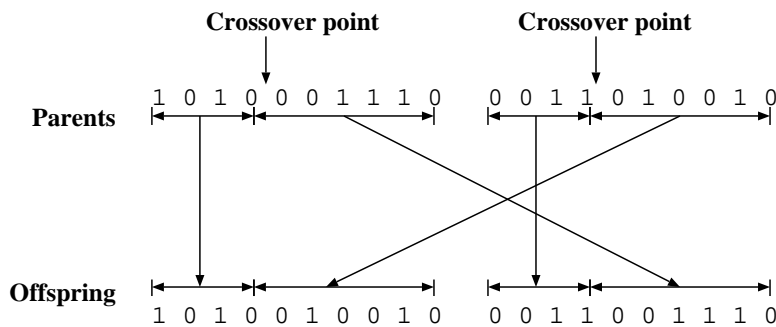


Figure 3.2: Single point crossover

```

Create Initial Population
while (not finished)
do
    choose parent1 and parent2 from population
    offspring = CrossOver(parent1, parent2)
    offspring.mutate()
    Population.replace(offspring)
od
return(population.bestSolution())

```

Figure 3.3: A typical genetic algorithm

ganisms. A genetic algorithm starts off with a set of solutions (called *chromosomes*²) which comprises the *population*. These solutions are coded as strings of symbols, hence the use of the term chromosome. In order to implement a GA, a *fitness* function must be defined. A *fitness* function maps a chromosome onto a single numeric value which indicates the *quality* of the solution it represents. The fitness of a solution determines the probability that that solution will be chosen as a parent, with solutions of high fitness being more likely to be chosen than solutions of low fitness.

In addition, two operations must be defined, these are as follows:

Crossover This operator takes two chromosomes and splits up the chromosome strings to produce two *head* and two *tail* segments. The tail segments are swapped to produce two new solutions or *offspring*. This is known as *single point crossover* and is illustrated in Figure 3.2. When the chromosomes are split into more than two segments, the process is referred to as *multipoint crossover*.

Mutation This is a random adjustment applied to a chromosome in order to some provide local searching. It also helps to ensure that no point in the search space has a zero probability of being explored.

A replacement algorithm determines whether the new solution should replace an existing solution in the population and if so, which solution should be replaced. A simple replacement scheme might be to place a solution into the population if it is

²The terms *solution* and *chromosome* are used interchangeably in this report.

```
graph.preprocess()
Create Initial Population
while (not finished)
do
    choose parent1 and parent2 from population
    offspring = CrossOver(parent1, parent2)
    offspring.mutate()
    offspring.localOptimise()

    If (offspring.suited())
        Population.replace(offspring)
od
return(population.bestSolution())
```

Figure 3.4: The genetic algorithm for GPP

better than the worst member of the population, and remove the worst member. See [2] for a more detailed discussion on replacements schemes for genetic algorithms. The typical structure of a genetic algorithm is shown in figure 3.3.

3.4 A Hybrid Genetic Algorithm for GPP

A genetic algorithm may be combined with another optimisation technique. Such an algorithm is known as a hybrid genetic algorithm. Bui and Moon [2] propose a hybrid genetic algorithm for finding solutions to GPP. The proposed algorithm applies a fast local improvement heuristic to the offspring before performing the replacement operation. This local improvement phase is based on the Kernighan-Lin algorithm as described in section 2.3. Additionally the proposed algorithm includes a problem specific, preprocessing phase called breadth first search (BFS) re-ordering.³ This consists of renaming the vertices of the graph to be partitioned in the order in which they are encountered by a breadth first search of the graph, starting from a random vertex. This speeds up convergence of the algorithm without impacting the quality of the solution [2]. The algorithm is described in figure 3.4. Each iteration through the while loop is referred to as a generation.

³The graph is assumed to be connected.

Chapter 4

Design & Implementation

The language used to implement the algorithms described, was Java. Java, as an object-oriented language, offers the typical advantages associated with such languages, that is, encapsulation, data hiding, and ease of code re-use. In addition to this, Java offers a number of advantages to a programmer over various other object-oriented languages, for example, C++. Java is *garbage collected*, that is, the programmer does not have to explicitly allocate and deallocate memory. Additionally, Java can perform subscript checking of arrays which reduces the chances of memory leakage. Java has greater support for exception handling (all exceptions must be caught in Java), which provides a uniform way of processing errors. These features tend to speed up the development process and reduce the number of errors in the code. Conversely, the run time checking performed by the Java interpreter results in slower execution than code developed in compiled languages, for example C++.

Another feature of Java is that it is *architecture neutral*. All primitive data types are defined explicitly by the Java *virtual machine*. This means that there is no need to port code for use on different hardware architectures. The code for this project was developed mainly on a Sun SPARCstation5, while the results obtained were run on a SGI O₂. The code was migrated without recompilation.

4.1 A Description of the Data Classes used in GPP

The classes used in the development of the graph partitioning methods may be divided into *Algorithmic* classes and *Data* classes. The algorithmic classes, consisting of SA, GA, LocalOpt, and KerLin have no public methods apart from their constructors, and *main*. In each case, *main* parses the users' input and calls the classes' constructor with this input. The constructors invoke the algorithm which actually performs the graph partitioning. Thus the algorithmic classes implement the high-level graph partitioning algorithms. The data class are Graph, Solution, and Population. Figure 4.1 shows the set of objects used, as well as the relationships between them. The methods of these classes are described in more detail below.

This section lists and describes the methods associated with the classes Graph, Solution and Population.

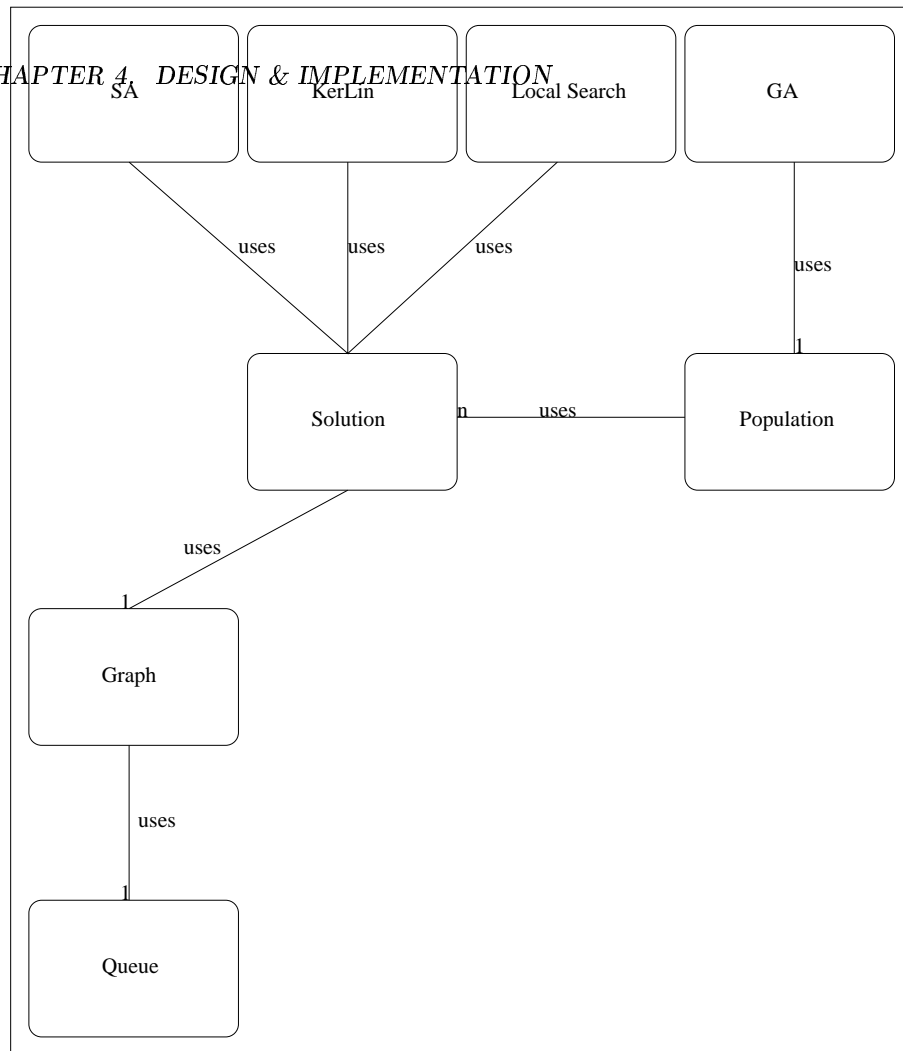


Figure 4.1: The object model for classes in GPP

4.1.1 The methods associated with class Graph

addEdge: Adds an edge to the graph.

bfsReorder: Breadth First Search (bfs) reordering of the graph is carried out in order to enhance the performance of the genetic algorithm. BfsReordering consists of renaming the vertices of the graph so that the index of a vertex indicates the order in which it was visited by a breadth first traversal starting from a random position within the graph. The result of this reordering is that local clustering in the graph corresponds to local clustering in the vector representing the partitioning of the graph. See [2] for a more detailed discussion of BFS reordering in graph partitioning.

generateEdges: This is used to generate edges at random on a graph. If the graph has size n and the input parameter has value k , where $0 < k, k < 1$, then n points on a 2-dimensional square grid of length 1, are chosen at random. For each point, an edge is created between it, and all other vertices whose Euclidean

distance away is less than or equal to k . (A graph generated in this way is referred to as a *geometric* graph.)

getOrder: Returns the *order* or the number of vertices of the graph.

getSize: Returns the *size* or the number of edges of the graph.

hasEdge: Returns boolean indicating if the input edge exists.

IsConnected: Returns boolean indicating if the graph is connected. This is determined by means of a breadth first traversal of the graph. If all vertices are visited then the graph is connected. If any vertices are not visited by the breadth first traversal, then they are unreachable, i.e the graph is disjoint.

printEdges: This prints the edges of the graph to the screen as pairs of vertices in the form (i,j). This was used for consistency checking.

readFromFile: This method reads a persistent (serialised) version of the class from a file. This file must have been generated as a result of using the *Serialize* method of an instance of the class Graph.

readMatrixFromFile: This method creates a graph by reading a text representation of an adjacency matrix from a file.

write: This writes the adjacency matrix to the screen. This was used for consistency checking.

writeToFile: This method serialises the class and writes it to a file specified by the user.

4.1.2 The methods associated with class Solution

The class Solution consists of a Graph and a vector representing the partitioning of the Graph. For efficiency reasons, the Graph is a static data member, that is, all instances of Solution access the one instance of Graph. A k -way partitioning of a graph of order n may be represented as a vector of length n with each element containing an integer in the range $[0, k - 1]$. If the i^{th} element contains the value p , then this represents the vertex i being in partition p .

clone: Creates a copy of itself. Recursively *clones* all data members.

getCost: Returns the number of edges cut + the imbalance of the partition.

getCutEdges: Returns the number of edges cut.

getImbalance: Returns the sum of the squares of the differences between the size of the partitions and the average size.

getPartition: Returns the vector representation of the partitioning.

getVertexPartition: Returns the partition to which the input vertex is currently assigned.

legalMigration: Moves a single vertex to a different partition.

- randomLegalMigration:** Performs *legalMigration* using a randomly generated vertex as input.
- mutate:** Mutates a block within the vector representation of the partitioning.
- postProcess:** Performs a random adjustment on the vector to *balance* the partitioning, that is, to allocate the same number of vertices to each partition.
- singlePointCrossOver:** Creates 2 new solutions resulting from combining the vectors as illustrated in 3.2
- randomSinglePointCrossOver:** This performs *singlepointCrossOver* using a randomly generated value as input.
- printStats:** Prints status information to the screen. Used for consistency checking.
- readFromFile:** Reads a serialised version of a Solution from a file.
- writeToFile:** Serialises the class and writes to a file specified by the user.
- setImbalanceWeight:** Sets the scaling factor for the imbalance of the partitioning (see 1.1).
- setPartition:** Sets the vector representing the partitioning to be the input value.
- switchVertices:** Switches the partition assignment of the two input vertices.
- switchRandomVertices:** Performs *switchVertices* on two random vertices from different partitions.

4.1.3 The methods associated with class Population

The *Population* class is a container class for *Solutions*. The Solutions are stored in a vector and are ordered based on their cost. The Population may not contain multiple occurrences of the same solution, thus successful replacement into the population is conditional on the Solution being of better quality than the worst Solution in the Population, and on the Population not already containing the Solution. The size of the population is limited by a user parameter passed to the constructor. When the population reaches this size, all subsequent additions to the population cause the worst element to be removed. The public methods to the Population class are as follows:

- replace:** This method adds the input Solution to the Population if
- The Solution is of lower cost than the worst Solution in the Population.
 - The Solution is not already contained in the Population.

If the above conditions hold, then the Solution is inserted into the vector according to its' ordering and, if necessary, the worst Solution is removed.

- size:** This returns the number of Solutions currently contained in the Population.

fitness: Each Solution in the population has a *fitness* value. This is based on the relative cost of the Solution. The fitness value F_i of a Solution i is calculated as follows:

$$F_i(C_w - C_i) + (C_w - C_b)/3 \text{ where}$$

- C_w Is the cost of the worst Solution.
- C_b Is the cost of the best Solution.
- C_i Is the cost of the Solution i .

This means that the best Solution is 4 times more likely to be picked than the worst Solution.

getParents: This returns two distinct Solutions from the Population. Solutions are picked with a probability proportional to their fitness value. This is accomplished as follows:

- Find the sum of the fitnesses of all the Solutions in the Population.
- Pick a random number less than this value.
- Traverse the list of Solutions incrementing a counter with each Solutions' fitness until the counter exceeds the random number.
- The Solution at which the traversal stops is the required Solution.

- This Solution and its' successor are returned.

getBestSolution: Returns the best Solution in the Population.

getWorstSolution: Returns the worst Solution in the Population.

getBestSolutionCost: Returns the cost of the best Solution in the Population.

getWorstSolutionCost: Returns the cost of the worst Solution in the Population.

Retrieval of parents and insertion into the population, both require a linear search. Thus as the population increases, both of these operations become more costly. The increase in time is linear with respect to the increase in population.

4.2 A Description of the Algorithmic classes used in GPP

4.2.1 Local Search

The code for the local search algorithm is shown in appendix A.7. The loop terminates when all vertices have been tested against all partitions without any combination bringing about a reduction in cost.

The local search function accepts the following parameters

Graph: The graph to be partitioned.

k: The number of partitions into which the graph should be divided.

weight: The weighting to be given to the partition imbalance.

Solution: The name of a file containing a Solution. This parameter is optional and if it is not supplied, a Solution is generated at random.

4.2.2 Simulated Annealing

The simulated annealing function accepts the following parameters

Graph: The graph to be partitioned.

k: The number of partitions into which the graph should be divided.

InitialProbability: This is the initial probability that moves will be accepted. The initial temperature (T_0) is calculated using this value (see below).

CoolingRate: This is the rate at which the temperature is reduced.

SizeFactor: The run length L is the number of migrations performed before the temperature is reduced. $L = SizeFactor \times GraphOrder$.

minRatio: If the ratio of accepted moves is less than *minRatio* for m successive iterations then the algorithm halts.

maxIt: This determines the parameter m as above.

weight: The weighting to be given to the partition imbalance.

Solution: The name of a file containing a Solution. This parameter is optional and if it is not supplied, a Solution is generated at random.

The code for simulated annealing is shown in figure A.12. The loop proceeds through 2 phases. Initially it performs 1000 random migrations. After each migration, the change in cost is recorded. Let Δ denote the average change in cost after 1000 migrations. The initial temperature T_0 is calculated as $T_0 = \log P / -\Delta$. The second phase of the loop performs the main graph partitioning functions. The percentage of accepted moves is sampled after every l moves where $l = runlength \times n$, where n is the order of the graph. The loop terminates when the percentage of migrations accepted is below *minRatio* for *maxIt* consecutive iterations.

4.2.3 Genetic Algorithm

The code shown in appendix A.4 shows the main loop executed by the genetic algorithm. The arguments to the main method are as follows:

Graph: The graph to be partitioned.

k: The number of ways the graph is to be split.

populationsize: The size of the population to be created.

maximumiterations: If the number of consecutive iterations in which no replacement occurs reaches *maximumiterations*, the algorithm halts.

maxKerLinIterations: This is the maximum number of vertices which may be switched in an iteration of the Kernighan-Lin optimisation phase.

4.2.4 The Kernighan-Lin Algorithm

The arguments to the KerLin class which implements the Kernighan-Lin algorithm are as follows:

Graph: The graph to be partitioned.

k: The number of [partitions into which the graph should be split.

Solution: This argument is optional. If a Solution is not supplied then a Solution is generated at random.

[**-f|q**] This switch specifies whether the full Kernighan-Lin algorithm is to be applied or if just one iteration is to be applied (quick optimisation).

The code for this is shown in appendix A.6

4.3 Testing Strategy

The object-oriented nature of Java allows for isolated unit testing as a first round of system testing. All classes in Java can contain a *main* function. Initially this was used to perform checks on all the class' methods. All classes have methods to indicate the internal state of the object. Additionally, the main method has access to the class' private data members so that values returned by methods can be checked for consistency with the state of the object. The *main* method in class Queue performs this function (see appendix A.11). Functional testing, that is, the ability of the algorithm to partition a graph, was provided by applying the algorithms to a graph of known minimum cost. The graph was created using the *CreateGraph* utility. See section 4.4 for a description of this. The test graph consisted of two random graphs joined by a single edge thus, the solution for a 2-way partitioning, is 1.

