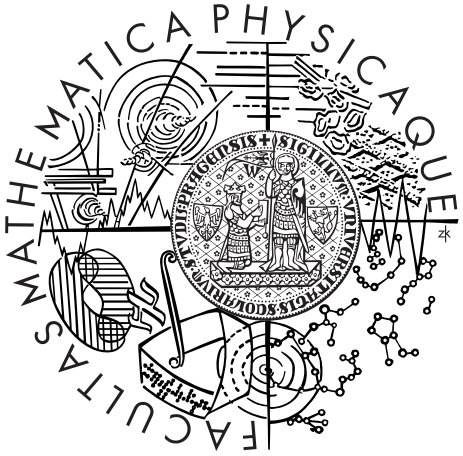
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Jan Kohout

Graph Clustering by Means of Evolutionary Algorithms

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Roman Neruda, CSc.
Study programme:	Computer Science
Specialization:	Theoretical Computer Science

Prague 2012

I would like to greatly thank my supervisor, Mgr. Roman Neruda, CSc., for the valuable advices and support during my work on this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, July 24, 2012 Jan Kohout

Název práce: Využití evolučních algoritmů pro shlukování v grafech.

Autor: Bc. Jan Kohout

Katedra: Katedra teoretické informatiky a matemtické logiky

Vedoucí diplomové práce: Mgr. Roman Neruda, CSc., Ústav informatiky AV ČR, v. v. i.

Abstrakt: Rozdělení vrcholů grafu do shluků podle jejich vzájemné podobnosti je užitečnou součástí datové analýzy, muže však být značně složité. Pro tento problem existuje mnoho různých algoritmů a přístupů, jedna z možností je využití genetických algoritmů. V této práci se zabýváme shlukovacími algoritmy obecně i v oblasti grafů. Navrhli jsme několik algoritmů založených na myšlence genetického algoritmu a tyto algoritmy byly porovnány na základě provedených experimentů. V rámci práce byla též vytvořena serverová aplikace obsahující implementace navržených algoritmů.

Klíčová slova: graf, shlukování, evoluční algoritmy, genetické algoritmy

Title: Graph Clustering by Means of Evolutionary Algorithms

Author: Bc. Jan Kohout

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Institute of Computer Science, Academy of Sciences of the Czech Republic

Abstract: Partitioning nodes of a graph into clusters according to their similarities can be a very useful but complex task of data analysis. Many different approaches and algorithms for this problem exist, one of the possibilities is to utilize genetic algorithms for solving this type of task. In this work, we analyze different approaches to clustering in general and in the domain of graphs. Several clustering algorithms based on the concept of genetic algorithm are proposed and experimentally evaluated. A server application that contains implementations of the these algorithms was developed and is attached to this thesis.

Keywords: graph, clustering, genetic algorithms, evolutionary algorithms

Contents

1. Introduction

The goal of a clustering task in general is to divide a given set of objects into subsets (clusters) such that objects from same subset are more similar to each other than objects that come from different subsets. This fact is expressed by an elementary clustering paradigm which says: "Intra-cluster homogeneity, inter-cluster heterogeneity" (as mentioned in [6]). The measure of similarity depends on the type of objects that should be clustered (if the objects are points in an Euclidean space the similarity measure can be, for example, the Euclidean distance between them). In case of graphs the objects that should be organized into homogenous clusters are nodes (vertices) of the graph. The measure of similarity of the nodes is then given by edges of that graph. If there is an edge between two nodes, these nodes are considered to be more similar than nodes that have not an edge between them. If we need more precision in measuring similarity of the nodes we can use weighted edges (the higher the weight the more similar the nodes are). According to this kind of similarity definition there should be higher density of edges inside the clusters than outside the clusters. Unlike in the case of metric spaces, it can be difficult to say how much similar each two nodes are. In Euclidean spaces, the distance metric is defined for each possible pair of objects. In graphs, we can usually measure similarity between adjacent nodes easily (using the weight of the edge between them), but it can be hard to design a reliable similarity measure for all nodes. So, the algorithms designed for clustering in metric spaces can't be easily modified to work properly on graph nodes. Because of this, some new algorithms are designed for graph clustering.

Graph clustering can have many applications in area of data analysis and exploration, especially when the analyzed data have naturally structure of a graph. For example, it can be utilized for communities detection in social networks or for organizing world wide web pages into groups for improving searching results among them. Another applications can be found in the field of computer networks.

If we need to specify the problem of clustering more preciously, we need some objective function which can be used for evaluation of any given clustering. The task of searching for the best clustering then becomes an optimization task of searching for global optimum of this objective function. Optimization problems of this kind are often NP-hard and so it can be in case of graph clustering. One way how to find feasible solutions of such hard search problems are evolutionary algorithms where the search process is realized by evolution of population consisting of possible solutions of a given problem. The evolutionary process is more or less inspired by the real biological evolution which should lead the members of the evolved population to approximate the unknown optimal solution better and better. One type of evolutionary algorithms is the genetic algorithm where the solutions are encoded using strings of numbers and evolved using evolutionary operators like selection, mutation and crossover.

The goal of this work is to design few genetic algorithms for solving graph clustering problems and compare and evaluate their performance on a set of graphs.

The structure of this thesis is following: In the first chapter, we analyze

clustering problems from a general point of view, especially the graph clustering problems. We review some different approaches to solving these problems, both evolutionary and non-evolutionary. In further sections of the chapter, we focus on genetic algorithms and their usage for clustering. In the second chapter, we propose several algorithms (called clusterizers) based on the idea of genetic algorithms and compare them by means of their main ideas and operators used. In the third chapter, we describe an application which was created as a part of this work. The application is used for practical demonstration of the algorithms proposed in second chapter. In the last chapter, we evaluate some experiments that were performed with the application to test the presented clusterizers and their configurations.

The created application (and its source code), other software mentioned in this work as well as files with graphs used for experiments are stored on the attached CD.

2. Graph clustering problem

2.1 Clustering in general

2.1.1 Basics

Clustering task's goal is to partition a given set of data objects into a finite set of groups according to similarities among the objects. These groups are usually called clusters. This can help with understanding the data, categorizing them and with finding new structure in the data which can be difficult to see without any processing. For example, clustering can be helpful when used for grouping search results at multiple levels or for detecting communities of users or customers with common interests.

Partitioning objects into clusters can be done in a couple of ways. In all cases, by clustering (partitioning) objects from a given set S into k clusters, we want to find subsets $S_1, ..., S_k$ such that:

1. $\forall i : S_i$ is nonempty
2. $S_1 \cup ... \cup S_k = S$

Furthermore, according to [8], we can distinguish non-overlapping and overlapping clustering. If the partitioning of objects into subsets $S_1, ..., S_k$ satisfies following condition:

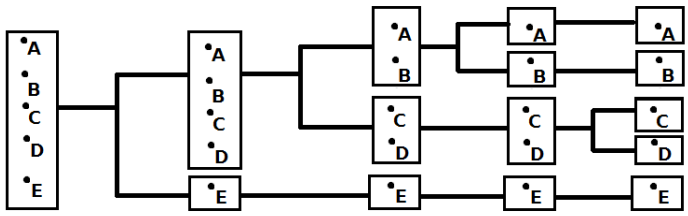
$$\forall i, j, i \neq j : \quad |S_i \cap S_j| = 0$$

we call such clustering *non-overlapping* or *exclusive*. Otherwise, we call the clustering *overlapping*. In other words, in *non-overlapping* clustering each object from the initial set S belongs exactly to one cluster, in *overlapping* clustering, one object can belong to more clusters than just one. *Overlapping* clustering can be further distinguished to *soft-clustering* and *fuzzy-clustering*. In *soft*-clustering, each object belongs fully to one or more clusters, in *fuzzy-clustering* additional degree of belonging is specified for each object and each cluster. The higher the degree, the more the object belongs to the cluster. Usually, we want the degrees of belonging to be within range $[0, 1]$ and that for one fixed object they sum to 1 over all clusters. In this case, the degree of belonging for object x_i to cluster S_j can be viewed as a probability that $x_i \in S_j$.