4.4 Tools and Utilities

The algorithmic classes (see section 4.2) provide the user with a command line interface to the graph partitioning algorithms. Invoking the algorithms from the command line causes one application of the algorithm to be performed, with the output being directed to *stdout*. A number of Bourne Shell scripts were developed in order to automate the process of running the algorithms under varying initial parameters. Additionally, Bourne Shell and Awk were used to create scripts for filtering and summarising the data. For the most part, graphs were generated randomly and stored in files (see section 4.1.1) however, a graph generating utility was also created. This is an interactive tool which allows the user to create a graph and add edges from a command line prompt. It also allows the user to inspect the current status of the graph, (i.e find out the current size, get a listing of all the edges etc.). This was written in Java (see appendix A.14 for a source code listing.).

Chapter 5

Results

The analysis of the genetic algorithm and of simulated annealing proceeded through the following phases:

- Analysis of simulated annealing with different parameter settings. Results are compared on the basis of
 - Running time of the simulation.
 - The cut size of the resulting solution.
- Analysis of the genetic algorithm with different parameter settings.
- Comparison of simulated annealing with the local search algorithm.
- Comparison of simulated annealing with the Kernighan-Lin algorithm.
- Comparison of simulated annealing with the genetic algorithm.
- Comparison of the performance of simulated annealing with the single vertex migrations and with double vertex migrations.

5.1 Parameters of Simulated Annealing

The following parameters were investigated with respect to the cost and the final running time of the algorithm:

- `coolingrate` This is the cooling ratio as described in section 3.1
- `runLength` This determines the frequency with which the temperature is decreased. If the graph has size s and `runLength` has value r , then the temperature is reduced after every $r \times s$ attempted migrations.
- `freezePoint` This is used to determine when the system is frozen. A counter is maintained and incremented every time a run is completed for which the ratio of accepted moves is `minRatio`, or less. The counter is reset if the acceptance

rate of a run is greater than `minRatio`. When the counter reaches a maximum value, the algorithm halts. Johnson et al. [1] set this value to be 5, however this is set as a user parameter in this implementation.

- `alpha` This is the weighting that is applied to the imbalance of a partitioning (α in eqn (2) in section 1.1).
- `initprob` This is used to determine the initial temperature of the system. Based on a trial run of 1000 iterations, a temperature is found at which the fraction of accepted move is approximately *initprob*.

Simulated annealing was applied to a *geometric* graph (see section 4.1.1) of order 480 and of size 70,802 using single vertex migrations. Results were obtained for 2, 5, 8, 10, 16 and 20 way partitioning. The results shown, are the average over 20 runs of each configuration. In each case, the same initial solution is used. The following sections detail the results obtained and the conclusions that may be drawn from the study.

5.1.1 The Imbalance Scaling (α)

The imbalance scaling (α) determines the extent to which unbalanced partitions are penalised. The graphs in figure 5.1.1 show the effects of varying α on the running time of the simulation, and on the quality of the solutions produced. The simulation used single vertex migrations (see section 3.2), thus the partition is not guaranteed to be partitioned. In most cases, a balanced partitioning was obtained when α was set to be 1 or higher. Figure 5.1.1 shows that the solution degrades significantly as α increases. The running time increases very sharply initially. In all the experiments conducted, the running time reduced slightly when α reached 2, and increased slowly thereafter. Based on observation of the graphs under test, 2 would seem to be the optimum value for α for balanced partitioning of this graph. If the constraint of balanced partitioning is relaxed, then a significant reduction in running time is gained. The reduction in cut size is generally not significant.

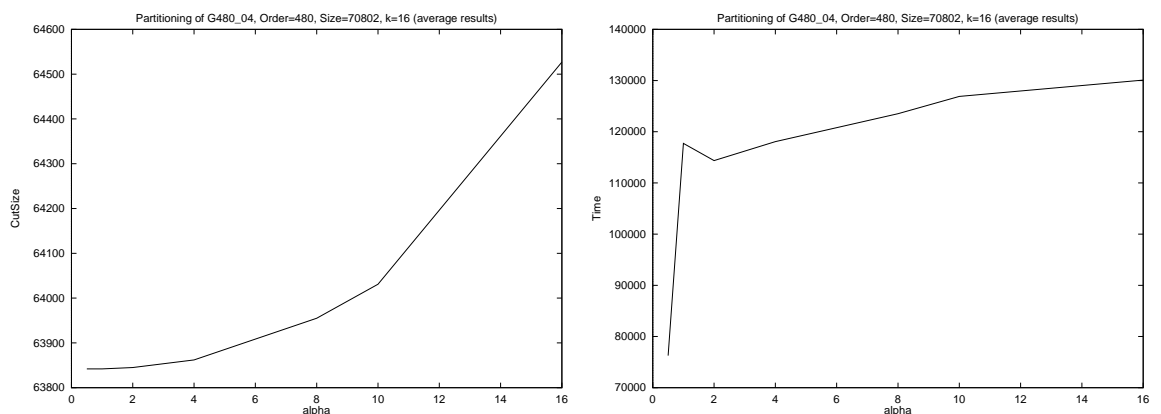


Figure 5.1: The effect of α on the solution quality and running time.

5.1.2 The Cooling Rate

The cooling rate determines the rate at which the temperature is reduced. The graph in figure 5.1.2 shows the effect on the cut size of the solution, as the cooling rate is varied. The results shown are for a 16-way split of the graph. The average over 20 runs is plotted. In most cases, increasing the the cooling rate (i.e slowing down the cooling process) resulted in better solution. Occasionally, increasing the cooling rate has reduced the solution quality, for example, the *spike* at 0.93 in figure 5.1.2. This may be attributed to the fact that at high temperatures, the algorithm is less likely to converge to a minimum. Thus the algorithm may fail to converge to a particular solution in the initial (high temperature) phase, and may not locate it at a later stage. The effect of the cooling rate on the running time is shown also in figure 5.1.2. Running time increases rapidly as the cooling rate increases.

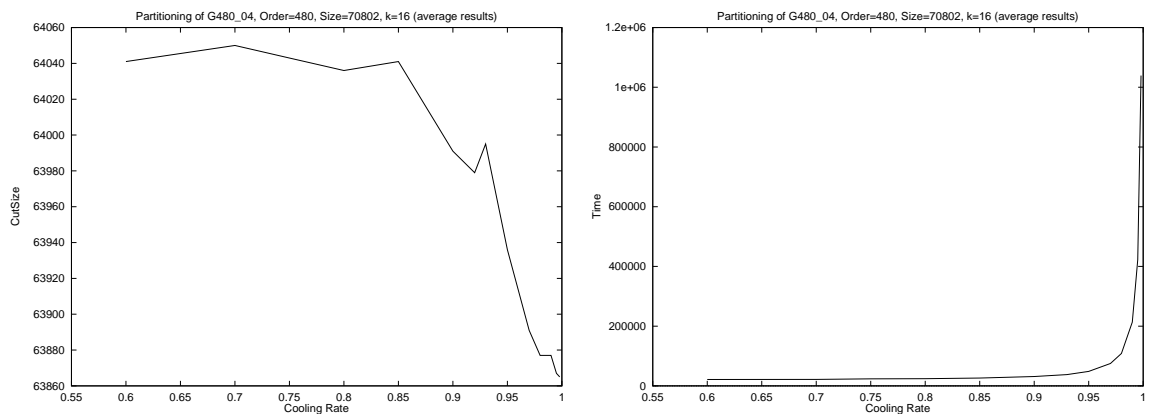


Figure 5.2: The effect of *Cooling Rate* on the solution quality and running time.

5.1.3 Initial Probability

The initial probability (*initprob*) determines the initial probability of acceptance of a move, that is, the initial temperature. Figure 5.1.3 shows the average results obtained for a 16-way split, over 20 runs. The first graph shows the effects of varying *initprob* on the solution cost. The second graph shows the running time of the algorithm. In all cases, the solution cost was found to have no apparent relationship with the initial probability. The running time was found to be approximately linear with respect to the initial probability.

5.1.4 Freeze Point

The freeze point determines when the annealing run is frozen and hence, when it should stop. When the fraction of accepted moves is less than *freezept* for a specified number of consecutive iterations, the simulation halts. Figure 5.1.4 shows the impact of this parameter on the solution cost and running time of the algorithm. A low value for *freezept* results in better quality solutions but increases the running

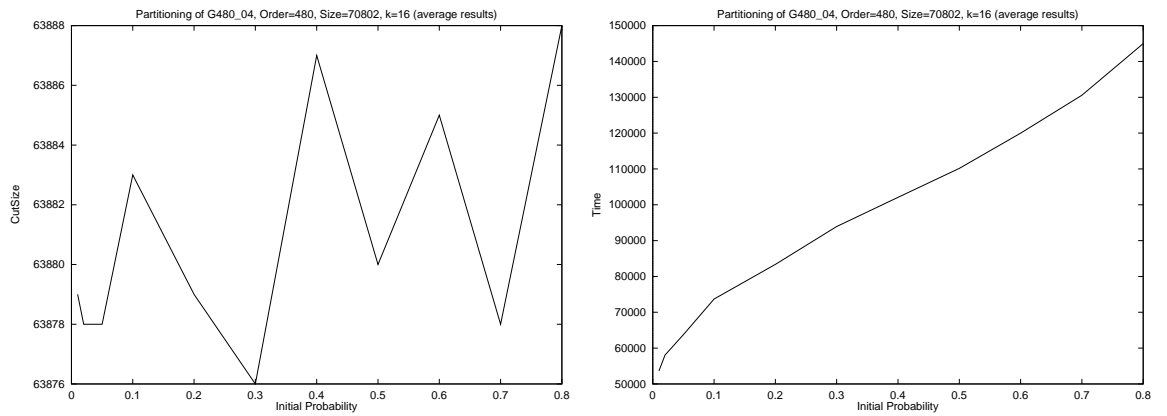


Figure 5.3: The effect of *Initial Probability* on the solution quality and running time.

time. The graph of solution quality in figure 5.1.4 is decreasing, that is, a decrease in freeze point does not reduce the quality of the solution.

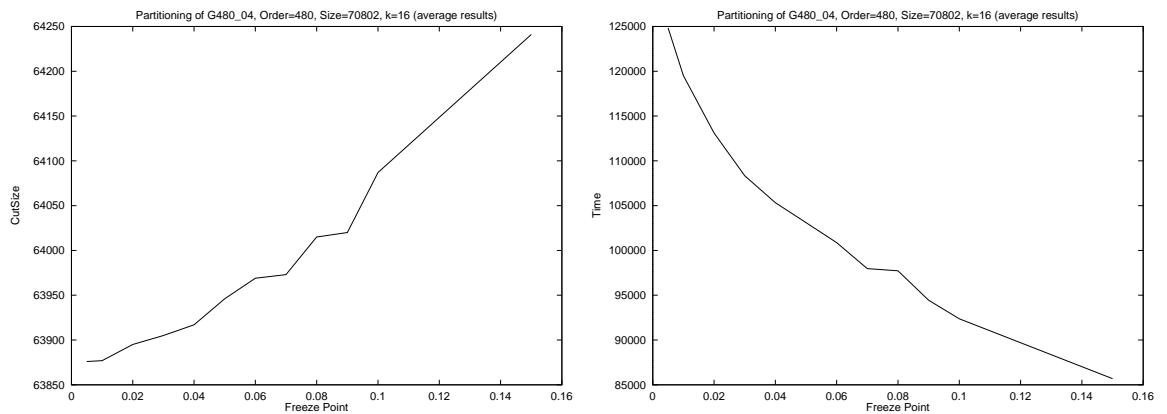


Figure 5.4: The effect of *Freeze Point* on the solution quality and running time.

5.1.5 Run Length

The run length determines the frequency with which the temperature is reduced (see section 4.2.2). The results are shown in figure 5.1.5. In all cases increasing the run length yielded a solution of better quality although, in most cases, increasing the run length beyond 2 did not yield a significant increase. The variation in running time was approximately linear.

In conclusion, in most cases, increasing the running time of the simulation, yields solutions of lower cost. In the case of the cooling rate parameter, the reduction in cost is gained at the expense of rapidly increasing running time. Additionally, this reduction in cost is not guaranteed to occur, and the solution quality may even degrade as a result of increasing this parameter. In the cases of the freeze point and run length parameters, the solution quality does not degrade as a result of lengthening

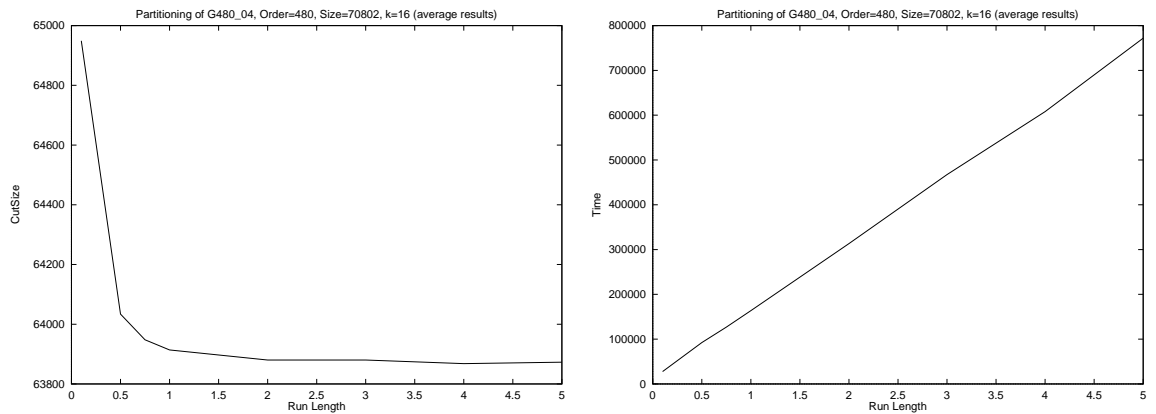


Figure 5.5: The effect of *Run Length* on the solution quality and running time.

the simulation time. The variation in running time with respect to the parameters is approximately linear in these two cases. Thus, it may be preferable to improve results by lengthening the running time using the runlength and freeze point parameters than by increasing the cooling rate. If the constraint of balanced partitioning is relaxed, then the simulation time is significantly reduced. The number of cut edges is not significantly reduced. The *initprob* parameter was found to have no direct correlation with the cost of the resulting solution.

5.2 Parameters of Genetic Algorithm

The following parameters were investigated with respect to their impact on the cost of the final solution produced, and on the running time of the algorithm:

- *populationSize* The size of the population to be used.
- *maximumIterations* If the algorithm fails to produce an improvement in the population *maximumIterations* times in a row, then the algorithm halts and the best member of the population is reported as the solution.
- *maximumKerLinIterations* This is the maximum amount of iterations which may be performed by the Kernighan-Lin local optimisation technique, that is, the number of vertices which it may swap.

In the following results, GA was applied to a geometric graph of order 480, and of size 72,696. The graph was split into 2, 5 and 10 partitions.

5.2.1 The Population Size

The population size is the number of solutions used by the GA (see section 3.3). A larger population should increase the chances of the algorithm finding the global minimum however, as the population increases, the time taken to retrieve parents from the population, and to insert new solutions into it, increases in a linear manner (see

section 4.1.3). The graphs in figure 5.2.1 show the effects of increasing the population size with respect to the running time of the algorithm, and the cost of the final solution produced.

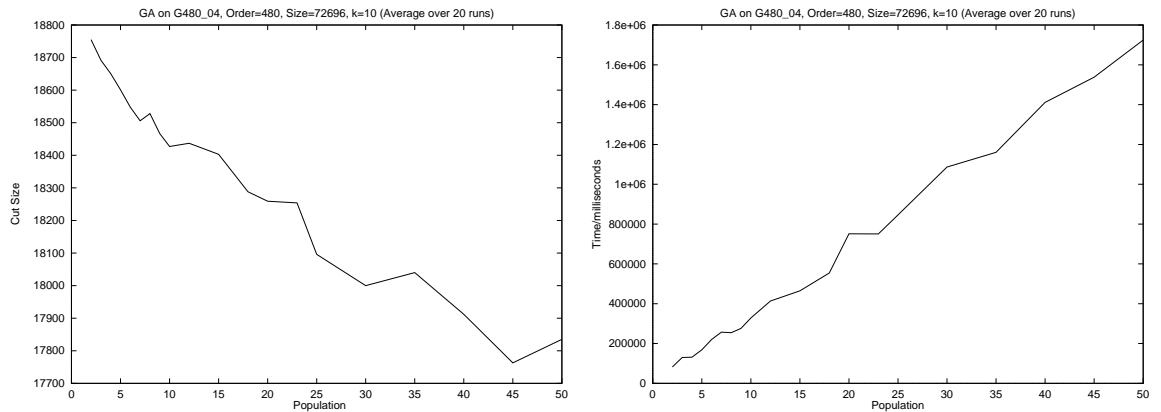


Figure 5.6: The effect of *Population Size* on the solution quality and running time.

5.2.2 Maximum Iterations

This parameter is used to determine when the algorithm should halt. After each generation, (see 3.4), if neither offspring solution is inserted into the population, a counter is incremented. If a solution is successfully inserted into the population, the counter is reset to zero. If the counter reaches the value specified by the maximum iterations parameter, the algorithm halts. The graphs in figure 5.2.2 show the effects of the parameter on the running time and solution quality of the algorithm. In general, high quality solutions are observed to be more likely to occur when the parameter is set high, although increasing the parameter does not guarantee that higher quality solutions will be obtained and, could diminish the solution quality significantly. The running time of the algorithm tends to increase as the parameter increases although, as with the solution cost, increasing the running time does not always increase when the parameter is increased.

5.2.3 Kernighan-Lin Iterations

GA uses a local optimisation technique based on the Kernighan-Lin algorithm (see section 2.4). The algorithm described in section 2.4 accepts parameter which limits the number of vertices which may be exchanged. The graphs in figure 5.2.3 show the impact of this parameter on the performance of GA. The parameter was found to have no direct influence on the running time of the algorithm, or on the cost of the final solution produced. One explanation for this is as follows: Reducing the number of vertices which may be swapped does not reduce the set of solutions which may be obtained by the algorithm. A reduction in this value may be compensated for by applying the algorithm multiple times to the solution. This should lengthen the execution time. Conversely, if the local optimisation phase is diminished, the probability of an offspring being accepted into the population is reduced. Thus the

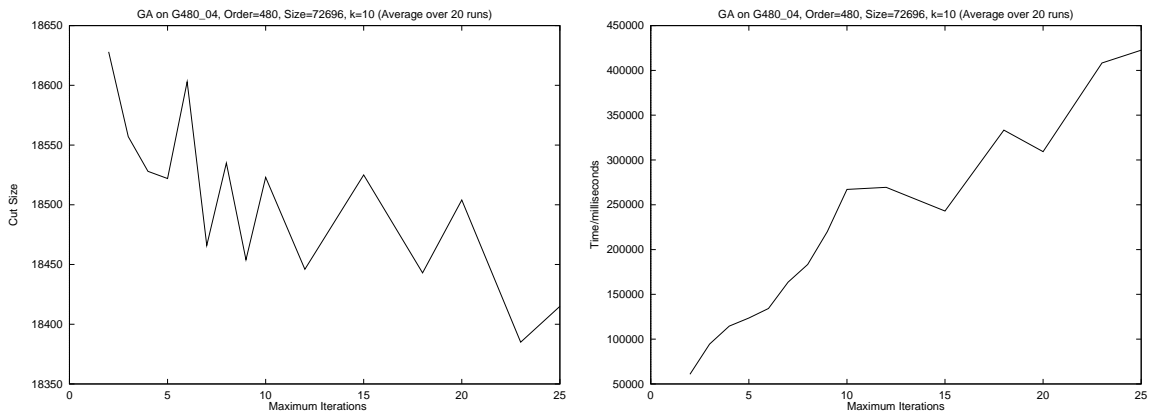


Figure 5.7: The effect of *Maximum Iterations* on the solution quality and running time.

system will appear to be frozen at an earlier stage, thus reducing the running time of the algorithm.

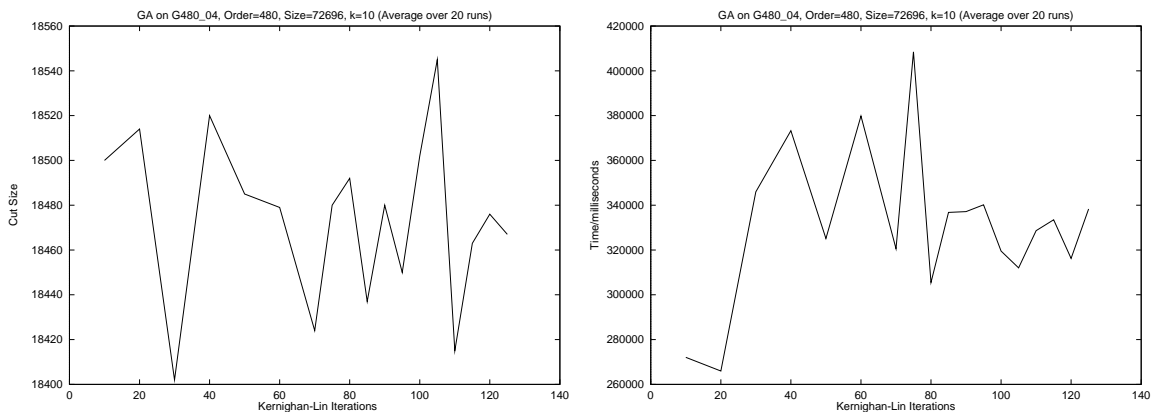
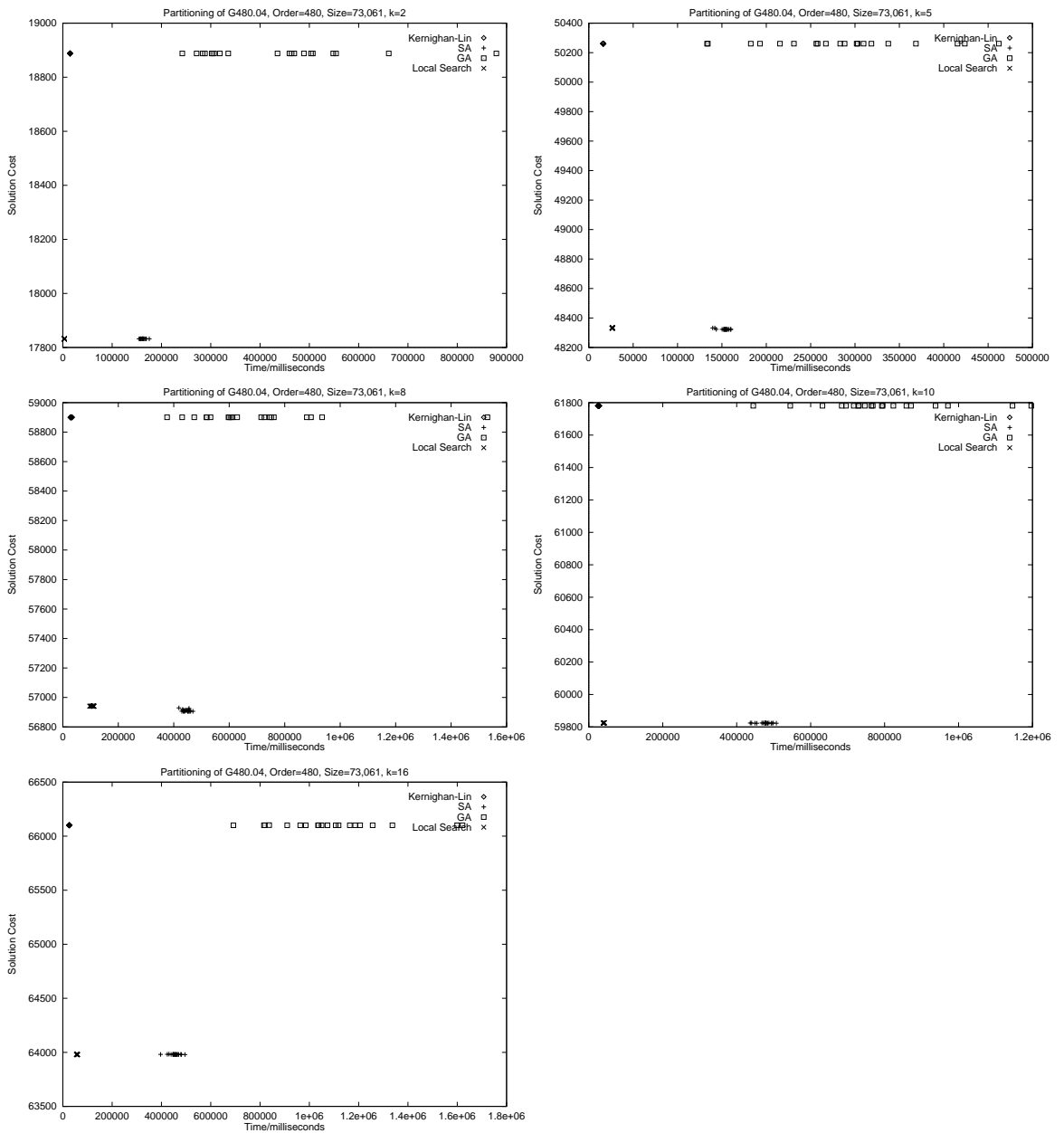


Figure 5.8: The effect of *Kernighan-Lin Iterations* on the solution quality and running time.

5.3 Comparison of GA, SA, Kernighan-Lin, and Local Search

Simulated annealing, genetic algorithm, Kernighan-Lin and local search were all applied to a geometric graph of order 480 and size 73,061. The following graphs indicate their respective performance with respect to the cut size of the solution obtained, and the time taken by the algorithm to achieve that solution. The algorithms were applied to 2, 5, 8, 10, and 16 way partitioning of the graph.



In general, simulated annealing yielded the best solutions. In most of the experiments performed, GA failed to improve on the solution produced by the Kernighan-Lin algorithm. In most cases, the Kernighan-Lin algorithm was the first to converge although the local search algorithm had the shortest execution time for 2-way partitioning (approx. 3 seconds). In conclusion, simulated annealing tended to provide the best solutions although, the local search algorithm produced solutions that were nearly as good and ran for a considerably shorter time. It is possible that GA would perform better if the population size were to be increased (see section 5.2.1). This should be investigated further.

5.4 Simulated annealing with single & double migrations

Simulated annealing may explore the solution space by picking two vertices which are not in the same partition, and swapping those vertices. Alternatively, it may pick a single vertex at random, and move it to a different partition. If the initial partition is balanced, and double vertex migrations are used, then the resulting partition is guaranteed to be balanced and so, there is no need to calculate the imbalance of the partition. Alternatively, the algorithm may perform single vertex migrations which are quicker to perform but which involves the overhead of calculating the partition imbalance. The following graphs show the relative performance of the simulation using single and double vertex migrations. The results shown are the average results for 5-way partitioning, obtained over 20 runs. Results are compared over a range of values for the parameters *Cooling Rate*, *FreezePoint*, *Initial Probability*, and *Size Factor* (run length). These results are shown in figures 5.4 to 5.9. In all cases, the single vertex migrations took longer to reach a balanced partition. The only case where double vertex migration yield a better solution than single vertex migrations is shown in figure 5.4. The solutions produced by double vertex migrations are of better quality when the cooling rate is below 0.4. In conclusion, it was found that using single vertex migrations yields better solutions but takes longer to converge.

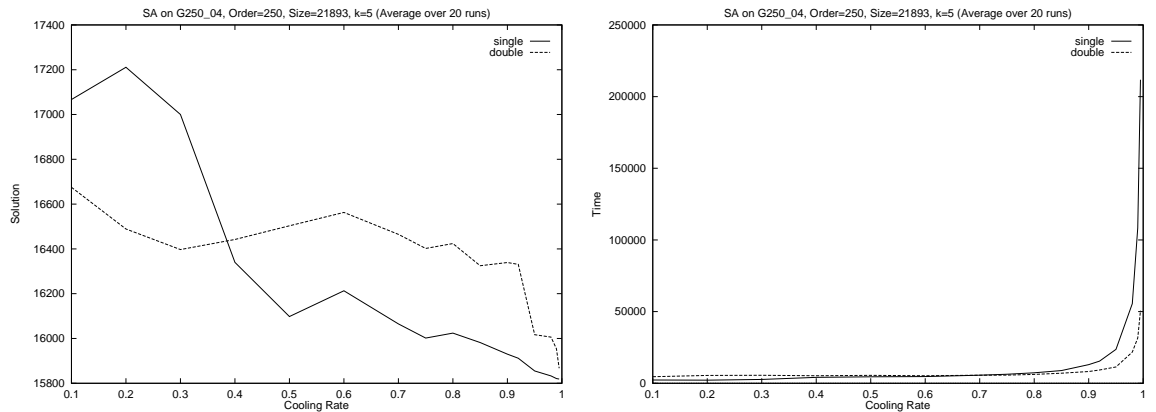


Figure 5.9: Comparison of single and double vertex migrations for different values of *Cooling Rate*

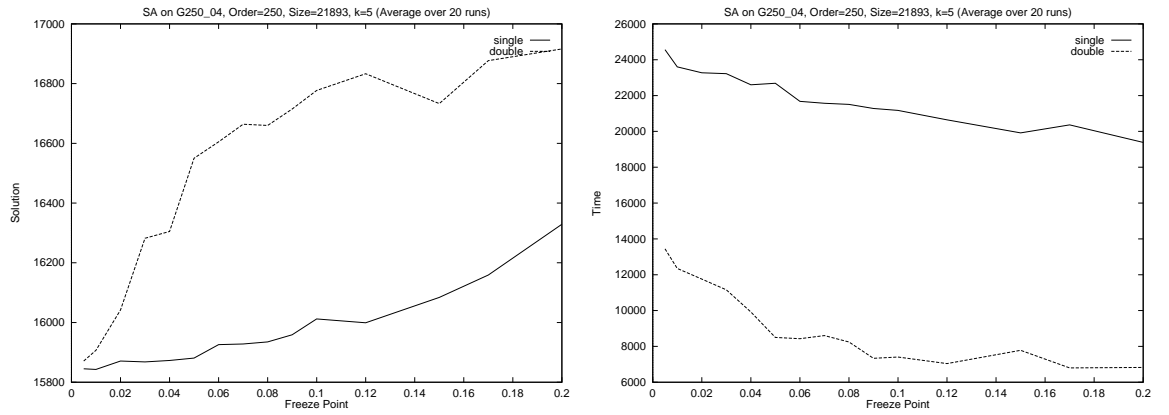


Figure 5.10: Comparison of single and double vertex migrations for different values of *Freeze Point*

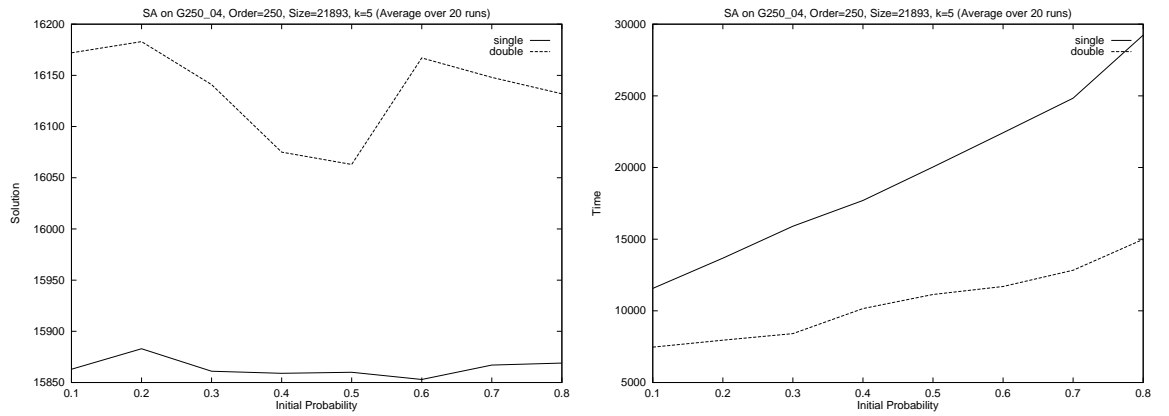


Figure 5.11: Comparison of single and double vertex migrations for different values of *Initial Probability*

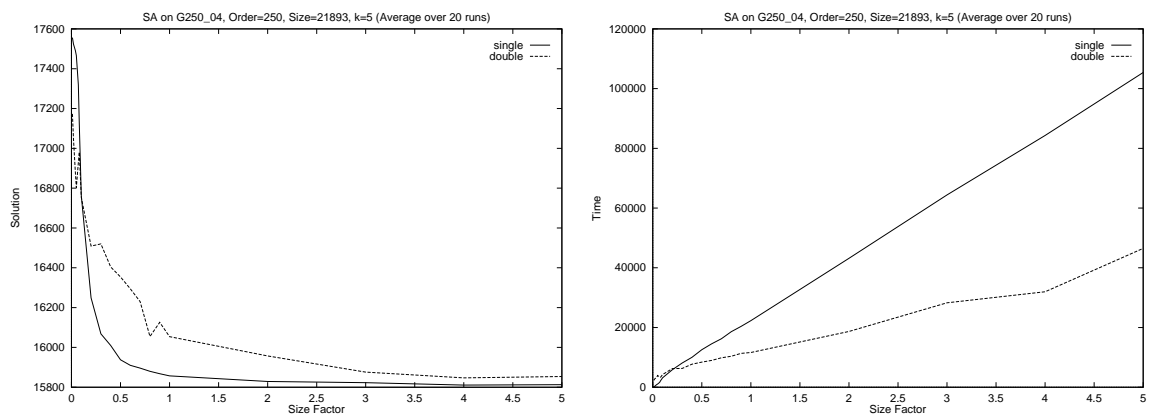


Figure 5.12: Comparison of single and double vertex migrations for different values of *Run Length*

Chapter 6

Conclusion & Future Work

In light of the experiments carried out, simulated annealing seems to be the most successful approach to graph partitioning out of the 4 methods explored. In particular, I made the following observations:

- Observation1: Lengthening the running time of simulated annealing generally yields higher quality solutions. In particular, reducing the freeze point and increasing the run length were found to be favourable ways in increasing the run time in order to increase solution quality. Increasing the cooling rate can yield better results, although this is not always the case.
- Observation2: The *initprob* parameter was found not to have a direct impact on the resulting solution.
- Observation3: GA yields better solutions when the population size is increased. This also increases the running time of the algorithm. The *MaxIterations* parameter was found to produce higher quality solutions with a greater frequency although increasing this parameter does not guarantee a gain in solution quality.
- Observation4: The *MaxKerLinIterations* parameter was found to have no direct impact on either the running time of the algorithm, or on the quality of the final solution.
- Observation5: Single vertex migrations were found to produce higher quality solutions than double vertex migrations. The single vertex migration algorithm generally converged more quickly than the double vertex migration algorithm.