Another aspect of any clustering task is whether we have any previous information about the value of k (i.e., the number of clusters). In case of *non-overlapping* clustering, we can always assume that $k \leq |S|$ because for $k > |S|$, the condition (1) from the definition of clustering above can't be satisfied. Typically, there is $k << |S|$. Sometimes, the character of the clustering task gives us the exact value of k. Some clustering algorithms (discussed later) that are designed for fixed-size clustering (i.e., for clustering tasks where the number of clusters is given beforehand) need to know the value of k a priori. If the k is not known from the assignment, the proper value of k is a part of solution of the clustering task.

Finally, the clustering solutions can be serialized into sequence of nested clusterings to create so called hierarchical clustering. Clustering at the lower level of the hierarchy is produced from the clustering at the upper level by splitting

Figure 2.1 An example of dendrogram showing hierarchical clustering.



one chosen cluster into two new clusters. One example of hierarchical clustering performed on the set of five objects $S = \{A, B, C, D, E\}$ is shown in the figure 2.1.

In this work, we will focus our attention to non-overlapping clusterings in the domain of directed weighted graphs.

2.1.2 Different approaches to clustering

To fulfill the elementary aim of clustering, the combination of sets $S_1, ..., S_k$ can not be chosen arbitrarily. Depending on the nature of objects that are being clustered, various functions for judging quality of the clustering can be defined. Independently on the concrete chosen function, all clustering quality measures should review the clustering solution in a way that intra-cluster homogeneity of the objects should be maximized while inter-cluster similarity is minimized. Let us introduce a definition of a general clustering measure:

Definition. *Let S be a set of objects and $\boldsymbol{C}$ a set of all possible clusterings of S. A **clustering quality measure** is a function f:*

$$f : \boldsymbol{C} \to \boldsymbol{R}$$

which assigns each possible clustering a real number that represents its quality.

When a clustering quality measure is defined, we can search for any optimal clustering $C \in \mathbf{C}$ of the set S according to the given quality measure.

Definition. *Let S be a set of objects, $\boldsymbol{C}$ a set of all possible clusterings of S and f a clustering quality measure. If $C \in \boldsymbol{C}$ satisfies:*

$$C \in \arg\max_{\boldsymbol{C}} f$$

*we call C an **optimal** clustering of S.*

Alternatively, depending on the definition of the function, the optimal clustering can be the clustering where f reaches its minimum (instead of maximum).

This way we can see the problem of clustering from the optimization perspective. Searching for the optimal clustering becomes an optimization task of the function which is used as a clustering quality measure. In general case, this optimization task can be NP-complete problem (depending on the quality measure used, proofs can be found in [10]). Another problem of clustering can be finding

the proper function to be used as quality measure. The definition of good clustering in agreement with the clustering paradigm is rather vague, thus, an exact formula for any usable quality measure may not be straightforward. These characteristics make clustering relatively complex problem. Because of that, artificial intelligence methods are usually involved, especially from the field of machine learning or evolutionary algorithms.

In some domains, formulation of a suitable function for clustering quality evaluation is easier. If the objects from the set S can be viewed as points in n-dimensional metric space, similarity of two objects can be expressed as a distance between these objects. Thus, the clustering quality measure can be based on the metric (i.e., the distance measure) used in this metric space. The text [8] presents few examples of clustering quality measures proposed in several works on clustering topic. For a clustering solution $C = \{S_1, ..., S_k\}$ in a metric space, clusters can be represented by n-dimensional vectors from this space. Then, an object x_i from the set S belongs the cluster represented whose representing vector is the nearest to x_i (among representing vectors of all clusters). Let us denote $s_1, ..., s_k$ the representing vectors for clusters $S_1, ..., S_k$. With this representation of clusters, we can define the quality measure subsequently:

$$f(C = \{S_1, ..., S_k\}) = \sum_{j=1}^{k} \sum_{x_i \in S_j} ||x_i - s_j||^p$$

In this case, representing vectors can be mean vectors of the clusters. The optimal clustering is such clustering that minimizes f, p is a parameter of f, in example presented in [8], there is $p = 2$. Another function, also shown in the survey [8], works with a set of so called medoids for representing clusters. The set of medoids $m_1, ..., m_k$ consists of k chosen objects from the set S. Each medoid represents one cluster and again, the cluster which any object belongs to is given by the medoid that is the closest to the object. The objective function f is following:

$$f(C = \{S_1, ..., S_k\}) = \sum_{i=1}^{|S|} ||x_i - m||$$

where m is the nearest medoid to the object x_i.

Just as we can define many different clustering quality measures, we can find many different approaches to clustering tasks. Let us list several main techniques used for clustering:

k-means

The k-means algorithm assumes that the number of clusters is known a priori. This algorithm works with the representation of clusters by mean vectors. An initial clustering is chosen randomly then following two steps are repeated until the clustering solution (i.e., the set of mean vectors of the clusters) remains unchanged. This algorithm is presented by pseudo-code as Algorithm 1.

Probably the main drawback of the k-means algorithm is the necessity to know the proper value of k before start of the algorithm. If we don't know the exact value of k, but at least we some some estimation for it, we can run the algorithm repeatedly and try all possible values for k from the estimated range. Then, the best result is taken as the final solution.

Algorithm 1 k-means clustering algorithm

1: Build the initial clustering solution randomly.
2: **repeat**
3: $\forall x \in S$: Put x into cluster S_j such that $||x - m_j||$ is minimal among all mean vectors from the actual clustering solution.
4: Actualize mean vectors of the clusters.
5: **until** The solution was not changed

Fuzzy k-means
This algorithm is a modification of the previous one to be used for overlapping clustering. The objective function being optimized is following:

$$f(C = \{S_1, ..., S_k\}) = \sum_{j=1}^{k} \sum_{x_i \in S} u_{ij}^m ||x_i - s_j||^2$$

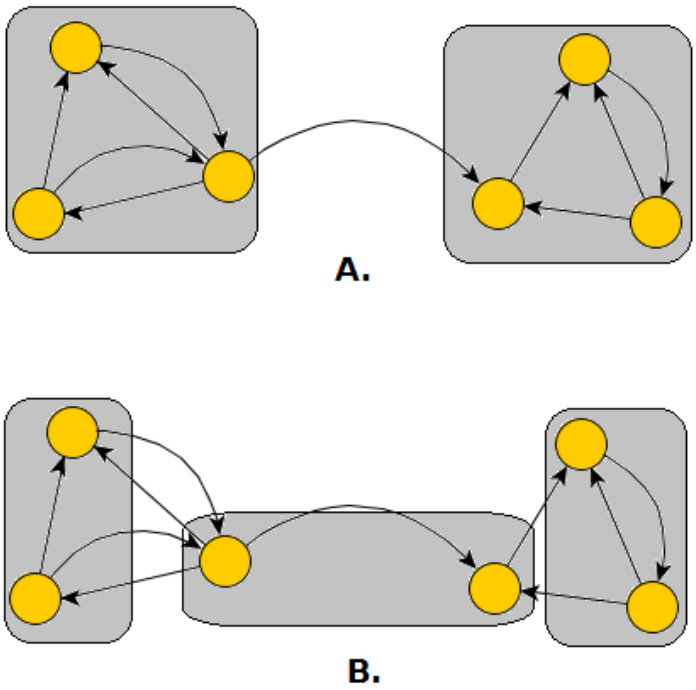
where u_{ij} is a degree of belonging for object x_i to cluster S_j, $m \geq 1$ is a parameter of the algorithm. The values of u_{ij} are assumed to be from the range $[0, 1]$. At each step of the algorithm, the values of u_{ij} and s_j are recalculated to optimize the objective function until enough quality is reached.