One potential means of speeding up execution of these algorithms is to parallelise them. Java code can be parallelised in a number of ways. One means of achieving this is with an external package (for example PVM). Alternatively, Java contains language constructs for distributed processing via the RMI (remote method invocation) API.

One means of achieving balanced partitioning using simulated annealing with single vertex migrations, is to increase the penalty parameter as the algorithm progresses (see section 3.2).

The crossover phase of GA incorporates a random adjustment phase. This is used to ensure that the solution generated from the two parent solutions is valid (balanced).

It is possible that this random adjustment provides sufficient mutation to render the explicit mutation phase redundant.

GA may further modified as follows: Currently, GA represents solutions as a string of integers with the i^{th} entry representing the partition occupied by the i^{th} vertex. Using this representation, the crossover operation is unlikely to produce a balanced solution. Thus a postprocessing phase must be applied to the solution. An alternative representation may make use of a technique known as *random keying*. Using this representation, a solution consists of a vector of random numbers. For a 2-way partitioning, let t be such that the number of entries in the vector of value less than or equal to t is $n/2$. All those positions in the vector containing numbers less than t are grouped into one partition, while all those with entries greater than the median are grouped into the other partition. This representation means that the crossover operation will always produce a valid solution. This representation may be generalised to k -way partitioning with minor modification. By eliminating the postprocessing phase, the running time of GA may be reduced.

Bibliography

- [1] D.S Johnson, C.R Aragon, L.A McGeoch, and C. Schevon. Optimisation by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 39(3):378-406, May-June 1991.
- [2] Thang Nguyen Bui and Byung Ro Moon. Genetic Algorithm and Graph Partitioning. *IEEE Transactions On Computers*, 35(7):841-855, July 1996.
- [3] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Syst. Tech Journal*, 29:291-307, February 1970.

Appendix A

Source Code

A.1 BadVertexSwapException

```
/*
 * BadVertexSwapException.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

public class BadVertexSwapException extends Exception {

    public BadVertexSwapException() {
        super("Attempt to swap two vertices in the same partition:");
    }

    public BadVertexSwapException(String s) {
        super("Attempt to swap two vertices in the same partition:" + " : " + s);
    }
}
```

A.2 BadPartitionException

```
/*
 * BadPartitionException.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

public class BadPartitionException extends Exception {

    public BadPartitionException() {
        super("Attempt to swap two vertices in the same partition:");
    }

    public BadPartitionException(String s) {
        super("Attempt to swap two vertices in the same partition:"
            + " : " + s);
    }
}
```

A.3 CostStructure

```
/*
 * CostStructure.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.io.*;

/*
 * The cost of a vertex is a tuple (i,x) where i is the
 * internal cost and x is the external cost.
 *
 * External cost = number of edges from that vertex to vertices
 * in different partitions.
 * Internal cost = number of edges from that vertex to vertices
 * in the same partition.
 *
 * The data members internal and external really should be private
 * and accessed via
 * accessor and mutator methods below but for efficiency reasons,
 * they are just
 * accessed directly.
 *
 */

public class CostStructure implements Cloneable, Serializable {
    public int internal;
    public int external;

    public CostStructure() {
        internal = 0;
        external = 0;
    }

    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            System.err.println("Error: Failed clone of Class costStructure
                                or derived class");
            System.err.println(e.getMessage());
        }
        return(o);
    }
}
```

```
public String toString() {
    return(internal + ":" + external);
}

public int getInternalCost() {
    return(internal);
}

public int getExternalCost() {
    return(external);
}

public void setInternalCost(int newCost) {
    internal = newCost;
}

public void setExternalCost(int newCost) {
    external = newCost;
}
}
```



```

if (g.isConnected()) {
    System.out.println("# Graph is connected.");
    g.BFSReordering(0);
}

if ((order % k) == 0) { // Only divide graph evenly
    adjacencyMatrix = g.getAdjacencyMatrix();
    partitionFactor = k;
    partitionSize = (int)order/k;
    offSpring[0] = (Solution)null;
    offSpring[1] = (Solution)null;

    numberGen = new Random();
    int splitSize = 0;
    int index = 0;
    int count = 0;

    System.out.println("# Creating new population...");
    for (int i=0; i<populationSize; i++) {
        gaPopulation.replace(new Solution(g, partitionFactor));
        System.out.print('#');
    }
    System.out.println("");
    System.out.println("# Best solution: " + gaPopulation.getLowestCost());

    // Main GA loop
    startTime = System.currentTimeMillis();
    try {
        while (count < maxNonImprovingIterations) {
            // count is incremented every time an offspring is generated
            // but not replaced into the population

            offSpring = gaPopulation.getParents();
            offSpring =
                offSpring[0].randomSinglePointCrossOver(offSpring[1]);

            offSpring[0] = offSpring[0].mutate(mutatePoint, 5);
            offSpring[1] = offSpring[1].mutate(mutatePoint, 5);
            offSpring[0].postProcess();
            offSpring[1].postProcess();

            KerLin kl1 = new KerLin(g, k);
            kl1.setSolution(offSpring[0].getPartition());
            offSpring[0].setPartition(kl1.kerLinQuickOptimise(order));

            kl1.setSolution(offSpring[1].getPartition());
            offSpring[1].setPartition(kl1.kerLinQuickOptimise(order));

            replacedSolution = gaPopulation.replace(offSpring[1]);
            replacedSolution |= gaPopulation.replace(offSpring[0]);
        }
    }
}

```

```

        if (replacedSolution) {
            count = 0;
        } else {
            count++;
        }
        iterationCount++;

        System.out.print("# Iteration: " + iterationCount +
            " Best solution: " +
                gaPopulation.getLowestCost());
        System.out.println(" Worst: " + gaPopulation.getHighestCost());
    }
} catch (Exception e) {
    System.err.println("Error: " + e.getMessage());
    System.exit(1);
}
finishTime = System.currentTimeMillis();
System.out.print("# Total Iterations: " + iterationCount +
    " Running Time: " + (finishTime - startTime));
System.out.println(" Best Solution: " + gaPopulation.getLowestCost());

} else {
    System.err.println("Error: Can only divide graph evenly!");
}
}

public static void main(String[] args) {

    if (args.length == 5) {
        String graphName = new String(args[0]);
        Integer split = new Integer(args[1]);
        Integer populationSize = new Integer(args[2]);
        Integer iterations = new Integer(args[3]);
        Integer localOptCutoff = new Integer(args[4]);

        System.out.print("# DeSerialising...");
        GeometricGraph g = GeometricGraph.readFromFile(graphName);
        System.out.println(" Finished!");
        GA ga = new GA(g,
            split.intValue(),
            populationSize.intValue(),
            iterations.intValue(),
            localOptCutoff.intValue());
    } else {
        System.err.println("Usage Error:
            GA GraphFileName SplitSize
            PopulationSize MaxNonImprovingIterations
            KerLinxchange size");
    }
}
}
}

```

A.5 GeometricGraph

```
/*
 * GeometricGraph.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.lang.Math;
import java.util.Random;
import java.io.*;

public class GeometricGraph implements Serializable{

    private Random numberGen = new Random();
    private int[][] adjacencyMatrix;
    private int order = 0;
    private int size = 0;
    private int i,n;

    public GeometricGraph(int order) {
        this.order = order;
        adjacencyMatrix = new int[order][order];
    }

    public GeometricGraph(int order, String filename) {
        this.order = order;
        adjacencyMatrix = new int[order][order];
        size = readMatrixFromFile(filename, adjacencyMatrix);
    }

    /**
     * This generates edges by creating n GraphNodes and connecting
     * those nodes of Euclidean distance less than cutOffPoint.
     */
    public void generateEdges(double cutOffPoint) throws Exception {

        double delta;
        GraphNode[] nodeArray;

        nodeArray = new GraphNode[order];

        for (n = 0; n<order; n++) {
            nodeArray[n] = new GraphNode();
        }
    }
}
```

```

if ((cutOffPoint >= 0) && (cutOffPoint <= 1)) {
    // The co-ordinates of the GraphNode objects exists within a 1x1 grid.
    // Thus a valid cutoffrange is between 0 and 1.
    for (n=0; n<order-1; n++) {
        for (i= n+1; i<order; i++) {
            delta = java.lang.Math.sqrt(
                java.lang.Math.pow((nodeArray[n].x - nodeArray[i].x),2.0) +
                java.lang.Math.pow((nodeArray[n].y - nodeArray[i].y),2.0));

            if (delta < cutOffPoint) {
                adjacencyMatrix[n][i] = (numberGen.nextInt() % 100) + 1;
                adjacencyMatrix[i][n] = adjacencyMatrix[n][i];
                size++;
            }
        }
    }
} else {
    throw(new Exception("Cutoff Point out of range: " + cutOffPoint));
}

/**
 * Adds an edge of weight 1 to the graph
 */
public void addEdge(int i, int j) throws Exception {
    if (i != j) {
        adjacencyMatrix[i][j] = 1;
        adjacencyMatrix[j][i] = 1;
        size++;
    } else {
        throw(new Exception("Cannot add edge from node " + i + " to " + j));
    }
}

/**
 * Adds an edge of weight w to the graph
 */
public void addEdge(int i, int j, int w) throws Exception {
    if (i != j) {
        adjacencyMatrix[i][j] = w;
        adjacencyMatrix[j][i] = w;
        size++;
    } else {
        throw(new Exception("Cannot add edge from node " + i + " to " + j));
    }
}

public int getEdgeWeight(int i, int j) throws NoSuchEdgeException {
    if (adjacencyMatrix[i][j] != 0) {
        return adjacencyMatrix[i][j];
    } else {
        throw( new NoSuchEdgeException(i,j));
    }
}

```

```

/**
 * @return a boolean indicating if an edge joins the vertices i & j
 */
public boolean hasEdge(int i, int j) {
    return ((adjacencyMatrix[i][j] != 0) && (i != j));
}

/**
 * This returns a boolean indicating that the graph is connected.
 * This is determined by traversing the graph in breadth first
 * order and checking that all vertices have been 'visited'
 *
 * @return boolean indicating if the graph is connected.
 */
public boolean isConnected() {
    int[] traversalOrder = breadthFirstTraverse(0);
    int index = traversalOrder.length;
    int visitOrder = traversalOrder[0];

    while((index-- > 0) && (visitOrder != 0))
        visitOrder = traversalOrder[index];

    // If index == -1 then the whole array has been traversed
    return(index == -1);
}

/**
 * This returns an array of int's indicating the vertices
 * which are adjacent to the input vertex.
 *
 * @return Array of int's indicating adjacent vertices.
 */
private int[] getAdjacentVertices(int vertex) {
    int index0 = order;
    int index1 = 0;
    int count = 0;
    int[] adjacentVertices;

    // Count the number of vertices adjacent to the target vertex.
    while (--index0 >= 0)
        if (adjacencyMatrix[vertex][index0] != 0)
            count ++;

    adjacentVertices = new int[count];
    index0 = order;
    while (--index0 >= 0) {
        if (adjacencyMatrix[vertex][index0] != 0)
            adjacentVertices[index1++] = index0;
    }
    return(adjacentVertices);
}

```

```
/**
 * This returns an array of int's indicating the order in which each
 * vertex was encountered. A value of 0 in a position means that that
 * vertex was not encountered in the Breadth First Search (BFS).
 *
 * @return Array of int's indicating traversal order of vertices.
 */
private int[] breadthFirstTraverse(int vertex) {
    Queue q = new Queue();
    int[] visited = new int[order];
    int[] adjacentVertices;
    Integer v = new Integer(vertex);
    int index = 0;
    int count = 1;

    // Initialise array to 0s
    index = order;
    while (--index >= 0)
        visited[index] = 0;

    visited[v.intValue()] = 1;
    q.enqueue(v);
    try {
        while(!q.isEmpty()) {
            v = (Integer)q.serve();
            adjacentVertices = getAdjacentVertices(v.intValue());

            index = adjacentVertices.length;
            while (index-- > 0) {
                if (visited[adjacentVertices[index]] == 0) {
                    q.enqueue(adjacentVertices[index]);
                    visited[adjacentVertices[index]] = ++count;
                }
            }
        }
    } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
        System.exit(1);
    }
    return(visited);
}
```

```
/**
 * Performs Breadth First Search ReOrdering on the graph, that is,
 * renames
 * the vertices so that their name corresponds to the order in which
 * they
 * were encountered by a breadth first search. This requires the graph
 * to be
 * connected.
 */
public void BFSReordering(int vertex) {

    int[] traversal = breadthFirstTraverse(vertex);
    int[][] newAdjacencyMatrix = new int[order][order];
    int i,j;

    for (i=0; i<order; i++) {
        for (j=0; j<order; j++) {
            newAdjacencyMatrix[traversal[i] - 1][traversal[j] - 1]
                = adjacencyMatrix[i][j];
        }
    }
    adjacencyMatrix = newAdjacencyMatrix;
}

/**
 * @return The size of the graph (The number of edges)
 */
public int getSize() {
    return(size);
}

/**
 * @return The order of the graph (The number of nodes)
 */
public int getOrder() {
    return(order);
}

/**
 * @return A reference to the adjacency matrix representing the graph.
 */
public int[][] getAdjacencyMatrix() {
    return(adjacencyMatrix);
}
```

```

/**
 * Prints every edge in the for i,j to System.out
 * Used for testing purposes
 */
public void printEdges() {
    int i;
    int j;

    for (i=0; i<order; i++) {
        for (j=i; j<order; j++) {
            if (adjacancyMatrix[i][j] != 0) {
                System.out.println(i + ", " + j);
            }
        }
    }
}

/**
 * Prints the adjacancy matrix to the screen (formatted into
 * columns). Purely for test purposes.
 */
public void write() {
    int currentEntry = 0;

    for (int i=0; i<order; i++) {
        for (int j=0; j<order; j++) {
            currentEntry = adjacancyMatrix[i][j];
            if (currentEntry < 10 && currentEntry >= 0) {
                System.out.print(" " + currentEntry);
            } else if ((currentEntry >= 10) ||
                (currentEntry < 0 && currentEntry > -10)) {
                System.out.print(" " + currentEntry);
            } else if (currentEntry <= -10) {
                System.out.print(" " + currentEntry);
            }
        }
        System.out.println("");
    }
}

/**
 * Saves the state of the object to a file.
 */
public void writeToFile(String filename) {

    try {
        ObjectOutputStream os = new
            ObjectOutputStream(new FileOutputStream(filename));
        os.writeObject(this);
        os.close();
    } catch (Exception e) {
        System.err.println("Error writing object: " + e.getMessage());
    }
}

```

```

public static GeometricGraph readFromFile(String filename) {
    GeometricGraph g = (GeometricGraph)null;
    try {
        ObjectInputStream in = new
            ObjectInputStream(new FileInputStream(filename));
        g = (GeometricGraph)in.readObject();
    } catch (Exception e) {
        System.err.println("Error reading object: " + e.getMessage());
    }
    return(g);
}

/**
 * This reads a text file describing a matrix and assigns that
 * matrix to that passed in.
 *
 * @return The size of the graph represented by the matrix read in
 */
public int readMatrixFromFile(String filename, int[][] matrix) {
    // The reference to a matrix is passed in rather than creating
    // a matrix and returning that. This saves on a potentially wasteful
    // memory allocation to accommodate a potentially very large matrix
    // which is typically destroyed afterwards.

    int graphSize = 0;
    try {
        FileReader fileReader = new FileReader(filename);
        StreamTokenizer st = new StreamTokenizer(fileReader);
        int rowLength = matrix.length;
        int columnLength = matrix[0].length;
        int row = 0;
        int col = 0;

        while (st.nextToken() != StreamTokenizer.TT_EOF) {
            if ((matrix[row][col] = (int)st.nval) != 0) {
                graphSize++;
            }

            col++;
            if (col == columnLength) {
                col = 0;
                row++;
            }
        }
    } catch (Exception e) {
        System.err.println("Exception: " + e);
    }
    return((int)graphSize/2);
}