Hierarchical clustering
This approach can be used for non-overlapping clustering if no estimation for optimal k is known or if we want to get more clustering solutions with different granularity. The idea is that we start with solution where $k = |S|$, it means that each cluster is composed by only one object. Next solutions are built level by level - clustering at the upper level is obtained from the clustering at the lower level by joining two most similar clusters together. This process can continue until we get a solution consisting of just one cluster (which contains all objects from S). The crucial step to get good clustering solutions is of course to find a good cluster similarity measure which depends on the nature of objects from S. Alternatively, the hierarchy can be built in top-down direction - starting with just one cluster containing all nodes and splitting clusters until $|S|$ one-node clusters are produced. The hierarchical clustering can be visualized by a dendrogram. One example of dendrogram is shown in the figure 2.1.

Self-organized maps
These clustering methods utilize artificial neural networks with unsupervised learning, known as Kohonen maps, and their modifications for cluster detection in the data set. By the adaptation process, neurons of the map tend to represent the data set more preciously. This allows easier identification of clusters that may be not so obviously found directly in the data.

Evolutionary algorithms
Algorithms based on evolutionary approach to searching for the final clustering by evolving population of solutions. The fitness used in the process of evolution can be based on any clustering quality measure, depending on the domain of the clustering problem. The advantage of this approach is that evolutionary algorithms can be adapted to various domains thus they can be used for clustering

Figure 2.2 An example of two clusterings on one graph, (A.) is intuitively better than (B.)



no only in metric spaces. Usage of evolutionary algorithms for clustering will be discussed more deeply further in this work.

2.2 Clustering on graphs

The aim of this work is to analyze evolutionary based algorithms for clustering nodes of directed graphs. For exact formulation of the problem, let us introduce the definitions needed.

Definition. *A **directed weighted graph** is an ordered pair G=(V,E) and a function $w : E \to \boldsymbol{R}$ where V is a set of nodes, $E \subseteq V \times V$ is a set of directed edges and w is a weight function, which assigns each edge a real number representing its weight.*

.

In this work, we will further assume that the range of the weight function w is a subset of interval $[0, 1]$. We will study non-overlapping clusterings of sets of graph nodes where the number of clusters is not known in advance.

2.2.1 Clustering quality measures for graphs

As a measure of similarity we will take the connectedness of nodes by the edges. This means that two nodes that are connected by an edge are more similar to each other than two nodes that are not connected. If we are working with weighted graph, we can also take to account weights of the edges (the higher the weight, the more similar the nodes are). Intuitively, a good cluster of nodes should be

a subset of the set V such that nodes within the cluster are more connected than nodes from different clusters. With this definition of good clustering, one can often for two given clusterings intuitively decide, which one is better and which one is worse, as shown on the figure 2.2. But if we want perform graph clusterings algorithmically by computers, we need more exact definition of a good clustering. Same as before, also for the graph clustering we will need any objective function to be optimized during the process of searching for optimal clustering. A sufficient clustering quality measure for graphs should take to account both intra-cluster density of edges and inter-cluster sparsity of edges. When looking for a proper quality measure, first try can something like a function called *Coverage*, mentioned in [6]:

$$Coverage(C) = \frac{\sum_{e' \in I_C(E)} w(e')}{\sum_{e \in E} w(e)} \tag{2.1}$$

where $I_C(E) \subseteq E$ is a set of graph edges that are intra-cluster edges in clustering C.

However, this is not a good quality measure for optimization, because the maximum is always reached for the clustering composed of just one cluster containing all nodes of the graph, independently on the graph edges. On the other hand, *Coverage* can be used as an additional quality measure when testing any clustering solution or as a part of another function.

Another function, called *Modularization Quality (MQ)*, is proposed in [4]. Here we can modify it for weighted graphs. Let C is a clustering composed of k clusters $V_1, ..., V_k$. Denote $I_i(E)$ a set of intra-cluster edges of cluster V_i, $Ex_{ij}(E)$ a set of inter-cluster edges between clusters V_i and V_j. Define:

$$A_i = \frac{\sum_{e \in I_i(E)} w(e)}{|V_i|^2} \tag{2.2}$$

and

$$B_{ij} = \begin{cases} 0 & i = j \\ \frac{\sum_{e \in Ex_{ij}(E)} w(e)}{2|V_i||V_j|} & i \neq j \end{cases} \tag{2.3}$$

. Finally, we can define $MQ(C)$ as follows:

$$MQ(C) = \begin{cases} A_1 & k = 1 \\ \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} B_{ij}}{\frac{k(k-1)}{2}} & k > 1 \end{cases} \tag{2.4}$$

Modularization Quality rates high intra-cluster densities (expressed by A_i) along with penalization for too many inter-cluster edges. A feasible clustering solution can be reached by maximization of MQ. *Modularization Quality* can be computed for clustering composed of any number of clusters. This allows this function to be used also in algorithms that don't need to know the proper value of k beforehand.

In [2], the authors present an alternative function for measuring clustering quality. They call the function *Performance*. Again, we can modify this function for weighted graphs. The definition is following:

$$Performance(C) = \frac{\sum_{e \in I_C(E)} w(e) + |Ex_C(nonE)|}{\frac{1}{2}|V||V-1|} \tag{2.5}$$

where $nonE = (V \times V) \setminus E$ and $Ex_C(nonE) \subseteq nonE$ is a set of pairs (u, v) such that $u, v \in V$ and u and v belong to different clusters in clustering C. The authors also show another formula for calculating $Performance$:

$$Performance(C) = 1 - \frac{2|E|(1 - 2Coverage(C)) + \sum_{i=1}^{k} |V_i||V_i - 1|}{|V||V - 1|} \tag{2.6}$$

(Note that the $Performance$ is originally designed for indirected graphs, if used for directed weighted graphs, the function values can sometimes be grater than 1. If it could cause any problems, we can slightly modify the formula by omitting $\frac{1}{2}$ in the denominator.)

The last function to be mentioned here is *Conductance*, defined in [6] and [2]. The *Conductance* of a graph is defined as the minimal *Conductance* among all possible cuts in that graph. Let us denote $\alpha = \sum_{e \in Ex_R(E)} w(e)$, $\beta = \sum_{e \in E(R,V)} w(e)$ and $\gamma = \sum_{e \in E(V \setminus R,V)} w(e)$, where $E(X, Y) \subseteq E$ stands for a set of edges that goes between sets $X \subseteq V$ and $Y \subseteq V$. Then the *Conductance* of a cut $R \subseteq V$ is defined:

$$Conductance(R) = \begin{cases} 0 & R \notin \{\emptyset, V\}, \sum_{e \in Ex_R(E)} w(e) = 0 \\ 1 & R = \emptyset \text{ or } R = V \\ \frac{\alpha}{\min(\beta,\gamma)} & \text{otherwise} \end{cases} \tag{2.7}$$

During the process of searching for high quality clustering, we want to obtain a clustering which has high intra-cluster *Conductances* A disadvantage of this function is that finding a proper cut R with the minimal *Conductance* is NP-hard. Thus, *Conductance* isn't directly applicable for repeated evaluation of clusterings. The authors of [2] present that an approximation algorithm for *Conductance* exists with approximation guarantee $O(\sqrt{\log(|V|)})$ when computing *Conductance* for graph $G = (V, E)$.

2.2.2 Non-evolutionary approaches to graph clustering

In this section, we will review few clustering techniques presented in literature, that are not based on evolutionary algorithms. The evolutionary based methods will by analyzed more deeply in next sections.

A general algorithm, which consecutively constructs hierarchy of clusterings, is shown in [6]. The algorithm utilizes greedy approach when choosing clusters for splitting or merging and is presented in the listing as Algorithm 2.

Algorithm 2 General greedy clustering algorithm

1: Let C_0 be an initial clustering solution, i := 0
2: **while** $\{C | C \in N(C_i)$, quality$(C) >$ quality$(C_i)\} \neq \emptyset$ **do**
3: $\quad C_{i+1}$:= solution $C \in N(C_i)$ with the highest quality
4: $\quad$ i := i+1
5: **end while**

There are two ways how the algorithm can build the hierarchy:

- bottom-up, when the initial solution C_0 consists of $|V|$ clusters, each containing just one node, and $N(C_i)$ is a set of solutions obtained from the solution C_i by merging two chosen clusters together.

- top-down, when the initial solution C_0 consists of one cluster, containing all nodes of the graph, and $N(C_i)$ is a set of solutions obtained from the solution C_i by splitting one chosen cluster into two clusters.

The `quality` function represents any clustering quality measure, which is used. When a complete clustering hierarchy is built, we can choose the level with the best quality (or with desired number of clusters).

Another algorithm is based on improving obtained clustering solution by local search using operations allowing swapping nodes between clusters, change cluster of one chosen node and starting new cluster by excluding one node from its original cluster. The algorithm is similar to the previous one, solution C_{i+1} is chosen from the set $N(C_i)$, until enough quality of the clustering is reached. The authors of [6] call this algorithm *Shifting*.

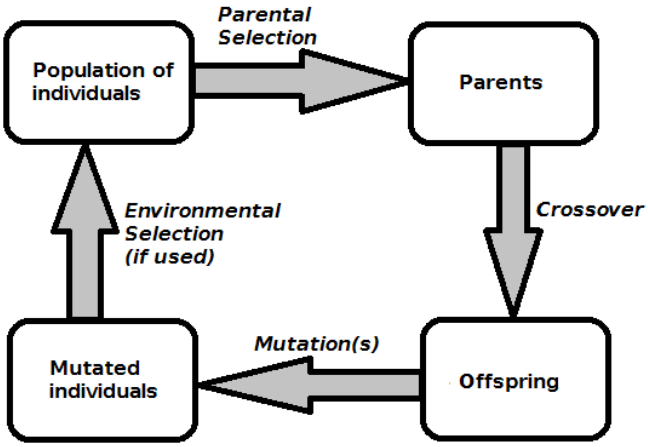
In [2], there is an algorithm based on Markov model called *Markov clustering* published. Main idea behind this algorithm is that a random walk performed on a given graph is more likely to stay inside a cluster before it visits enough nodes instead of leaving the cluster. The algorithm holds a transition matrix $|V| \times |V|$ and iteratively updates it until a recurrent state of some period is reached. Then, a clustering solution is constructed from the final matrix where clusters are induced by connected components of the graph. The connected components are determined by values in the final transition matrix. One iteration of the algorithm computes e-th power of the matrix (simulating e steps of the random walk) and then re-normalizes it.

Another algorithm presented by the same authors is called *Iterative Conductance Cutting*. This algorithm repeatedly splits clusters chosen by a criterion using the *Conductance* function. Because computing real value of *Conductance* is NP-hard, an approximation is used (which is mentioned above). The algorithm is improving the clustering while there exists any cluster with *Conductance* lower than a threshold value α.

2.3 Genetic algorithms

Genetic algorithms, originally proposed by John Holland in [7], are formally a subclass of evolutionary algorithms. They can be viewed as a general search meta heuristic for solving optimization problems. The searching is iterative, population-based. The algorithm holds a population of candidate solutions, encoded into individuals of the population, and evolves them using operations (so called genetic operators) inspired by the real biological evolutionary process. By performing smaller changes in the individuals during evolution, the best solutions should tend to approximate the optimal solution of the problem better and better. The genetic operators that perform the changes correspond to reproduction and mutation processes in the nature, that make creatures more adapted for the environment they are living in. The environment is simulated by a fitness function which rates individuals according to quality of the solution that they

Figure 2.3 Basic cycle of the genetic algorithm



encode. Better solutions are more likely to be selected to the next generation and for applying genetic operators. The algorithm can repeat evolutionary steps until an individual representing solution with desired quality is found or upper bound for number of iterations is reached.

Usually, there are three main types of genetic operators:

- **Selection** - Genetic operators of this kind select individuals that will be involved in creating new population of individuals in one iteration of the genetic algorithm.
- **Crossover** - These operators recombine individuals to produce new individuals and thus new solutions. The offspring individuals are typically constructed by combining some pieces of information taken from the parental individuals. Number of parents and offsprings depends on the concrete crossover operator.
- **Mutation** - Operators of this type are responsible for performing smaller or bigger changes in, typically newly created, individuals. Mutations allow new features of the individuals to emerge and help with preventing population from stagnation in local optimum in the search space.

More about genetic operators will be discussed in further sections.

Evolution cycle of a generic genetic algorithm is illustrated in the figure 2.3. An initial population can be generated randomly or with help of some heuristic. Size of the population can be one of the parameters of the algorithm. Typically, number of individuals in population remains constant during all iterations, but some variants of genetic algorithm can be modified to work with changing population size. Genetic algorithms can be used for solving wide range of problems. The way how to encode a solution into an individual depends on the domain of the problem solved. In original Holland's genetic algorithm, the solutions were always encoded into individuals as bit vectors of some fixed length. On the one hand, this can simplify design of all operators and analysis of the algorithm, on the other hand, it can be impractical for some problems to express their solutions by bit vectors. Such individuals can be too large and evaluation of the fitness

function can be unnecessarily complicated. So, another encodings of solutions are often used. Individuals can be represented as vectors of real numbers or as permutations of integers or even as some more sophisticated structures, if it can be suitable for genetic operators design and for the fitness function, which is used.

2.3.1 More on genetic operators

In this section we will investigate several ways how can be the genetic operators used, independently on the concrete problem. Of course, we can also sometimes use operators that are designed for some specific problem (they use some extra knowledge about the domain of the problem). Some special operators designed for graph clustering will be discussed in sections with concrete algorithms.

Selection
Selection operators are used for selecting individuals that will be used for generating offspring for the next generation. In other words, they select parents of the new population. Sometimes, there can be two selections performed in one iteration - parental selection and environmental selection. The first one selects parents from the old generation that will be used for generating offspring and that will undergo crossovers and mutations and become candidates for the new generation. The environmental selection operator selects individuals that will become new generation. These individuals are selected from a set (or multiset) of parental individuals selected by the parental selection and their offspring. So, if the parent is of higher quality than the offspring (i.e., has higher fitness value), it can be selected for the next generation more likely than the offspring, despite that the parent is from the old generation.

As we have said before, selection operators should select individuals according to the quality of the solution they represent. Quality of the solution is expressed by the fitness function, which rates the individuals. Because the genetic algorithm is in principle an optimization task of the fitness function, it is crucial to use some function, that describes the real problem correctly and preciously.

Individuals can be selected in several ways according to their fitness value. Probably most commonly used approaches are following:

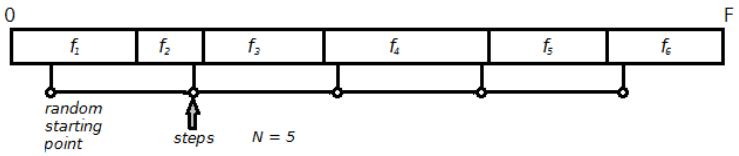
- **Roulette selection** - This type of selections selects individuals proportionally to the fitness function value. For i-th individual in the population, let us denote p_i the probability that the individual i will be selected, f_i fitness value of the individual i and F the sum of fitness values of all individuals from the population. Then, in the roulette selection, p_i is set by the following formula:

  $$p_i = \frac{f_i}{F}$$

  The weak point of this selection can exhibit when there are just few individuals with high fitness value while the rest of population has significantly lower fitness values. In this case, roulette selection can lead to very low diversity in the population and the evolution process can get stuck in a state too far from the optimum.

- **Tournament selection** - This selection operator has two parameters - size of the tournament s and a probability p. When selecting an individual,

Figure 2.4 Principle of the stochastic universal sampling selection



the selection is performed in this way: s individuals are taken randomly from the population and one from these s individuals is selected. The probability, that the i-th best individual from these s individuals wil be selected is $(1-p)^{i-1}p$. This means, that if the best one is not selected (probability p), then we try to select the second best individual etc. until any individual is selected. By changing parameters of the tournament, we can influence the population diversity (smaller tournaments will produce more diversity in the population).