```

```

public static GeometricGraph readAdjacencyList(String filename) {

    int graphOrder = 0;
    int graphSize = 0;
    int vertex = 0;
    GeometricGraph g = (GeometricGraph)null;

    try {
        FileReader fileReader = new FileReader(filename);
        StreamTokenizer st = new StreamTokenizer(fileReader);

        graphOrder = (int)st.nval;
        st.nextToken();
        graphSize = (int)st.nval;
        g = new GeometricGraph(graphOrder);

        while (st.nextToken() != StreamTokenizer.TT_EOF) {
            vertex = (int)st.nval;
            while (st.nextToken() != StreamTokenizer.TT_EOL) {
                g.addEdge(vertex, (int)st.nval);
            }
        }
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
    return(g);
}

public void printAdjacencyList() {
    int i;
    int j;
    String row = new String("");

    System.out.println(getOrder() + " " + getSize());
    for (i=0; i<order; i++) {
        row = "";
        for (j=i; j<order; j++) {
            if (adjacencyMatrix[i][j] != 0) {
                row = row + (" " + j);
            }
        }
        if (row.length() != 0) {
            System.out.println(i + " " + row);
        }
    }
}

```

```
public static void main(String[] args) {
    /*
     * At the moment the main function provides a user interface to creating
     * a new instance of class GeometricGraph and writing it to a file.
     * This should be moved to a separate method.
     */

    if (args.length == 3) {
        try {
            System.out.println("Creating new graph of order " + args[0]);
            GeometricGraph g = new
                GeometricGraph(new Integer(args[0]).intValue());
            g.generateEdges(new Double(args[1]).doubleValue());
            System.out.println("Graph has size: " + g.getSize());
            System.out.println("Writing graph to file " + args[2]);
            g.writeToFile(args[2]);
            if (g.isConnected()) {
                System.out.println("Graph is connected.");
            } else {
                System.out.println("Graph is unconnected");
            }
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    } else {
        System.err.println("Usage Error: GeometricGraph Order
                            CutOffPoint FileName");
    }
}

class GraphNode implements Serializable {
    public double x;
    public double y;
    private Random numberGen = new Random();

    GraphNode () {
        x = numberGen.nextDouble();
        y = numberGen.nextDouble();
    }
}
```

A.6 KerLin

```

/*
 * KerLin.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.util.Random;

public class KerLin {
    private GeometricGraph graph;
    private int[] [] graphAdjacencyMatrix;
    private int[] partitionVector;
    private int[] differenceVector;
    private int[] [] exchangeVector;
    private int[] gainList;
    private boolean[] exchanged;
    private CostStructure[] vertexCostVector;
    private int[] degree;
    private int graphOrder;
    private int max_i;
    private int max_j;
    private int k;

    public KerLin(GeometricGraph g,
                 int k,
                 int[] initialSolution) {

        graph = g;
        partitionVector = initialSolution;
        graphOrder = g.getOrder();
        graphAdjacencyMatrix = g.getAdjacencyMatrix();
        differenceVector = new int[graphOrder];
        exchangeVector = new int[graphOrder][2];
        gainList = new int[graphOrder];
        exchanged = new boolean[graphOrder];
        max_i = 0;
        max_j = 0;
        this.k = k;
        int i;
        int j;

        // Initialise degree vector
        degree = new int[graphOrder];

        for(i=0; i<graphOrder; i++) {
            degree[i] = 0;
        }
    }

```

```

for(i=0; i<graphOrder; i++) {
    for(j=i+1; j<graphOrder; j++) {
        if (graphAdjacencyMatrix[i][j] != 0) {
            degree[i]++;
            degree[j]++;
        }
    }
}

// Initialise vertexCostVector
vertexCostVector = new CostStructure[graphOrder];
for (i=0; i<graphOrder; i++) {
    vertexCostVector[i] = new CostStructure();
}

}

public KerLin(GeometricGraph g, int k) {
    graph = g;
    graphOrder = g.getOrder();
    graphAdjacencyMatrix = g.getAdjacencyMatrix();
    differenceVector = new int[graphOrder];
    exchangeVector = new int[graphOrder] [2];
    gainList = new int[graphOrder];
    exchanged = new boolean[graphOrder];
    partitionVector = new int[graphOrder];
    max_i = 0;
    max_j = 0;
    this.k = k;
    int i;
    int j;

    // Initialise degree vector
    degree = new int[graphOrder];

    for(i=0; i<graphOrder; i++) {
        degree[i] = 0;
    }

    for(i=0; i<graphOrder; i++) {
        for(j=i+1; j<graphOrder; j++) {
            if (graphAdjacencyMatrix[i][j] != 0) {
                degree[i]++;
                degree[j]++;
            }
        }
    }

    // Initialise vertexCostVector
    vertexCostVector = new CostStructure[graphOrder];
    for (i=0; i<graphOrder; i++) {
        vertexCostVector[i] = new CostStructure();
    }
}

```

```

public void setSolution(int[] newSolution) {
    System.arraycopy(newSolution, 0, partitionVector, 0,
                     partitionVector.length);
}

public int[] kerLinQuickOptimise(int maxExchangeSize) {
    // Performs only one iteration of Kernighan Lin

    int i;
    int j;
    int kerLinGain = 0;
    int[] optimisedSolution = new int[graphOrder];

    // Perform pairwise optimisation on partitions
    try {
        for(i=0; i<k; i++) {
            for(j=i+1; j<k; j++) {
                kerLinGain = phaseOneOptimisation(maxExchangeSize, i, j);
            }
        }
        optimisedSolution = partitionVector;
    } catch (Exception e) {
        System.err.println("Exception: " + e.getMessage());
        System.exit(-1);
    }
    return(optimisedSolution);
}

public Solution kerLinQuickOptimise(Solution s, int maxExchangeSize) {
    // Performs only one iteration of Kernighan Lin

    int i;
    int j;
    int kerLinGain = 0;
    Solution newSolution = (Solution)s.clone();

    System.arraycopy( s.getPartition(), 0, partitionVector, 0,
                     partitionVector.length);
    // Perform pairwise optimisation on partitions
    try {
        for(i=0; i<k; i++) {
            for(j=i+1; j<k; j++) {
                kerLinGain = phaseOneOptimisation(maxExchangeSize, i, j);
            }
        }
        newSolution.setPartition(partitionVector);
    } catch (Exception e) {
        System.err.println("Exception: " + e.getMessage());
        System.exit(-1);
    }
    return(newSolution);
}

```

```

public int[] kerLinOptimise(int maxExchangeSize) {

    int i;
    int j;
    int kerLinGain = 0;
    int[] optimisedSolution = new int[graphOrder];

    try {
        do {
            kerLinGain = 0;

            // Perform pairwise optimisation on partitions
            for(i=0; i<k; i++) {
                for(j=i+1; j<k; j++) {
                    kerLinGain = phaseOneOptimisation(maxExchangeSize, i, j);
                }
            }
        } while (kerLinGain != 0);
        optimisedSolution = partitionVector;
    } catch (Exception e) {
        System.err.println("Caught Exception: " + e.getMessage());
        System.exit(-1);
    }
    return(optimisedSolution);
}

private int phaseOneOptimisation(int maxExchangeSize,
                                int pPartition,
                                int qPartition)
    throws BadPartitionException {

    int index = 0;
    int maxIndex = partitionVector.length;
    boolean foundp = false;
    boolean foundq = false;
    int maxGain;
    int randomVertexP = 0;
    int randomVertexQ = 0;

    // Check that both partitions exist
    index = 0;
    while ((index < maxIndex) && !(foundp && foundq)) {
        foundp |= (partitionVector[index] == pPartition);
        foundq |= (partitionVector[index] == qPartition);
        index++;
    }

    if ((foundp && foundq) && (pPartition != qPartition)) {
        // Initialise the values of internal and external for each
        // CostStructure in the
        // vector. This depends on the partitions that are under c
        // onsideration.
        calculateCostVector(pPartition, qPartition);
    }
}

```

```

// Find an arbitrary vertex p in pPartition and q in qPartition
index=0;
while ((index < graphOrder) && ((randomVertexP == 0) ||
    (randomVertexQ == 0))) {
    if (partitionVector[index] == pPartition) {
        randomVertexP = index;
    }
    if (partitionVector[index] == qPartition) {
        randomVertexQ = index;
    }
    index++;
}
max_i = randomVertexP;
max_j = randomVertexQ;

// Initialise differenceVector
for(index=0; index<graphOrder; index++) {
    differenceVector[index] = vertexCostVector[index].external -
        vertexCostVector[index].internal;
}

// Initialise exchange vector and gainList
for (index=0; index<graphOrder; index++) {
    exchanged[index] = false;
    gainList[index] = Integer.MIN_VALUE;
}

index=0;
while (index<maxExchangeSize) {
    // Get two vertices yielding maximum gain
    try {
        max_i = getMaxGainVertex(pPartition);
        max_j = getMaxGainVertex(qPartition);
    } catch (Exception e) {
        // No exchanges left, break out of loop
        break;
    }
    maxGain = gain(max_i, max_j);
    exchangeVector[index][0] = max_i;
    exchangeVector[index][1] = max_j;

    exchanged[max_i] = true;
    exchanged[max_j] = true;
    gainList[index] = maxGain;

    // Update the gain list
    updateDifferenceVector();
    index++;
}

// Find optimum k such that k exchanges yields greatest improvement
int maximumGain = 0;
int currentGain = 0;

```

```

    int k = -1;

    index = 0;
    while ((index < maxExchangeSize) &&
           (gainList[index] != Integer.MIN_VALUE)) {

        currentGain += gainList[index];
        if (currentGain > maximumGain) {
            maximumGain = currentGain;
            k = index;
        }
        index++;
    }

    if (maximumGain > 0) {
        // Make changes as specified by exchangeVector
        for (index=0; index<=k; index++) {
            exchange(exchangeVector[index][0], exchangeVector[index][1]);
        }
    }
    return(maximumGain);
} else {
    throw(new BadPartitionException());
}
}

private int getMaxGainVertex(int partition)
    throws NoVerticesAvailableException {

    int index=0;
    int maxVertex = 0;
    boolean foundVertex = false;

    // Find any vertex in partition that is available for swapping
    while ((index<graphOrder) && (!foundVertex)) {
        if ((partitionVector[index] == partition) && (!exchanged[index])) {
            maxVertex = index;
            foundVertex = true;
        }
        index++;
    }

    if (foundVertex) {
        // Find the best vertex to swap
        while (index<graphOrder) {
            // If the vertex is in the partition and is available
            if ((partitionVector[index] == partition) && (!exchanged[index])) {
                if (differenceVector[index] > differenceVector[maxVertex]) {
                    maxVertex = index;
                }
            }
            index++;
        }
    } else {

```

```

        throw(new NoVerticesAvailableException());
    }
    return(maxVertex);
}

private void updateDifferenceVector() {
    // Difference vector contains differences between
    // internal & external(wrt)cost
    // of vertices available for swapping
    // A value of Integer.MIN_VALUE indicated that the vertex is
    // not available for swapping

    int index;
    int delta;

    // Need to generate arbitrary newMax_i & newMax_j

    for(index=0; index<graphOrder; index++) {
        if (exchanged[index]) {
            differenceVector[index] = Integer.MIN_VALUE;
        } else {
            if (partitionVector[index] == partitionVector[max_i]) {
                delta = 2 * (graphAdjacencyMatrix[index][max_i] -
                    graphAdjacencyMatrix[index][max_j]);
                differenceVector[index] += delta;

            } else if (partitionVector[index] == partitionVector[max_j]) {
                delta = 2 * (graphAdjacencyMatrix[index][max_j] -
                    graphAdjacencyMatrix[index][max_i]);
                differenceVector[index] += delta;

            } else if ((partitionVector[index] != partitionVector[max_i]) &&
                (partitionVector[index] != partitionVector[max_j])) {
                // The vertex is not available for switching since it is
                // in a different partition
                differenceVector[index] = Integer.MIN_VALUE;
            }
        }
    }
}
}

```

```

private void calculateCostVector(int pPartition, int qPartition) {
    // For each vertex calculate its cost. Each entry contains a
    // CostStructure.
    // The cost vector does not convey the absolute cost of each
    // vertex, it conveys its
    // internal cost and its external cost with respect to a partition
    // under consideration.

    int n;
    for(n=0; n<graphOrder; n++) {
        if (partitionVector[n] == pPartition) {
            vertexCostVector[n] = getVertexCostWrt(n, qPartition);
        }
        if (partitionVector[n] == qPartition) {
            vertexCostVector[n] = getVertexCostWrt(n, pPartition);
        }
    }
}

private int getInternalCost(int vertex) {
    // Returns the internal cost of a vertex

    int internalCost = 0;
    for(int n=0; n<graphOrder; n++) {
        if ((graph.hasEdge(vertex, n)) &&
            (partitionVector[n] == partitionVector[vertex])) {
            internalCost++;
        }
    }
    return(internalCost);
}

private int getExternalCost(int vertex) {
    // Returns the external cost of a vertex

    int externalCost = 0;
    for(int n=0; n<graphOrder; n++) {
        if ((graph.hasEdge(vertex, n)) &&
            (partitionVector[n] != partitionVector[vertex])) {
            externalCost++;
        }
    }
    return(externalCost);
}

```

```

private CostStructure getVertexCostWrt(int vertex, int targetPartition) {
    /* Returns a CostStructure with the vertex external
    * cost with respect to targetPartition and its internal cost.
    * The external cost is with respect to a particular partition
    * rather than with respect to all other partitions
    * because at any time, migrations between only two partitions
    * is considered (k-way partitioning is performed
    * in a pairwise manner.
    */
    int vertexPartition = partitionVector[vertex];
    CostStructure c = new CostStructure();
    for(int n=0; n<graphOrder; n++) {
        if (graphAdjacencyMatrix[vertex][n] != 0) {
            if ((partitionVector[n] == vertexPartition) && (n != vertex)) {
                c.internal++;
            }
            if (partitionVector[n] == targetPartition){
                c.external++;
            }
        }
    }
    return(c);
}

private int getCostBetweenPartitions(int pPartition, int qPartition)
    throws BadPartitionException {

    int i=0;
    int j=0;
    int interPartitionCost = 0;

    if ((pPartition == qPartition) || (pPartition >= k) ||
        (pPartition < 0) || (qPartition >= k) || (qPartition < 0)) {
        throw (new BadPartitionException());
    } else {
        for (i=0; i<graphOrder; i++) {
            for (j=i+1; j<graphOrder; j++) {
                if (graphAdjacencyMatrix[i][j] != 0) {
                    // If vertices i and j are connected
                    if (((partitionVector[i] == pPartition) &&
                        (partitionVector[j] == qPartition)) ||
                        ((partitionVector[i] == qPartition) &&
                        (partitionVector[j] == pPartition))) {
                        /* If one vertex is in pPartition and
                        * the other is in qPartition
                        */
                        interPartitionCost++;
                    }
                }
            }
        }
    }
    return(interPartitionCost);
}

```

```

private int gain(int i, int j) {
    // Returns the gain resulting from switching vertices i and j
    int gainVal;

    if ((i == j) || (partitionVector[i] == partitionVector[j]) ||
        exchanged[i] || exchanged[j]) {
        gainVal = Integer.MIN_VALUE;
    } else {
        // gain(i,j) = Di + Dj - 2Aij
        gainVal = differenceVector[i] + differenceVector[j] -
            (2 * graphAdjacencyMatrix[i][j]);
    }
    return(gainVal);
}

private void exchange(int i, int j) {
    // Updates the state to reflect switch of vertices i and j.

    // Switch the vertices.
    int tmpPartition = partitionVector[i];
    partitionVector[i] = partitionVector[j];
    partitionVector[j] = tmpPartition;

    // Update cost vector
    calculateCostVector(partitionVector[i], partitionVector[j]);
}

public int getCost() {
    /* Returns the total external cost of the partitioning.
    * Each vertex has an internal cost and also an external
    * cost with respect to the current partition
    * under consideration.
    * It is not the case in general that degree[n] =
    * vertexCostVector[n].internal + vertexCostVector[n].external
    * except where k = 2.
    */
    int totalExternalCost = 0;
    int n;

    for(n=0; n<graphOrder; n++) {
        // totalExternalCost += (degree[n] - getInternalCost(n));
        totalExternalCost += getExternalCost(n);
    }
    return((int)totalExternalCost/2);
}

```

```

private int testGetCost() {
    // For test purposes
    int i=0;
    int j=0;
    int cost=0;
    for (i=0; i<graphOrder; i++) {
        for (j=i; j<graphOrder; j++) {
            if (graph.hasEdge(i,j)) {
                if (partitionVector[i] != partitionVector[j]) {
                    // vertices are in different partitions
                    cost++;
                }
            }
        }
    }
    return(cost);
}

protected void report() {
    // Write some status information to System.out, used only for testing.

    int n;
    int vertexExternalCost;
    int vertexInternalCost;
    CostStructure[] partitionCosts = new CostStructure[k];

    for(n=0; n<k; n++) {
        partitionCosts[n] = new CostStructure();
    }

    for(n=0; n<graphOrder; n++) {
        vertexExternalCost = getExternalCost(n);
        vertexInternalCost = (degree[n] - vertexExternalCost);
        // vertexInternalCost = getInternalCost(n);
        partitionCosts[partitionVector[n]].internal += vertexInternalCost;
        partitionCosts[partitionVector[n]].external += vertexExternalCost;
    }

    for(n=0; n<k; n++) {
        partitionCosts[n].internal /= 2;
    }

    // Write status to stdout
    for(n=0; n<k; n++) {
        System.out.print(n + ": " + partitionCosts[n].internal +
            ", " + partitionCosts[n].external + " ");
    }
    System.out.println("Cost: " + getCost() + ", " + testGetCost());
}