- **Stochastic universal sampling** - It is an another type of fitness-proportional selection, which was introduced by J. Baker in [1]. This selection also operates with values f_i and F as defined above. For describing work of this operator, we can imagine that we put f_i values of all individuals consecutively to create a continuous chain of total length F. Then, we choose a random number from 0 to F as a starting point. If we want to select N individuals, we will perform N steps on the chain, starting from that starting point. The length of one step will be $\frac{F}{N}$ (if we reach the end of the chain, we can continue form the beginning again). In each step, one individual - the one on whose f_i we are actually "staying" - is selected. This process is illustrated in the figure 2.4. The individuals with longer f_i segment (i.e., greater fitness value) has greater chance that we will end some of the steps on their segment and, thus, we will select these individuals.

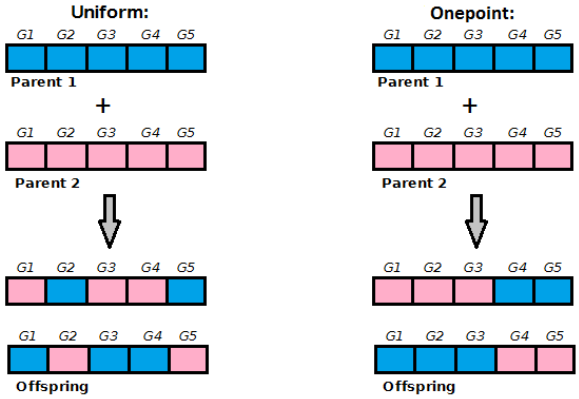
- **Rank based selection** - This approach can be useful, if we want to eliminate problems that can emerge when differences among fitness function values in the population are too high. In rank based selection, all individuals are sorted by their fitness values and the probability that an individual will be selected is adequate to the rank of this individual in the sorted population. The probability of selection depends only on the rank, not on the concrete value of the fitness.

In combination with any of the selection operators mentioned above, another technique called *elitism* can be used. When using *elitism*, few best individuals from the population are automatically transferred to the next generation, independently on the selection operator. Of course, the amount of individuals to be transferred should not be too high but, on the other hand, it can help not to loose good solutions that were discovered earlier. The basic version of elitism is to preserve the best individual for next iteration, which is usually useful because it makes the sequence of best individual's fitness values non-decreasing.

Crossover

The crossover operators are used to construct new individuals from the parental

Figure 2.5 Examples of *Uniform* and *Onepoint* crossover operators.



individuals. Offspring individuals are built by recombining segments of information from the parents. The crossover operator can change the encoded solution very significantly, so usage of it may not be always helpful. Depending on the kind of the problem and on the encoding of solutions into individuals, it can be sometimes hard to design crossover operator that would not destroy the solutions completely. So, in some domains, crossover operators are not used and individuals are modified only by mutations. On the other hand, crossover can bring totally new solutions and protect the searching process from getting stuck in local optimum.

Design of crossover operators is dependent on the type of encoding of individuals and also on the domain of the problem. There are two types of general crossover operators widely used, designed for individuals that are represented as vectors of bits, real or integer numbers or anything else. The elements of the vector are usually referred as *genes* of the individual. The crossover operators patterns are:

- **Multi-point crossover** - For a given number of crossover points q and the total length (i.e., number of genes) of the parents n, q positions between 1 and n are chosen. These are these same in both (or more) parents. This divides parental vectors into $q + 1$ segments of *genes*. The offspring individuals are constructed by exchanging these segments between the parents. Typically, one-point or two-point crossover is used (i.e., $q = 1$ or $q = 2$).

- **Uniform crossover** - In this case, single *genes* are exchanged between parents. For each *gene* of the offspring individual(s), the operator randomly chooses from which of the parents will it be copied.

As we have mentioned above, these two crossover concepts are proposed for individuals represented by one-dimensional vectors, but they can be also extended for usage on more complicated structures if it can make sense. An example of such crossover operator, which is used on more complicated structures, can be the one used in *genetic programming*, where the individuals can be trees representing LISP expressions. The crossover operator then exchanges whole subtrees between the parents.

Very often it can be helpful to use crossover operators that are specialized for the concrete problem and that can work with some additional information about

the solutions. They can compose the offspring solution in a "smarter" way than by random exchange of some *genes*. On the other hand, this can be dangerous because this type of search in the space of solutions can be "too local" and the really optimal solution can be missed. The principles of crossover operators are shown in the figure 2.5.

Mutation
Mutation operators are involved to perform changes by setting new states of some *genes* that can not be done by crossover operators, because they are not contained in any of the parents. This helps with exploring new solutions in the search space of the problem. Again, as in the case of crossover operators, we can introduce some general concepts of mutation operators for vector representations, that can be also adopted for more complex representations.

- **Bit flipping** - If bit-vector representation is used, the mutation can change bit of each *gene* to the opposite value with some probability p (typically, $p < 0.1$ is used) - this means that pn of all *genes* of the mutated individual are changed on the average.

- **Value changes** - If the individuals are represented by vectors of numbers, the mutation can be similar as above, but instead of bit flipping, the value of each mutated *gene* is changed by adding or subtracting a random value from a given range. Alternatively, gene can be mutated by generating completely new value, which replaces the original value.

- **Values swapping** - For some kind of problems, it can be purposeful to swap randomly selected genes in the mutation process. For example, in a traveling salesman problem, where individuals are permutations of nodes, swapping values between two selected genes can be useful.

In mutation operators, there is also a lot of space for involving some refining mechanisms, such as local search procedures. For example, such a refining mutation can perform few refining steps while the fitness function value of the mutated individual is rising. These mutation operators can be quite complicated, so we can call them "smart" mutations, because they do not modify the individuals just randomly, but according to some additional information about good and bad solutions of the problem. The usage of local search procedures must be considered carefully, because they can improve the performance of the genetic algorithm greatly, but they can also prevent it from reaching the global optimum.

Exploration vs. Exploitation
One of the very basic dilemmas (not only) in the genetic algorithm is usage of the proper amount of local search refining operators together more "radical" operators that change the individuals more significantly and usually independently on the original solution encoded in the individual. This is also referred as the problem of exploration versus exploitation.

Searching for the optimal solution (i.e., maximization or minimization of the fitness function) can be viewed as walking on surface modeled by the fitness function, called a fitness landscape. Generally, the fitness landscape is n-dimensional object which may not be continuous. In this landscape, we want to reach the

"highest" point, which is the global maximum of the fitness function (let us assume, that we want to maximize the fitness, minimization can be converted into maximization easily). The individuals of the current population can be represented as points in the fitness landscape (using their fitness function values). In the selection phase, the individuals that are standing "higher" have greater chance to be selected. The mutations and crossovers can be interpreted as shifting the individuals on the fitness landscape. The core of the exploration vs. exploitation problem is how to perform the shifting. Of course, by walking in the fitness landscape, we want to reach the highest point as possible. The exploring approach performs random changes in the individuals, that can look like random "jumping" on the landscape. This process relies on the selection which prefers individuals that "jumped" to higher positions. On the other hand, the exploiting process tries to exploit the starting positions of the individuals as much as possible. It usually tries to shift the individual not very far from the starting point but in such a direction that the fitness function rises. The local search techniques are often referred as hill-climbing (this corresponds with walking only uphill in the fitness landscape). For example, if the derivation(s) of the fitness function can be computed, the exploiting technique can be based on some gradient method of searching.

Both of these approaches have some disadvantages and advantages. While the exploration avoids getting stuck in local maximums, the convergence to the global maximum can be very slow or the maximum can be missed, especially if the set of optimal solutions is very small (the optimal solution does not lie in a plane area). The process of exploitation can prone to get stuck in a local maximum, because in this case, all solutions from the nearest neighborhood of the local optimum does not improve the fitness value, so the local search procedure, e.g. the hill-climbing, can not leave the local maximum. On the other hand, if the fitness landscape isn't too wavy, the exploring strategy can reach the global maximum faster and more probably, because it can better benefit from good solutions found so far - the exploring process can often waste even very promising solutions.

2.3.2 Genetic algorithm performance

Despite that the basic idea behind genetic algorithms is quite simple, it may not be so easy to analyze their behavior more deeply. One example of the genetic algorithm performance analysis is so called Schema theorem, presented by John Holland. It focuses on the original version of the genetic algorithm, where the individuals are represented with bit vectors, one-point crossover and bit flipping mutation operators are used. Before formulating the theorem, we will introduce the definition of *schema*:

Definition. *Let the population in the genetic algorithm contains individuals represented with bit vectors (i.e., vectors composed only of characters* 0 *and* 1*) of length* $\boldsymbol{n}$*. Then the* ***schema*** *is an arbitrary string over the alphabet* $\{0, 1, *\}$ *of length* $\boldsymbol{n}$*. The* ***order*** *of the schema* $\boldsymbol{s}$ *is a number of* 0 *and* 1 *characters in* $\boldsymbol{s}$*. The* ***defining length*** *of the schema* $\boldsymbol{s}$ *is a length of the shortest continuous substring of* $\boldsymbol{s}$*, which contains all* 0 *and* 1 *characters from* $\boldsymbol{s}$*.*

Example: Let's have a schema $**01*10110*1$, then the order of this schema is 8 and defining length is 10.

Now we can use schemas for describing features of the population, instead of working with concrete individuals. Following definitions show the way how schemas can be used for analyzing population evolution.

Definition. *For an individual* $\boldsymbol{I}$*, let* $\boldsymbol{I}[j]$ *denotes a character at the j-th position of the vector, which represents the individual* $\boldsymbol{I}$*. An individual* $\boldsymbol{I}$ *matches with schema* $\boldsymbol{s}$*, if:*

$$\forall j : \boldsymbol{I}[j] = \boldsymbol{s}[j] \quad or \quad \boldsymbol{s}[j] = *$$

.

Definition. *For a schema* $\boldsymbol{s}$*, the fitness function value of* $\boldsymbol{s}$ *is defined as an average value of fitness function values of all individuals (from the current population) matching with the schema* $\boldsymbol{s}$*.*

Finally, we can formulate the Schema theorem:

Theorem. *In populations of the genetic algorithm, schemas with above-average fitness, low order and short defining length increase exponentially during the evolution.*

The proof of this theorem can be found in [7].

Based on the theorem above is the building blocks hypothesis, presented by David E. Goldberg. This hypothesis assumes that the genetic algorithm, according to the Schema theorem, builds the successful solutions from short schemas with low order and above-average fitness, because their amount expands exponentially. These schemas are referred as building blocks. The hypothesis tries to explain how the genetic algorithm can found feasible solutions in huge search spaces quite quickly. From this perspective, the best strings representing the solutions are not built by trying all possible combinations of bits, but they are continuously improved by recombining (i.e.,mutating and crossing over) good schemas. This makes the search process more efficient.

An alternative theory for analyzing genetic algorithm performance is built in [12]. The theory of schemas and building blocks is often criticized because of the lack of preciously proven theorems and exact formulations. Because of this, the author of [12] chooses another approach when the genetic algorithm is viewed as a special case of so called *Random Heuristic Search* algorithm (the class of *RHS* algorithms is also defined). The search process of the algorithm is modeled by states P_0,...,P_n that represent populations and a transition rule(s) τ summarizing how the system can change from a state P_i to state P_{i+1}. Populations (i.e., states of the search process) are multisets of individuals sampled from some search space Ω and represented by vectors of the proportions of each element from Ω in the population. For example, if $\Omega = \{A, B, C, D\}$ and population $P_i = \{A, A, B, D\}$, then the vector representing the population will be $(0.5, 0.25, 0.0, 0.25)$. By running some *RHS* algorithm, a sequence of states $P_0, \tau(P_0)), \tau(\tau(P_0)), ...$ is produced. This sequence of states can be viewed as so called *Markov chain*, because the state $\tau(P_i)$ depends only on the state P_i, not on the previous ones. Transitions between states can be expressed by a matrix Q, where $Q_{p,q}$ means the probability, that $q = \tau(p)$ for some populations (states) p and q. In the genetic algorithm, transition rules between states are given by the basic version of genetic operators.

From this point of view, the performance and convergence of the basic genetic algorithm is deeply analyzed in [12]. This theory should bring some results in areas of asymptotical behavior of the genetic algorithm. The benefits of this approach should be all above in the formal way in which is the theory built and rigorous proofs are constructed (unlike as in the theory of schemas). The drawback is that it can be difficult to apply the theory in practical problems, because the analysis can be intractable even for moderate population and individual sizes.

2.4 Genetic algorithms used for clustering problems

An overview of clustering techniques based on genetic algorithms is presented in [8]. In these reviewed algorithms, each individual represents a possible clustering solution. Fitness functions used mostly utilize some clustering quality measure. Very often, a clustering quality measure can be used directly as a fitness function. The effectiveness of the algorithm depends on encoding used. Thus, the algorithms vary in type of encoding solutions into individuals. The possible encodings mentioned in [8] are:

- **Binary encoding** - this encoding can be used in several ways. The first possibility is that if medoid-based representation of clusters is usable, each individual can be vector of n bits (where n is the total number of objects for clustering) and the bit at the i-th position marks whether the object x_i is a medoid of any cluster. This means that if the individual represents a clustering solution composed of k clusters, the vector contains exactly k bits set to 1 (other bits set to 0). For this encoding, we need $O(n)$ space for each individual. The drawback of this representation can be less effective evaluation of the individuals, because the clustering solutions have to be reconstructed repeatedly by computing the nearest medoids for all objects.

  Another type of binary encoding is based on bit matrices. Each individual is represented by a bit matrix $n \times k$, where n is the number of objects and k is the upper bound for the number of clusters (if no upper bound is known, we can take $k = n$). In this matrix, the bit at the position i, j is set to 1, if the object x_i belongs to the j-th cluster (otherwise, the bit is set to 0). The advantage of this encoding is that it can be used in various domains of objects and, if the number of clusters is low and known a priori, the solution can be reconstructed quite easily from the individual. We can also use this approach for overlapping clustering, because we can express, that an object belongs to more than one cluster. The disadvantage can be the space complexity (and, thus, also the time complexity) when the number of clusters is unlimited. In general case, we need $O(n^2)$ space for one individual.

- **Real encoding** - if the number of clusters is known and the objects exist in some metric space, we can use the encoding in which each individual represents coordinates of all k cluster representatives. These representatives can be, for example, medoids or cluster centers (same as in k-means algorithm). The usability of this approach depends on the number of attributes

of objects, because we need $O(km)$ space for one individual, where m is the number of dimensions.

- **Integer encoding** - The authors of [8] mention two ways how to utilize integer encoding. One option is straightforward - each individual is represented be a vector of $|S| = n$ elements, i-th position of the vector corresponds to the i-th object from S. The integer number j at the i-th position of the vector tells, that in the clustering solution represented by the individual, the object x_i belongs to the cluster number j. The advantage of this encoding is that it can be used for wide range of domains of the objects and that the clustering solutions can be easily obtained from the individuals. The drawback is that the search space can become larger, because one clustering solution can be represented in many ways, for example individuals (1,1,1,2,2,3) and (5,5,5,7,7,8) represent the same clustering solution (of 6 objects into 3 clusters). So, at least the fitness function should be independent on concrete cluster numbers (to evaluate the quality of solutions correctly). Also the genetic operators can be designed to count with this fact. Another possibility is to use some kind of normalization (called a renumbering procedure in [8]), which renumbers the clusters in all individuals such that the individuals representing the same solutions will contain the same values of genes.