```

```

private void test() {
    System.out.println("Graph has size: " + graph.getSize());
    int totalDegree = 0;
    for (int i=0; i<graphOrder; i++) {
        totalDegree += degree[i];
    }
    totalDegree = (int) totalDegree/2;
    System.out.println("Calculate edges: " + totalDegree);
    System.out.println("Cost: " + getCost());
    int xcost=0;
    for (int i=0; i<graphOrder; i++) {
        for (int j=i; j<graphOrder; j++) {
            if (graph.hasEdge(i, j) && (partitionVector[i] !=
                partitionVector[j])) {
                xcost++;
            }
        }
    }
    System.out.println("Calculate Cost: " + xcost);

    for (int i=0; i<graphOrder; i++) {
        System.out.println(getInternalCost(i) + "    " +
            getExternalCost(i) + "    " + degree[i]);
    }
}

public static void main(String[] args) {

    int k=0;
    int graphOrder;
    int numberOfArgs = args.length;
    long startTime = 0;
    long finishTime = 0;
    String graphName = new String("");
    String optSwitch = new String("");
    String solutionFileName = new String("");
    GeometricGraph g = (GeometricGraph)null;
    Solution initialSolution = (Solution)null;

    String tmpString = new String("");
    int i=0;
    if ((numberOfArgs != 5) && (numberOfArgs != 7)) {
        System.err.println("Error: KerLin -g GraphFile -k
            Partition Size [-sol Solution] -f|-q");
        System.exit(1);
    } else {
        while (i < numberOfArgs) {
            tmpString = args[i];
            if (tmpString.equals("-g")) {
                g = GeometricGraph.readFromFile(args[++i]);
            } else if (tmpString.equals("-k")) {
                k = new Integer(args[++i]).intValue();
            } else if (tmpString.equals("-q") || tmpString.equals("-f")) {
                optSwitch = new String(args[i]);
            }
        }
    }
}

```

```

        } else if (tmpString.equals("-sol")) {
            solutionFileName = new String(args[++i]);
        } else {
            System.err.println("Unrecognised Argument: " + args[i]);
            System.exit(1);
        }
        i++;
    }
}

System.out.println("# Using graph of order: " + g.getOrder() +
    " and size: " + g.getSize() + ", k = " + k);
if (solutionFileName.equals("")) {
    // Generate random solution
    initialSolution = new Solution(g,k);
} else {
    // Read solution from file
    initialSolution = Solution.readFromFile(solutionFileName);
    initialSolution.setGraph(g);
}

Solution s2 = (Solution)initialSolution.clone();
KerLin kl = new KerLin(g, k, initialSolution.getPartition());
graphOrder = g.getOrder();

// Start Kernighan-Lin algorithm
System.out.println("# Initial Cost: " + kl.getCost());
startTime = System.currentTimeMillis();

if (optSwitch.equals("-f")) {
    kl.kerLinOptimise(graphOrder);
} else if (optSwitch.equals("-q")) {
    kl.kerLinQuickOptimise(graphOrder);
} else {
    System.err.println("Error: Illegal option: " + optSwitch);
}

finishTime = System.currentTimeMillis();
System.out.println("# Total running time in milliseconds: " +
    (finishTime - startTime));
System.out.println("# Solution: " + kl.getCost());
}
}

```

A.7 LocalOpt

```

/*
 * LocalOpt.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.util.Random;

public class LocalOpt {

    Random numberGen;

    public LocalOpt(GeometricGraph g, int partitionSize, double iWeight) {
        System.out.print("# Creating solution...");
        Solution s = new Solution(g,partitionSize);
        System.out.println("done!");

        localOpt(g, partitionSize, iWeight, s);
    }

    public LocalOpt(GeometricGraph g,
                    int partitionSize,
                    double iWeight,
                    String solnFile) {

        System.out.print("# Reading solution from file.....");
        Solution s = Solution.readFromFile(solnFile);
        System.out.println("..done");
        s.setGraph(g);
        localOpt(g, partitionSize, iWeight, s);
    }

    private void localOpt(GeometricGraph g,
                           int k,
                           double iWeight,
                           Solution initSolution) {

        double delta = 0;
        double newCost = 0;
        long startTime = System.currentTimeMillis();
        long finishTime;
        Solution currentSolution = initSolution;
        Solution newSolution;
        currentSolution.setImbalanceWeight(iWeight);
        double currentCost = currentSolution.getCost();
        int graphOrder = g.getOrder();

```

```

int partitionCount = 0;
int vertexCount = 0;
int activeVertex = 0;
int iterationCount=0;

System.out.println("# Starting Local Optimisation!");
while (vertexCount < graphOrder) {
    while (partitionCount < k) {
        newSolution =
            currentSolution.legalMigration(activeVertex, partitionCount);
        currentCost = currentSolution.getCost();
        newCost = newSolution.getCost();
        delta = (newSolution.getCost() - currentSolution.getCost());

        if (delta < 0) {
            currentSolution = (Solution)newSolution.clone();
            System.out.println(iterationCount + " " +
                currentSolution.getCost() + " " +
                currentSolution.getCutEdges());

            partitionCount = 0;
            vertexCount = 0;
        } else {
            partitionCount++;
        }
        iterationCount++;
    }
    partitionCount = 0;
    vertexCount++;

    activeVertex++;
    activeVertex %= graphOrder;
}

finishTime = System.currentTimeMillis();
System.out.println("# Total running time in milliseconds: " +
    (finishTime - startTime));
System.out.println("# Solution: " + currentSolution.getCost() +
    " " + currentSolution.getCutEdges());
}

public static void main(String[] args) {

    int numberOfArgs = args.length;
    int i = 0;
    String tmpString = new String();

    GeometricGraph g = (GeometricGraph)null;
    int partition = 0;
    double initProb = 0;
    double rate = 0;
    int runLength = 0;
    double minRatio = 0;
    double imbalanceScaling = 1;
    String filename = "";

```


A.8 NoSuchEdgeException

```
/*
 * NoSuchEdgeException.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

public class NoSuchEdgeException extends Exception {

    public NoSuchEdgeException() {
        super("No Such Edge:");
    }

    public NoSuchEdgeException(String s) {
        super("No Such Edge:" + " : " + s);
    }

    public NoSuchEdgeException(int i, int j) {
        super("No Such Edge (" + i + ", " + j + ")");
    }

    public NoSuchEdgeException(int i, int j, String s) {
        super("No Such Edge (" + i + ", " + j + ")" + " : " + s);
    }
}
```

A.9 NoVerticesAvailableException

```
/*
 * NoVerticesAvailableException.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

public class NoVerticesAvailableException extends Exception {

    public NoVerticesAvailableException() {
        super("No Vertices Available for Swapping");
    }

    public NoVerticesAvailableException(String s) {
        super("No Vertices Available for Swapping:" + " : " + s);
    }
}
```

A.10 Population

```
/*
 * Population.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.util.Vector;
import java.util.Random;
import java.util.Enumeration;

public class Population {

    private Vector populationVector;
    private double lowestCost;
    private double highestCost;
    private double totalCost;
    private int maxPopSize;
    private Random numberGen;

    public Population(int capacity) {
        populationVector = new Vector(capacity);
        maxPopSize = capacity;
        numberGen = new Random();
        highestCost = 0;
        lowestCost = Double.MAX_VALUE;
    }

    public boolean replace(Solution s) {
        int populationSize = populationVector.size();
        int index = 0;
        double currentCost = 0;
        boolean replaceOccured = false;

        if (populationVector.isEmpty()) {
            populationVector.addElement(s);
            totalCost = s.getCost();
            lowestCost = s.getCost();
            highestCost = s.getCost();
            replaceOccured = true;
        } else {
            if (s.getCost() < highestCost || (populationSize < maxPopSize)) {
                // Check for Solution of equal cost
                int i = populationSize;
                double cost = s.getCost();
                boolean foundEqualSolution = false;
            }
        }
    }
}
```

```
while ((i-- > 0) && (!foundEqualSolution))
    foundEqualSolution =
        (cost == ((Solution)populationVector.elementAt(i)).getCost());

if (!foundEqualSolution) {
    if (populationSize == maxPopSize) {
        totalCost -= highestCost;
        populationVector.removeElementAt(--populationSize);
    }

    index = 0;
    currentCost =
        ((Solution)populationVector.firstElement()).getCost();
    while ((s.getCost() > currentCost) &&
        (index < (populationSize - 1))) {
        index++;
        currentCost =
            ((Solution)populationVector.elementAt(index)).getCost();
    }

    if (s.getCost() > currentCost) {
        populationVector.addElement(s);
    } else {
        populationVector.insertElementAt(s, index);
    }

    // Recalculate highest and lowest cost
    lowestCost =
        ((Solution)populationVector.firstElement()).getCost();
    highestCost =
        ((Solution)populationVector.lastElement()).getCost();

    totalCost += s.getCost();
    replaceOccured = true;
}
}
return(replaceOccured);
}
```

```

public Solution[] getParents() throws Exception {

    int solutionIndex;
    int randomIndex;
    int populationSize;
    double totalFitness;
    double currentFitness;
    Solution[] parents = new Solution[2];

    populationSize = populationVector.size();

    if (populationSize != 0) {
        if (populationSize == 1) {
            solutionIndex = 0;
        } else {
            totalFitness =
                (populationSize * ((4*highestCost - lowestCost)/3)) - totalCost;
            randomIndex = (Math.abs(numberGen.nextInt()) % (int)totalFitness);

            solutionIndex = 0;
            currentFitness =
                fitness((Solution)populationVector.elementAt(solutionIndex));
            while (randomIndex > currentFitness) {
                currentFitness +=
                    fitness((Solution)populationVector.elementAt(solutionIndex++));
            }
        }
        parents[0] = (Solution)populationVector.elementAt(solutionIndex);
        parents[1] = (Solution)populationVector.
            elementAt(++solutionIndex % populationSize);
        return(parents);
    } else {
        throw (new Exception("Population is empty"));
    }
}

public double fitness(Solution s) {
    return((((4 * highestCost) - lowestCost)/3) - s.getCost());
}

public int size() {
    return(populationVector.size());
}

public Solution getBestSolution() {
    return((Solution)populationVector.elementAt(0));
}

public Solution getWorstSolution() {
    return((Solution)populationVector.lastElement());
}

public double getHighestCost() {
    return(highestCost);
}

```

```
}

public double getLowestCost() {
    return(lowestCost);
}

private void printCosts() {
    // For testing purposes

    Enumeration e = populationVector.elements();
    while (e.hasMoreElements()) {
        System.out.print(((Solution)e.nextElement()).getCost() + " ");
    }
    System.out.println("");
}

public static void main(String[] args) {

    Population p = new Population(10);
    GeometricGraph g = GeometricGraph.readFromFile("data/G200.0.2");
    Solution s;
    Solution[] parents = new Solution[2];
    double totalFitness = 0;

    try {
        for (int i=0; i<20; i++) {
            s = new Solution(g, 2);
            p.replace(s);
            System.out.println("Size: " + p.size() +
                               " Best: " + p.lowestCost +
                               " Worst: " + p.highestCost);

            p.printCosts();
            parents = p.getParents();
            System.out.println(" Parent cost: " + parents[0].getCost());
            System.out.println(" Parent cost: " + parents[1].getCost());
        }
    } catch (Exception e) {
        System.out.println("Error: " + e.toString() + " " + e.getMessage());
    }
}
}
```

A.11 Queue

```
/*
 * Queue.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

public class Queue {

    QueueNode head = new QueueNode();
    QueueNode tail = new QueueNode();
    int qSize;

    public Queue() {
        head = new QueueNode();
        tail = head;
        qSize = 0;
    }

    /**
     * Checks to see if the Queue is empty
     * @return boolean indicating if Queue is empty
     */
    public boolean isEmpty() {
        return(qSize == 0);
    }

    /**
     * @return An int indicating the number of elements in the Queue
     */
    public int getSize() {
        return(qSize);
    }

    /**
     * @return String consisting of concatenation of all the
     * Queue elements in String form
     */
    public String toString() {
        int index = qSize;
        String s = new String("");
        QueueNode node = head;

        while (--index >= 0) {
            s += node.data.toString() + " ";
            node = node.next;
        }
        return(s);
    }
}
```

```
/**
 * Adds an Object to the Queue
 * @param      The Object to be enQueued
 */
public void enqueue(Object o) {
    QueueNode newQueueNode = new QueueNode();
    tail.data = o;
    tail.next = newQueueNode;
    tail = tail.next;
    qSize++;
}

/**
 * Adds an int to the Queue
 * This method is superfluous and should be removed at some stage
 * @param      The int to be enQueued
 */
public void enqueue(int i) {
    enqueue(new Integer(i));
}

/**
 * Returns the element currently at the head of the Queue or else
 * throws an Exception if the Queue is empty.
 * @return      The element currently at the head of the Queue.
 */
public Object serve() throws Exception {
    if (qSize > 0) {
        Object returnVal = head.data;
        head = head.next;
        qSize--;
        return(returnVal);
    } else {
        throw(new Exception("Queue is empty."));
    }
}
```

```
public static void main(String[] args) {
    /*
     * Some test code.
     *
     */
    Queue q = new Queue();
    q.enqueue("First");
    q.enqueue("Second");
    q.enqueue("Third");
    System.out.println(q.toString());
    try {
        while (!q.isEmpty()) {
            System.out.print("Size: " + q.getSize() + " ");
            System.out.println(q.serve());
        }
        System.out.println("Size: " + q.getSize());
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}

class QueueNode {

    Object data;
    QueueNode next;

    public QueueNode() {
        data = null;
        next = null;
    }
}
```

A.12 SA

```

/*
 * SA.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.util.Random;

public class SA {

    Random numberGen;

    public SA(GeometricGraph g,
              int partitionSize,
              double initProb,
              double coolingRate,
              float sizeFactor,
              double minRatio,
              double iWeight,
              int maxIterations,
              boolean singleVertexMigrations) {

        Solution s = new Solution(g,partitionSize);
        if (singleVertexMigrations) {
            Solution newSol = singleVertexMigrationAnnealing(g,
                                                              partitionSize,
                                                              initProb,
                                                              coolingRate,
                                                              sizeFactor,
                                                              minRatio,
                                                              iWeight,
                                                              s,
                                                              maxIterations);

            singleVertexMigrationAnnealing(g,
                                           partitionSize,
                                           0.1,
                                           coolingRate,
                                           (float)0.4,
                                           minRatio,
                                           10,
                                           newSol,
                                           maxIterations);
        } else {
            doubleVertexMigrationAnnealing(g,
                                           partitionSize,
                                           initProb,

```

```

        coolingRate,
        sizeFactor,
        minRatio,
        s,
        maxIterations);
    }
}

public SA(GeometricGraph g,
        int partitionSize,
        double initProb,
        double coolingRate,
        float sizeFactor,
        double minRatio,
        double iWeight,
        String solnFile,
        int maxIterations,
        boolean singleVertexMigrations) {

    Solution s = Solution.readFromFile(solnFile);
    s.setGraph(g);

    if (singleVertexMigrations) {
        singleVertexMigrationAnnealing(g,
                partitionSize,
                initProb,
                coolingRate,
                sizeFactor,
                minRatio,
                iWeight,
                s,
                maxIterations);
    } else {
        doubleVertexMigrationAnnealing(g,
                partitionSize,
                initProb,
                coolingRate,
                sizeFactor,
                minRatio,
                s,
                maxIterations);
    }
}

private Solution singleVertexMigrationAnnealing(GeometricGraph g,
        int k,
        double initProb,
        double coolingRate,
        float sizeFactor,
        double minRatio,
        double iWeight,
        Solution initSolution,
        int maxIterations) {

```

```

numberGen = new Random();
double temperature;
double delta = 0;
double initDelta = 0;
double newCost = 0;
long finishTime;
int index = 0;
int numberAcceptedMoves = 0;
double ratioAcceptedMoves = 1;
int minPercentCount = 0;
Solution currentSolution = initSolution;
Solution bestSolution;
Solution newSolution;
Solution testSolution;
currentSolution.setImbalanceWeight(iWeight);
double currentCost = currentSolution.getCost();
double bestCost = currentCost;
int activeVertex = 0;
int partitionCount = 0;
int migrationCount = 0;
int graphOrder = g.getOrder();
int runLength = (int)(graphOrder * sizeFactor);
int totalIterations = 0;
int targetPartition = 0;
int verticesExamined = 0;
bestSolution = (Solution)currentSolution.clone();
boolean freeVertex[] = new boolean[graphOrder];
boolean currentVertexAccepted = false;

// Initialise array freeVertex to True. Initially all vertices are
// available for migrations.
for (index=0; index<graphOrder; index++) {
    freeVertex[index] = true;
}

// Perform breadth first search re-ordering on graph
g.BFSReordering(Math.abs(numberGen.nextInt()) % graphOrder);

index = 1000;
testSolution = (Solution)currentSolution.clone();
while (index > 0) {
    newSolution = testSolution.randomLegalMigration();
    delta = newSolution.getCost() - testSolution.getCost();

    if (delta > 0) {
        initDelta += delta;
        index--;
    }
    testSolution = newSolution;
}
initDelta /= 1000;

// Calculate temperature based on average uphill delta