  Another way how to encode clustering solutions, when the representation by medoids can be used and the value of k is known beforehand, is to encode a solution into an individual as an array of indices of the cluster representatives. This means that each individual is represented by a vector of k elements and an integer number j at the i-th position means, that the object x_j represents the cluster i (i.e., x_j is a medoid). The disadvantage of this representation is that, in some cases, different individuals can represent same solutions, for example: (1,3,5) and (1,5,3) - both of these individuals represent same clusters (induced by objects x_1, x_3 and x_5). The authors recommend to use the renumbering procedure here, too.

The problem of clustering provides wide range of procedures, that can be implemented as genetic operators, especially as mutations. Multi-point or uniform crossover operators can give sense for all types of encoding mentioned above. Some crossover operators, that take to account the nature of the problem, can be designed. For example, some kind of greedy merging or splitting clusters from the parental solutions could be promising. Many mutation operators can be also proposed for clustering tasks. The general patterns for mutation operators can be adapted for the encodings listed above. This can lead to mutation operators that change cluster assignment for a chosen node, swap nodes between clusters and merge or split selected clusters. Clusters that will be affected by the mutation can be selected randomly (more exploring way) or according to some heuristic or local search technique (more exploiting way). Many "smart" mutations can be also involved to increase performance of the genetic algorithm search. The authors of [8] notice periodical usage of few steps of other clustering algorithms as a refining procedure. If possible, several steps of k-means algorithm can be performed to refine some individuals from the actual population. These refining techniques can be implemented as special variants of mutation operators.

3. Proposed algorithms

As the main goal of this thesis, several evolutionary clustering algorithms for the graph domain (i.e., for clustering nodes of directed graphs), based on the genetic algorithm pattern, were proposed and tested. The algorithms (called genetic clusterizers in this work) differ in genetic operators used and also in the way how the evolutionary search process is guided. Results of tests of the implemented algorithms are published in chapter 5.

3.1 Simple Genetic Clusterizer (SGC)

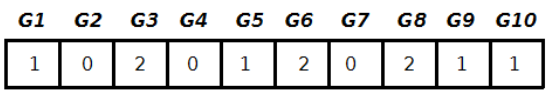
This is the basic version of the genetic algorithm used for clustering graph nodes. The core of this algorithms copies the scheme of a general genetic algorithm, as presented earlier. The evolution runs in cycles, in each cycle several implemented genetic operators are applied. The size of the population is constant during all iterations and can be set at the beginning, same as the total number of iterations. The algorithm runs until the desired number of cycles is reached. At the end of the search process, a solution represented by the best individual (i.e., the individual with the highest fitness) from the population is returned as the result. Now let us summarize main characteristics of the algorithm, such as encoding, fitness function and operators used:

- **Encoding** Because there is not naturally defined metric on set of graph nodes, it is not usable to encode clusters by their representatives (medoids or anything else). Probably the most straightforward way for encoding graph clusterings is to directly assign the number of cluster to each graph node. The clusters are numbered with integer numbers, so the integer encoding is used, as described above. Each individual is represented by a vector of $|V|$ integers, the number at the i-th position identifies the cluster, that the node i belongs to. This encoding is illustrated in the figure 3.1.

- **Fitness function** A clustering quality measure referred as *Performance* (defined in the section 2.2.1, see equations 2.5 and 2.6) was used as a fitness function in this genetic algorithm. Alternatively, another clustering quality measure could be used as a fitness function, for example *Modularization Quality* (see equations 2.2, 2.3 and 2.4), which was used as a fitness function in [4].

- **Selection** The *Tournament* selection is used in this algorithm, with tournament size 8 and $p = 0.75$. This type of selection has been described

Figure 3.1 An example of integer encoding. 10 genes encode a clustering of 10 nodes into 3 clusters (numbered 0, 1 and 2)



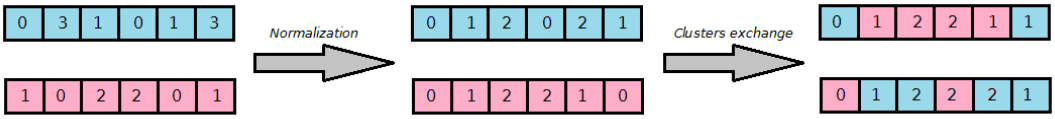
earlier. This selection operator was chosen because it provides possibility of influencing diversity of the population. Generally, small tournament sizes can be used to increase population diversity.

- **Crossover** Several crossover operators were implemented that work with the encoding of solutions into integer vectors. The *Uniform* crossover can be used - in this case this type crossover performs exchanging of cluster assignments in matching pairs of nodes. The *Twopoint* crossover can be also used. In this case, crossover points are set regularly so the blocks of genes being exchanged have almost same length ($\frac{|V|}{3} \pm 2$ because we have two points). Another crossover operator which is more cluster-oriented is called *Clusterwise* crossover. This crossover is similar to the *Uniform* crossover with the difference that the exchange of information between parents is made at the level of clusters, not individual genes. The operator works in the following manner: the clusters in both solutions are numbered such that the cluster containing the node number 1 has number 0, cluster number 1 is the cluster which contains node with the lowest number that does not belong to the cluster 0 etc. By this process, the clusters in both parental solutions are numbered from 0 to $k_1 - 1$ ($k_2 - 1$ respectively), where k_1 and k_2 are the cluster counts from the solutions. Then, clusters with matching numbers are randomly exchanged like gene in the *Uniform* crossover, the unaffected genes by the cluster exchange are copied from parents into offsprings. An example of the *Clusterwise* crossover is presented in the figure 3.2.

- **Mutations** We have implemented both types of mutations: the "blind" ones that modify individuals randomly without any additional information and the "smart" ones, that support exploiting searching by refining the solutions. The uniformed mutations copy the general patterns for mutations in genetic algorithms, as presented above. According to the representation of solutions by integer vectors, random mutations perform swapping cluster assignments of selected nodes or changing cluster numbers of randomly chosen nodes, this can lead to obtaining new clusters or merging existing clusters together. The list of all mutations implemented is following:

  - *Basic mutation*: This is an implementation of the uninformed value changing mutation. The specified number of nodes is randomly chosen and for each of them the assignment of the cluster number is randomly changed (values are uniformly selected from the specified range).
  - *Swapping mutation*: This mutation is also an example of uninformed mutation - it interchanges cluster numbers in a randomly selected pair of nodes. Again, the interchange can be repeated several times.
  - *Splitting mutation*: In this mutation operator, a cluster is randomly selected which is split into two clusters (if the maximal number of clusters is not reached). Nodes from the original cluster are divided into new clusters randomly.
  - *Joining mutation*: This is an opposite operator to the previous one. Instead of splitting cluster, this mutation randomly selects two clusters from the original solution and merges them together to create one cluster.

Figure 3.2 Demonstration of *Clusterwise* crossover (6 nodes, both solutions contains 3 clusters).



– *MinDensity splitting mutation*: This is the first mutation which supports more exploiting solution quality than exploring totally new solutions. It is similar to the *Splitting mutation* above but the cluster to split is not chosen randomly. The cluster selected for splitting is the cluster with the lowest intra-cluster edge density, which is defined as follows (for a cluster S_i):

$$Density(S_i) = \frac{\sum_{e \in I_i(E)} w(e)}{|S_i|(|S_i| - 1)}$$

Unlike in the case of the *Splitting mutation*, nodes from the original cluster are not distributed randomly to new clusters, instead of this, on node is selected from which the process of building the new cluster is started. During the building of the new cluster, nodes from the original cluster are added to the new cluster while density of the new cluster increases. Nodes that didn't increase density of the new cluster remain in the original cluster.

– *MaxCut joining mutation* This is also a more exploiting version of the *Joining mutation* presented earlier. The two clusters selected for joining are chosen such that the total sum of all edge weights between the clusters is maximal among all possible pairs of clusters from the solution.