```

```

temperature = -initDelta/(Math.log(initProb));
System.out.println("# Temperature: " + temperature +
    " Cost: " + currentSolution.getCost() +
    " Cut Edges " + currentSolution.getCutEdges());

/* Vertices are picked at random without repetition.
 * When a vertex 'n' is picked false is recorded
 * in position n in the freeVertex array. When a vertex
 * is accepted the array is reset (to true).
 */
long startTime = System.currentTimeMillis();
activeVertex = Math.abs(numberGen.nextInt()) % graphOrder;
currentVertexAccepted = false;
while (minPercentCount < maxIterations) {
    targetPartition = Math.abs(numberGen.nextInt()) % k;

    // Try active vertex in all partitions starting with one
    // at random and advancing (and wrapping if necessary)
    while (partitionCount < k) {
        targetPartition++;
        targetPartition %= k;

        newSolution = currentSolution.legalMigration(activeVertex,
                                                    targetPartition);

        currentCost = currentSolution.getCost();
        newCost = newSolution.getCost();
        delta = (newSolution.getCost() - currentSolution.getCost());

        if (delta < 0) {
            currentSolution = (Solution)newSolution.clone();
            numberAcceptedMoves++;
            currentVertexAccepted = true;

            if (newCost < bestCost)
                bestSolution = (Solution)currentSolution.clone();
        } else if (delta > 0) {
            if (probabilityFunction(delta, temperature)) {
                currentSolution = (Solution)newSolution.clone();
                numberAcceptedMoves++;
                currentVertexAccepted = true;
            }
        }
    }

    // After trying vertex activeVertex with partition
    // partitionCount, try activeVertex with partitionCount + 1.
    partitionCount++;
    migrationCount++;
    totalIterations++;

    if (migrationCount == runLength) {
        ratioAcceptedMoves = (double)numberAcceptedMoves/runLength;
        if (ratioAcceptedMoves > minRatio) {
            minPercentCount = 0;
        } else {

```

```

        minPercentCount++;
    }

    // Reduce the temperature and reset the count
    temperature *= coolingRate;
    migrationCount = 0;
    numberAcceptedMoves = 0;
}
}
// Finished trying vertex: activeVertex against all partitions
// Reset the partitionCount and find a new vertex.

partitionCount = 0;

// If the vertex has been selected and cannot be migrated, it
// is no longer considered for further migration
// If it has been accepted, then all vertices are made available
// for migration again.
if (currentVertexAccepted == false) {
    // Make this vertex unavailable for selection
    freeVertex[activeVertex] = false;
} else {
    // Reset the 'availability' of vertices
    for (index=0; index<graphOrder; index++) {
        freeVertex[index] = true;
    }
}

/* If all the vertices have been tried then
 * make all vertices available again.
 */
if (verticesExamined == graphOrder) {
    for (index=0; index<graphOrder; index++) {
        freeVertex[index] = true;
    }
    verticesExamined = 0;
}

// Search for a vertex that has not already been tried
do {
    activeVertex = (Math.abs(numberGen.nextInt()) % graphOrder);
} while (!freeVertex[activeVertex]);
verticesExamined++;
}

finishTime = System.currentTimeMillis();
System.out.println("# Total running time in milliseconds: " +
    (finishTime - startTime) +
    ": Total Iterations: " + totalIterations);

System.out.println("# Best Solution: " + bestSolution.getCost() +
    " " + bestSolution.getCutEdges() +
    " " + bestSolution.getImbalance());
return(bestSolution);

```

```

}

private void doubleVertexMigrationAnnealing(GeometricGraph g,
                                           int k,
                                           double initProb,
                                           double coolingRate,
                                           float sizeFactor,
                                           double minRatio,
                                           Solution initSolution,
                                           int maxIterations) {

    numberGen = new Random();
    double temperature;
    double delta = 0;
    double initDelta = 0;
    double newCost = 0;
    long finishTime;
    int index = 0;
    int numberAcceptedMoves = 0;
    double ratioAcceptedMoves = 1;
    int minPercentCount = 0;
    Solution currentSolution = initSolution;
    Solution bestSolution;
    Solution newSolution;
    Solution testSolution;
    double currentCost = currentSolution.getCost();
    double bestCost = currentCost;
    int migrationCount = 0;
    int graphOrder = g.getOrder();
    int runLength = (int)(graphOrder * sizeFactor);
    int totalIterations = 0;
    bestSolution = (Solution)currentSolution.clone();
    boolean freeVertex[] = new boolean[graphOrder];
    boolean currentVertexAccepted = false;

    // Initialise array freeVertex to True. Initially all vertices are
    // available for migrations.
    for (index=0; index<graphOrder; index++) {
        freeVertex[index] = true;
    }

    // Perform breadth first search re-ordering on graph
    g.BFSReordering(Math.abs(numberGen.nextInt()) % graphOrder);

    index = 1000;
    testSolution = (Solution)currentSolution.clone();
    while (index > 0) {
        newSolution = testSolution.switchRandomVertices();
        delta = newSolution.getCost() - testSolution.getCost();

        if (delta > 0) {
            initDelta += delta;
            index--;
        }
    }
}

```

```

        testSolution = newSolution;
    }
    initDelta /= 1000;

    // Calculate temperature based on average uphill delta
    temperature = -initDelta/(Math.log(initProb));
    System.out.println(temperature + " " + currentSolution.getCost() +
        " " + currentSolution.getCutEdges());

    /* Vertices are picked at random without repetition.
     * When a vertex 'n' is picked false is recorded
     * in position n in the freeVertex array.
     * For each vertex, check if swapping it with every
     * other vertex can yield a gain.
     */
    long startTime = System.currentTimeMillis();
    int vertexCount = 0;
    int swapVertex = 0;
    int activeVertex = 0;

    while (minPercentCount < maxIterations) {
        swapVertex = Math.abs(numberGen.nextInt()) % graphOrder;
        activeVertex = Math.abs(numberGen.nextInt()) % graphOrder;

        while (vertexCount < graphOrder) {
            swapVertex++;
            swapVertex %= graphOrder;
            newSolution = currentSolution.switchVertices(activeVertex,
                swapVertex);

            currentCost = currentSolution.getCost();
            newCost = newSolution.getCost();
            delta = (newSolution.getCost() - currentSolution.getCost());
            migrationCount++;
            totalIterations++;

            if (delta < 0) {
                currentSolution = (Solution)newSolution.clone();
                vertexCount = 0;
                numberAcceptedMoves++;
                currentVertexAccepted |= true;

                if (newCost < bestCost)
                    bestSolution = (Solution)currentSolution.clone();
            } else if (delta > 0) {
                if (probabilityFunction(delta, temperature)) {
                    currentSolution = (Solution)newSolution.clone();
                    vertexCount = 0;
                    numberAcceptedMoves++;
                    currentVertexAccepted |= true;
                } else {
                    vertexCount++;
                }
            } else {
                // Don't explore plateaus
            }
        }
    }

```

```

        vertexCount++;
    }

    if (migrationCount == runLength) {
        ratioAcceptedMoves = (double)numberAcceptedMoves/runLength;
        if (ratioAcceptedMoves > minRatio) {
            minPercentCount = 0;
        } else {
            minPercentCount++;
        }

        temperature *= coolingRate;
        migrationCount = 0;
        numberAcceptedMoves = 0;
    }
}
vertexCount = 0;

/* If the vertex has been selected and cannot be migrated,
 * it is no longer considered for further migration
 * If it has been accepted, then all vertices are made
 * available for migration again.
 */

if (currentVertexAccepted == false) {
    freeVertex[activeVertex] = false;
} else {
    for (index=0; index<graphOrder; index++) {
        freeVertex[index] = true;
    }
}

// Search for a vertex that has not already been tried

do {
    activeVertex = (Math.abs(numberGen.nextInt()) % graphOrder);
} while (!freeVertex[activeVertex]);
}

finishTime = System.currentTimeMillis();
System.out.println("# Total running time in milliseconds: " +
    (finishTime - startTime) +
    ": Total Iterations: " + totalIterations);
System.out.println("# Best Solution: " + bestSolution.getCost() +
    " " + bestSolution.getCutEdges());
}

```

```
private boolean probabilityFunction(double delta, double temp) {

    double index = (-delta/temp);
    double probability = Math.exp(index)/2;
    int cutoffValue = (int)Math rint(1000000 * probability);
    int testValue = Math.abs(numberGen.nextInt() % 1000000);

    if (testValue < cutoffValue) {
        return(true);
    } else {
        return(false);
    }
}

public static void main(String[] args) {

    int numberOfArgs = args.length;
    int i = 0;
    String tmpString = new String();
    GeometricGraph g = (GeometricGraph)null;
    int partition = 0;
    double initProb = 0;
    double rate = 0;
    float sizeFactor = 0;
    double minRatio = 0;
    double imbalanceScaling = Integer.MIN_VALUE;
    String filename = "";
    int maxIterations = 0;
    boolean singleVertexMigrations = false;

    if ((numberOfArgs != 15) &&
        (numberOfArgs != 17) &&
        (numberOfArgs != 19)) {

        System.err.println("Error:");
        System.err.println("SA -g GraphFile
            -p Partition Size
            [-sol Solution File]
            -i initProb
            -l Size Factor
            -r Cooling Rate
            -f Freeze Percent
            [-s Imbalance Scaling]
            -mx maxIterations
            -[single|double]");
    }
}
```

```

} else {
    while (i < numberOfArgs) {

        tmpString = args[i];
        if (tmpString.equals("-g")) {
            // System.out.print("Reading graph from file...");
            g = GeometricGraph.readFromFile(args[++i]);
            // System.out.println("done");
        } else if (tmpString.equals("-p")) {
            partition = new Integer(args[++i]).intValue();
        } else if (tmpString.equals("-i")) {
            initProb = new Double(args[++i]).doubleValue();
        } else if (tmpString.equals("-l")) {
            sizeFactor = new Float(args[++i]).floatValue();
        } else if (tmpString.equals("-r")) {
            rate = new Double(args[++i]).doubleValue();
        } else if (tmpString.equals("-f")) {
            minRatio = new Double(args[++i]).doubleValue();
        } else if (tmpString.equals("-s")) {
            imbalanceScaling = new Double(args[++i]).doubleValue();
        } else if (tmpString.equals("-sol")) {
            filename = args[++i];
        } else if (tmpString.equals("-mx")) {
            maxIterations = new Integer(args[++i]).intValue();
        } else if (tmpString.equals("-single")) {
            singleVertexMigrations = true;
            if (imbalanceScaling == Integer.MIN_VALUE) {
                System.err.println("Error: a value for Imbalance
                    Scaling must be supplied for
                    single vertex migrations");

                System.exit(-1);
            }
        } else if (tmpString.equals("-double")) {
            singleVertexMigrations = false;
            if (imbalanceScaling != Integer.MIN_VALUE)
                System.out.println("#Warning: Value for Imbalance
                    Scaling will be ignored!");
        } else {
            System.err.println("Error: arg: " + args[i]);
            System.exit(1);
        }
        i++;
    }

    System.out.println("# Running SA on graph of order " +
        g.getOrder() +
        " and size " + g.getSize());
    System.out.println("# graph Connected: " + g.isConnected());

```

```
    if (filename.equals("")) {
        SA s = new SA(g,
            partition,
            initProb,
            rate,
            sizeFactor,
            minRatio,
            imbalanceScaling,
            maxIterations,
            singleVertexMigrations);
    } else {
        SA s = new SA(g,
            partition,
            initProb,
            rate,
            sizeFactor,
            minRatio,
            imbalanceScaling,
            filename,
            maxIterations,
            singleVertexMigrations);
    }
}
}
```

A.13 Solution

```

/*
 * Solution.class
 * Part of package project
 *
 * Author: Philip Bradley
 *
 */

package project;

import java.util.Random;
import java.io.*;

public class Solution implements Cloneable, Serializable {

    private static GeometricGraph graph;
    private static int[] [] adjacencyMatrix;
    private int[] partitionVector;
    private int[] partitionImbalance;
    private Random numberGen;
    private int order = 0;
    private int partitionSize = 0;
    private int partitionFactor = 0;
    private int cutEdges = 0;
    private double alpha = 1;

    public Solution(GeometricGraph g, int k) {
        graph = g;
        order = graph.getOrder();

        if ((order % k) == 0) {
            partitionVector = new int[order];
            adjacencyMatrix = graph.getAdjacencyMatrix();
            partitionImbalance = new int[k];
            partitionFactor = k;
            partitionSize = (int)order/k;

            numberGen = new Random();

            /* Initialise partitionVector, If partitionVector[i] == k
             * then vertex i is in partition k.
             * For each partition p 0<=p<k, pick a vertex v at
             * random without repetition and assign p to it.
             */
            int splitSize = 0;
            int index = 0;
            int i = 0;
            int j = 0;

            i = order;
            while (i-- > 0)
                partitionVector[i] = 0;

```

```
// Start at 1 because all vertices are in partition 0 by default
i = 0;
while (++i < partitionFactor) {
    splitSize = 0;
    while (splitSize < partitionSize) {
        index = Math.abs(numberGen.nextInt() % order);
        if (partitionVector[index] == 0) {
            partitionVector[index] = i;
            splitSize++;
        }
    }
}

// Initialise partitionImbalance vector
i = partitionFactor;
while (i-- > 0) {
    // Initialise partition sizes
    partitionImbalance[i] = 0;
}

// Initialise cutEdges
for (i=0; i<order; i++) {
    for (j=i; j<order; j++) {
        if (g.hasEdge(i,j)) {
            if (partitionVector[i] != partitionVector[j]) {
                // vertices are in different partitions
                cutEdges++;
            }
        }
    }
}

} else {
    System.err.println("Error: The graph may only be divided evenly!");
}
}
```

```

public Solution(GeometricGraph g, int k, int[] pvector) {

    graph = g;
    order = graph.getOrder();
    int index;
    int i=0;
    int j=0;
    numberGen = new Random();

    if (pvector.length == order) {
        if ((order % k) == 0) {
            partitionVector = pvector;
            adjacencyMatrix = graph.getAdjacencyMatrix();
            partitionImbalance = new int[k];
            partitionFactor = k;
            partitionSize = (int)order/k;

            // Initialise partitionImbalance vector
            index = partitionFactor;
            while (index-- > 0) {
                partitionImbalance[index] = -partitionSize;
            }

            // Calculate imbalance of each partition and total external cost
            index = partitionVector.length;
            while(index-- > 0)
                partitionImbalance[partitionVector[index]]++;

            // Initialise cutEdges
            for (i=0; i<order; i++) {
                for (j=i; j<order; j++) {
                    if (g.hasEdge(i,j)) {
                        if (partitionVector[i] != partitionVector[j]) {
                            // vertices are in different partitions
                            cutEdges++;
                        }
                    }
                }
            }
        }
    }

    /**
     * Used to initialise a Solution if has been read
     * from a file and no other instance of Solution exists.
     */
    public void setGraph(GeometricGraph g) {
        graph = g;
        adjacencyMatrix = g.getAdjacencyMatrix();
    }
}

```

```

/**
 * @return The array representing the partitioning of the graph
 */
public int[] getPartition() {
    return(partitionVector);
}

public void setPartition(int[] pvector) {

    int index;
    int i=0;
    int j=0;

    if (pvector.length == order) {
        System.arraycopy(pvector, 0,
            partitionVector, 0,
            partitionVector.length);

        // Initialise partitionImbalance vector
        index = partitionFactor;
        while (index-- > 0) {
            partitionImbalance[index] = -partitionSize;
        }

        // Calculate imbalance of each partition and total external cost
        index = partitionVector.length;
        while(index-- > 0)
            partitionImbalance[partitionVector[index]]++;

        // Recalculate cutEdges
        cutEdges = 0;
        for (i=0; i<order; i++) {
            for (j=i; j<order; j++) {
                if (graph.hasEdge(i,j)) {
                    if (partitionVector[i] != partitionVector[j]) {
                        // vertices are in different partitions
                        cutEdges++;
                    }
                }
            }
        }
    }
}

```

```
/**
 * This changes the factor by which the imbalance
 * of the partitioning is scaled.
 *
 * @see getImbalance()
 */
public void setImbalanceWeight(double w) {
    // This is the factor by which the imbalance is scaled.
    alpha = w;
}

public void printStatus() {
    // Test code
    int index;
    int arrayLength = 0;

    System.out.println("Partition Imbalance");
    System.out.println("=====");
    index = -1;
    arrayLength = partitionImbalance.length;
    while (++index < arrayLength)
        System.out.print(partitionImbalance[index] + " ");
    System.out.println("");

    System.out.println("Partition Cost");
    System.out.println("=====");
    index = -1;

    System.out.println("Cutedges: " + cutEdges);
    System.out.println("Solution Cost: " + getCost());
    System.out.println("-----");
    System.out.println("");
}
```

```

public Object clone() {
    Solution s = (Solution)null;
    int length;
    try {
        // This is the basic clone step
        s = (Solution)super.clone();

        // Need to clone all the non primitive members also.
        length = partitionVector.length;
        s.partitionVector = new int[length];
        System.arraycopy(partitionVector, 0,
            s.partitionVector, 0,
            length);

        length = partitionImbalance.length;
        s.partitionImbalance = new int[length];
        System.arraycopy(partitionImbalance, 0,
            s.partitionImbalance, 0,
            length);

    } catch (CloneNotSupportedException e) {
        System.err.println("System error: Clone failed");
        System.err.println(e.getMessage());
    }
    return(s);
}

/**
 * @return A copy of the Solution in which a vertex has been
 * moved to a different partition.
 */
public Solution randomLegalMigration() {
    int vertex = Math.abs(numberGen.nextInt() % order);
    int sourcePartition = partitionVector[vertex];
    int destinationPartition =
        Math.abs(numberGen.nextInt() % partitionFactor);

    while (sourcePartition == destinationPartition) {
        destinationPartition = Math.abs(numberGen.nextInt() % partitionFactor);
    }

    Solution newSolution = (Solution)null;

    try {
        newSolution = (Solution)this.clone();
        newSolution.migrate(vertex, sourcePartition, destinationPartition);
    } catch (Exception e) {
        System.err.println("Clone Error: " + e.getMessage());
    }
    return(newSolution);
}

```

```

/**
 * @return A copy of the Solution in which the input
 * vertex has been moved to the input partition.
 */
public Solution legalMigration(int vertex, int destinationPartition) {
    int sourcePartition = partitionVector[vertex];

    if (sourcePartition == destinationPartition) {
        return(this);
    } else {
        Solution newSolution = (Solution)null;
        try {
            newSolution = (Solution)this.clone();
            newSolution.migrate(vertex, sourcePartition, destinationPartition);
        } catch (Exception e) {
            System.err.println("Clone Error: " + e.getMessage());
        }
        return(newSolution);
    }
}

private void migrate(int vertex,
                    int sourcePartition,
                    int destinationPartition) {

    Random numberGen = new Random();
    int vertexInternalCost = 0;
    int vertexExternalCost = 0;
    int vertexExternalCostWrtTarget = 0;
    int currentPartition = 0;

    for (int i=0; i<order; i++) {
        if ((adjacencyMatrix[i][vertex] != 0) && (vertex != i)) {
            currentPartition = partitionVector[i];
            if (currentPartition == sourcePartition) {
                vertexInternalCost++;
            } else {
                if (currentPartition == destinationPartition) {
                    // The cost with respect to the destination partition
                    vertexExternalCostWrtTarget++;
                }
            }
        }
    }

    // Perform the actual migration
    partitionVector[vertex] = destinationPartition;
    partitionImbalance[sourcePartition] -= 1;
    partitionImbalance[destinationPartition] += 1;
    cutEdges += (vertexInternalCost - vertexExternalCostWrtTarget);
}

```

```

/**
 * @return A copy of the Solution in
 * which the input vertices are exchanged.
 */
public Solution switchVertices(int vertex0, int vertex1) {

    Solution newSolution = (Solution)null;
    try {
        newSolution = (Solution)this.clone();
    } catch(Exception e) {
        System.err.println("Clone Error: " + e.getMessage());
    }

    try {
        newSolution.swapVertices(vertex0, vertex1);
    } catch(BadVertexSwapException e) {
        // Tried to swap two vertices which are in same partition.
        // This is a Null operation.
    }
    return(newSolution);
}

/**
 * This performs switchVertices with two
 * random vertices from different partitions.
 */
public Solution switchRandomVertices() {
    int vertex0 = Math.abs(numberGen.nextInt()) % order;
    int vertex1 = Math.abs(numberGen.nextInt()) % order;

    // Make sure both vertices are from different partitions
    while (partitionVector[vertex0] == partitionVector[vertex1]) {
        vertex1++;
        vertex1 %= order;
    }
    return(switchVertices(vertex0, vertex1));
}

private void swapVertices(int vertex0,
                          int vertex1)
    throws BadVertexSwapException {

    Random numberGen = new Random();
    int vertex0InternalCost = 0;
    int vertex0ExternalCost = 0;
    int vertex0ExternalCostWrtTarget = 0;
    int vertex1InternalCost = 0;
    int vertex1ExternalCost = 0;
    int vertex1ExternalCostWrtTarget = 0;
    int vertex0Partition = partitionVector[vertex0];
    int vertex1Partition = partitionVector[vertex1];
    int currentPartition = 0;

```

```

if ((partitionVector[vertex0] == partitionVector[vertex1]) ||
    (vertex0 == vertex1)) {
    throw(new BadVertexSwapException());
} else {
    // Calculate Internal and External cost of each vertex, and the
    // external cost of each with respect to the target partition.
    for (int i=0; i<order; i++) {
        if ((adjacencyMatrix[i][vertex0] != 0) && (i != vertex0)) {
            currentPartition = partitionVector[i];
            if (currentPartition == vertex0Partition) {
                vertex0InternalCost++;
            } else {
                if (currentPartition == vertex1Partition) {
                    // The cost with respect to the destination partition
                    vertex0ExternalCostWrtTarget++;
                } else {
                    // The cost wrt to all other partitions
                    vertex0ExternalCost++;
                }
            }
        }
    }

    if ((adjacencyMatrix[i][vertex1] != 0) && (i != vertex1)) {
        currentPartition = partitionVector[i];
        if (currentPartition == vertex1Partition) {
            vertex1InternalCost++;
        } else {
            if (currentPartition == vertex0Partition) {
                // The cost with respect to the destination partition
                vertex1ExternalCostWrtTarget++;
            } else {
                // The cost wrt to all other partitions
                vertex1ExternalCost++;
            }
        }
    }
}

// Perform the actual migration
partitionVector[vertex0] = vertex1Partition;
partitionVector[vertex1] = vertex0Partition;

cutEdges += ((vertex0InternalCost - vertex0ExternalCostWrtTarget) +
             (vertex1InternalCost - vertex1ExternalCostWrtTarget));
cutEdges += (2 * adjacencyMatrix[vertex0][vertex1]);
}
}

```

```

public void balance() {
    // Redistribute the nodes so that each partition has the same amount.
    int partitionIndex = 0;
    int vertexIndex=0;

    for(int i=0; i<partitionFactor; i++) {
        // Search for a partition with too few vertices
        while((partitionIndex < partitionFactor) &&
            (partitionImbalance[partitionIndex] >= 0)) {
            partitionIndex++;
        }

        if (partitionIndex < partitionFactor) {
            // Found a partition with too few vertices
            while((partitionImbalance[partitionIndex] < 0) &&
                (partitionImbalance[i] > 0)) {
                int maxVertices = (partitionImbalance[i] -
                    partitionImbalance[partitionIndex]);
            }
        }
    }
}

private void balancePartitions(int srcPartition, int destPartition) {
    int maxVertices = (partitionImbalance[srcPartition] -
        partitionImbalance[destPartition]);
    int gainList[] = new int[order];
    int vertexCount = 0;
    int maxGain = Integer.MIN_VALUE;
    int swapVertex = 0;
    int i=0;

    // Set all values to min
    while(i<order)
        gainList[i++] = Integer.MIN_VALUE;

    // Calculate gain for each vertex in srcPartition
    for (i=0; i<order; i++) {
        if (partitionVector[i] == srcPartition) {
            gainList[vertexCount++] = gain(vertexCount, destPartition);
        }
    }

    // find vertex yielding maximum gain
    i=0;
    while (gainList[i] != Integer.MIN_VALUE) {
        if (gainList[i++] > maxGain)
            swapVertex=i;
    }
}

```

```

private int gain(int partition, int vertex) {
    return (2 * (getExternalCostWRT(vertex,partition) -
                getInternalCost(vertex)));
}

private int getInternalCost(int vertex) {
    // Returns the internal cost of a vertex

    int internalCost = 0;
    for(int n=0; n<order; n++) {
        if ((graph.hasEdge(vertex, n)) &&
            (partitionVector[n] == partitionVector[vertex])) {
            internalCost++;
        }
    }
    return(internalCost);
}

private int getExternalCostWRT(int vertex, int targetPartition) {

    int vertexPartition = partitionVector[vertex];
    int cost = 0;

    for(int n=0; n<order; n++) {
        if (graph.hasEdge(vertex,n)) {
            if (partitionVector[n] == targetPartition){
                cost++;
            }
        }
    }
    return(cost);
}

/**
 * Crossover consists of taking two vectors, breaking each into two
 * segments and recombining opposite segments to produce two new vectors.
 *
 * @param mate A Solution with which to perform the crossover operation.
 * @param crossOverPoint The point in the chromosome
 *                        at which to perform crossover
 * @return The two derived Solutions contained in an array.
 */
public Solution[] singlePointCrossOver(Solution mate, int crossOverPoint) {
    Solution[] children = new Solution[2];
    int[] childVector0 = new int[order];
    int[] childVector1 = new int[order];

    System.arraycopy(this.partitionVector, 0,
                     childVector0, 0,
                     crossOverPoint);
    System.arraycopy(mate.partitionVector, 0,
                     childVector1, 0,
                     crossOverPoint);
}

```

```

    System.arraycopy(mate.partitionVector, crossOverPoint,
                    childVector0, crossOverPoint,
                    (order - crossOverPoint));
    System.arraycopy(this.partitionVector, crossOverPoint,
                    childVector1, crossOverPoint,
                    (order - crossOverPoint));
    children[0] = new Solution(this.graph, partitionFactor, childVector0);
    children[1] = new Solution(this.graph, partitionFactor, childVector1);
    return(children);
}

/**
 * This performs singlePointCrossOver at a random legal position.
 * @return The two derived Solutions contained in an array.
 */
public Solution[] randomSinglePointCrossOver(Solution mate) {
    int crossOverPoint = Math.abs(numberGen.nextInt() % order);
    return(singlePointCrossOver(mate, crossOverPoint));
}

/**
 * Mutation consists of changing the values in a
 * segment of a chromosome in some way.
 *
 * @param startPoint The position in the Chromosome
 *                  at which to start the mutation.
 * @param blockLength The length of the block to be mutated.
 *
 * @return A copy of the Solution with the block mutated.
 */
public Solution mutate(int startPoint, int blockLength) {

    int finishPoint = startPoint + blockLength;
    int i;
    int[] newVector = new int[order];
    System.arraycopy(partitionVector, 0, newVector, 0, order);

    if (finishPoint < order) {
        for (i=startPoint; i<=finishPoint; i++) {
            newVector[i] %= partitionFactor;
        }
    } else {
        for (i=startPoint; i<order; i++) {
            newVector[i] %= partitionFactor;
        }
        finishPoint %= order;
        for (i=startPoint; i<=finishPoint; i++) {
            newVector[i] %= partitionFactor;
        }
    }
    return(new Solution(this.graph, partitionFactor, newVector));
}

```

```

/**
 * Adjust the input vector so that all values occur equal numbers
 * of times thus representing a balanced solution.
 */
public void postProcess() {

    // Redistribute the nodes so that each partition has the same amount.
    int partitionIndex = 0;
    for(int i=0; i<partitionFactor; i++) {
        // Search for a partition with too many vertices
        while((partitionIndex < partitionFactor) &&
            (partitionImbalance[partitionIndex] <= 0)) {
            partitionIndex++;
        }

        if (partitionIndex < partitionFactor) {
            // Found a partition with surplus vertices
            while((partitionImbalance[partitionIndex] > 0) &&
                (partitionImbalance[i] < 0)) {
                // Search for a vertex in partition partitionIndex
                int vertex = Math.abs(numberGen.nextInt() % order);
                while (partitionVector[vertex] != partitionIndex) {
                    vertex++;
                    vertex %= order;
                }
                // Move the vertex to partition i.
                partitionVector[vertex] = i;
                partitionImbalance[partitionIndex]--;
                partitionImbalance[i]++;
            }
        }
    }

    // Recalculate cutEdges
    int i=0;
    int j=0;
    cutEdges = 0;
    for (i=0; i<order; i++) {
        for (j=i; j<order; j++) {
            if (graph.hasEdge(i,j)) {
                if (partitionVector[i] != partitionVector[j]) {
                    // vertices are in different partitions
                    cutEdges++;
                }
            }
        }
    }

    // Reset imbalance vector
    for (i=0; i<partitionFactor; i++) {
        partitionImbalance[i] = 0;
    }
}

```

```
/**
 * @return The number of edges cut in the partitioning
 */
public int getCutEdges() {
    return(cutEdges);
}

/**
 * @return The sum of the squares of the differences
 * between the size of the partitions and the average size.
 */
public int getImbalance() {
    int index = partitionFactor;
    int imbalanceCost = 0;
    while (index-- > 0)
        imbalanceCost += Math.pow(partitionImbalance[index],2);

    return(imbalanceCost);
}

/**
 * @return The number of edges cut + the imbalance of the partition
 */
public double getCost() {
    return(cutEdges + (alpha * getImbalance()));
}

/**
 * @return The partition to which the input vertex belongs.
 */
public int getVertexPartition(int vertex) {
    return(partitionVector[vertex]);
}

public void writeToFile(String filename) {
    try {
        ObjectOutputStream os =
            new ObjectOutputStream(new FileOutputStream(filename));
        os.writeObject(this);
        os.close();
    } catch (Exception e) {
        System.err.println("Error writing object: " + e.getMessage());
    }
}
```

```
/**
 * @param filename The name of a file containing a Serialized Solution
 *
 * @return The Solution read from the file.
 */
public static Solution readFromFile(String filename) {
    Solution s = (Solution)null;
    try {
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream(filename));
        s = (Solution)in.readObject();
    } catch (Exception e) {
        System.err.println("Error reading object: " + e.getMessage());
    }
    return(s);
}

private void testCrossOver(Solution s) {
    Solution[] offspring = new Solution[2];
    System.out.println("Crossing to following vectors..");

    int i=0;
    for (i=0; i<order; i++) {
        System.out.print(this.partitionVector[i] + " ");
    }
    System.out.println();
    for (i=0; i<order; i++) {
        System.out.print(s.partitionVector[i] + " ");
    }
    System.out.println();

    offspring = randomSinglePointCrossOver(s);
    System.out.println("Derived vectors");
    for (i=0; i<order; i++) {
        System.out.print(offspring[0].partitionVector[i] + " ");
    }
    System.out.println();
    for (i=0; i<order; i++) {
        System.out.print(offspring[1].partitionVector[i] + " ");
    }
    System.out.println();
}
```

```
public static void main(String[] args) {

    if (args.length == 3) {
        GeometricGraph g = GeometricGraph.readFromFile(args[0]);
        int k = new Integer(args[1]).intValue();
        String solutionFileName = args[2];

        Solution s = new Solution(g,k);
        System.out.print("Writing to file.....");
        s.writeToFile(solutionFileName);
        System.out.println("done");
        System.out.println("");
        System.out.print("Reading Solution from file....");
        s = Solution.readFromFile(solutionFileName);
        s.setGraph(g);
        System.out.println("done");
    } else {
        System.err.println("Usage Error: Solution GraphFile k SolutionFile");
    }
}
}
```

A.14 CreateGraph

```
import java.io.*;
import java.util.*;
import project.GeometricGraph;

public class CreateGraph {

    GeometricGraph g;
    BufferedReader in;

    CreateGraph(int order) {
        g = new GeometricGraph(order);
        try {
            in = new BufferedReader(new InputStreamReader(System.in));
            String currentInput = new String("");

            while ((!currentInput.equals("q")) &&
                    (!currentInput.equals("quit")) &&
                    (!currentInput.equals("exit"))) {

                System.out.print(">");
                currentInput = in.readLine();
                parse(currentInput);
            }
        } catch (Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

```

private void parse(String input) {
    try {
        StringTokenizer st = new StringTokenizer(input);
        String arg0 = new String();
        String arg1 = new String();
        String arg2 = new String();
        String fileName = new String("");

        if (st.countTokens() == 3) {
            arg0 = st.nextToken();
            arg1 = st.nextToken();
            arg2 = st.nextToken();

            if ((arg0.equals("addEdge")) ||
                (arg0.equals("a")) ||
                (arg0.equals("add"))) {

                System.out.println("Adding edge " + arg1 + "," + arg2);
                g.addEdge(new Integer(arg1).intValue(),
                    new Integer(arg2).intValue());
            }
        } else {
            if ((input.equals("q")) ||
                (input.equals("quit")) ||
                (input.equals("exit"))) {
                System.exit(0);
            } else if (input.equals("save")) {
                System.out.print("Filename: ");
                fileName = in.readLine();
                g.writeToFile(fileName);
                System.out.println("Saved to file " + fileName);
            } else if ((input.equals("s")) || (input.equals("size"))) {
                System.out.println(g.getSize());
            } else if ((input.equals("edges")) || (input.equals("e"))) {
                g.printEdges();
            } else if ((input.equals("h")) || (input.equals("help"))) {
                System.out.println("addEdge|add|a i j");
                System.out.println("quit|exit|q");
                System.out.println("save filename");
                System.out.println("s|size");
                System.out.println("e|edges");
                System.out.println("h|help");
            } else if (input.equals("")) {
                // null
            } else {
                System.out.println("Unknown command: " + input);
            }
        }
    } catch (Exception e) {
        System.out.println("Exception: " + e);
        System.exit(1);
    }
}

```

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.err.println("Error: CreateGraph order");
        System.exit(1);
    } else {
        System.out.println("        Testing...");
        CreateGraph c = new CreateGraph(new Integer(args[0]).intValue());
    }
}
```