– *Cluster refining mutation*: This mutation selects one cluster from the solution which will be refined. Refinement of the cluster is performed with usage of the cluster density, as defined above. The cluster for refinement is selected according to the following rule: with probability p, the cluster with the minimal density is selected for refinement, otherwise, with probability $1 - p$, the cluster with the maximal density is selected. The refinement of the cluster works subsequently: For each edge $e = (v, u)$ which crosses the border of the cluster (i.e., one node of the edge lies inside the cluster and one node lies outside), examine all of the three following possibilities and continue with the one with the highest density of the refined cluster: (i) exclude both nodes u and v from the refined cluster, (ii) include both nodes in the refined cluster (iii) leave the original state (e crosses the border). This process is repeated for each border-crossing edge (according to the original state of the cluster before refinement).

To make usage of combination of selected mutations easier, the concept of *Master mutation* was introduced. This mutation does not modify the individuals directly,

but when it is called for mutating the individual, it selects another mutation from a list of available mutation operators and the individual is mutated by this selected mutation instead. Mutations from the list are selected with preset probabilities. The list of mutation operators is created at the beginning of the algorithm. This technique provides possibility to use several mutation operators in the evolution, but in each iteration, only one of them is applied.

A summarization of SGC using pseudo-code can be viewed in the listing of Algorithm 3.

Algorithm 3 Simple genetic clusterizer (SGC)

```
1: Set mutations to be used in the Master mutation
2: while Maximal number of evolution cycles not reached do
3:     Select parents from the current generation, preserve the best individual
4:     Create offsprings by the crossover
5:     Mutate individuals using the Master mutation
6: end while
7: Return the best individual
```

Some combinations of genetic operators and their settings for this algorithm were experimentally evaluated and the results are presented as a part of the chapter 5.

3.2 Two Phase Genetic Clusterizer (TPGC)

This algorithm is based on the previous one, but tries to deal with the problem of exploration versus exploitation. As the name of this clusterizer indicates, the evolution process runs in two phases - the exploring phase and the refining phase. These two phases are regularly alternated during the running of the algorithm. The phases differ in set of genetic operators used and, furthermore, a refining procedure is repeatedly called in the refining phase. Lengths of both phases are set via parameters of the algorithm. Again, the concept of *Master mutation*, as described in the previous algorithm, is used. The phases differ in the lists of mutation operators that are used. In the exploring phase, following mutations are used in the list:

- *Basic mutation* - with increased percentage of genes that will affected by the mutation. Typically, less than 10% of genes are changed by uninformed value-changing mutations, but because we want to support exploration more significantly during this phase, we usually modify more than 10% of genes in this mutation
- *Splitting mutation*
- *Joining mutation*

On the other hand, in the refining phase, mutations in the list are changed to:

- *Cluster Refining mutation*
- *MinDensity Splitting mutation*

- *MaxCut Joining mutation*

Moreover, a special procedure that refines one selected individual is applied in each iteration of the refining phase. The individual for refining is selected randomly from the population. Depending on the algorithm's settings, more (all of them randomly selected) individuals can be refined in each cycle. The refining procedure takes each node of the graph and tries to add it to each of the clusters encoded in the individual which is being refined to improve quality of the solution. At the end of the algorithm, the refining procedure is called for the last time to improve the best solution, just before it is returned as the result.

The last improvement of this algorithm is the feature called *Autostop*. The algorithm can be stopped either by reaching the maximum number of iterations or when the condition for earlier stopping is satisfied. The condition is composed of two parameters: *AutostopTolerance* value and *MaxNotImprovingCycles* value. The condition is satisfied iff the fitness function value of the best individual improves by less then *AutostopTolerance* in each of last *MaxNotImprovingCycles* iterations of the evolution. This can prevent the algorithm from performing useless iterations when the chance for significant improvement of the solution is very low. The *Autostop* feature can be turned on or off by the user.

The search process of TPGC is illustrated by pseudo-code in the listing of Algorithm 4.

Algorithm 4 Two phase genetic clusterizer (TPGC)

```
1: while Maximal number of evolution cycles not reached do
2:   if Refining phase is on then
3:     Select individuals for the next generation
4:     Mutate individuals with refining phase mutations
5:     Refine randomly selected individual(s) by the refining procedure
6:   else
7:     Select parents from the current generation
8:     Create offsprings by the crossover
9:     Mutate individuals with exploring phase mutations
10:  end if
11:  if Autostop condition satisfied then
12:    Escape from the evolution loop
13:  end if
14: end while
15: Refine the best individual by the refining procedure and return it
```

3.3 Iterative Genetic Clusterizer (IGC)

This clusterizer is also based on SGC (or TPGC) but this algorithm uses it repeatedly. If there is no estimation for the optimal number of clusters in the final solution, the search space of all possible solutions can be large. The extent of the search space makes it harder for the SGC algorithm to head towards promising and really good solutions during the search process. IGC tries to make the searching more structured by successive increasing the upper bound for

the number of clusters and repeated executing of SGC or TPGC with this newly set limit. The searching starts with the minimal value for the maximal number of clusters, which is equal to 1 (all nodes belong to just one cluster). Then, this limit is incremented by 1 in each iteration while the clustering solutions obtained by execution of internal SGC or TPGC are improving. Written by pseudo-code, the idea if IGC is listed as Algorithm 5.

Algorithm 5 Iterative genetic clusterizer (IGC)

```
1: Set the maximal limit for clusters max_clusters to 1
2: repeat
3:    Execute SGC with parameter max_clusters
4:    if New best solution was discovered then
5:       Remember the new solution
6:    end if
7:    max_clusters := max_clusters + 1
8: until New best solution was not discovered in last k iterations
```

IGC has some extra parameters for judging quality of solutions and for deciding whether to stop or continue with next execution of SGC or TPGC. At first, quality of the solutions can be measured by more than one function - not used for the fitness function (because SGC or TPGC is used in the core which has its own fitness), but these quality measures are used for deciding whether the solution discovered in the current iteration (with the given limit for number of clusters) should be preserved as the new best solution found so far. The quality measures involved in quality judging can set by the user. When measuring quality of the solution, these functions are voting - if the solution is better than the old best solution according to the given function, this function votes for the solution to be the new best solution. The user can also set how many votes are needed to replace the old best solution by the new solution. We can demonstrate it by example:

Let us assume, that we have the best solution discovered in one of the earlier iterations of IGC. In the i-th iteration, SGC or TPGC is executed (with increased limit for maximal number of clusters) and the returned solution is evaluated. We can assume, that we following functions were set for measuring the quality: $Performance$, MQ and $Coverage$. Moreover, the number of votes needed for replacing the best solution was set to 2. Let the old best solution (which has been discovered in earlier iterations) has $Performance = 0.853$, $MQ = 0.211$ and $Coverage = 0.499$ and the solution discovered in the last iteration has $Performance = 0.778$, $MQ = 0.325$ and $Coverage = 0.651$. In this case, the new solution obtains two votes (from MQ and $Coverage$ functions) for becoming new best solution. Because of the settings of the algorithm, two votes are enough to replace the stored best solution with this new one. Then, IGC continues with increasing the cluster limit and begins new iteration.

Next parameter which can influence behavior of the algorithm expresses the "sensitivity" of the algorithm. This parameter is referred as the value k in the listing. This value determines how many iterations of IGC can be maximally performed in row without discovering new best solution. Because with increasing limit for clusters the search space becomes larger, it can be harder to find better

solution even if there exists some better one which profits from the increased number of clusters. Because of this, it is usually not advisable to demand improvement in each iteration of IGC. On the other hand, a proper limit for number of not improving iterations can prevent the algorithm from useless tries to find solutions with unnecessarily high number of clusters. In other words, by setting proper value of k the algorithm can be stopped just after reaching the optimal number of clusters (which is not assumed to be known before executing IGC).

4. Software

For practical experiments and evaluation of the proposed algorithms, a standalone application was developed as a part of this work. This application, called *Graph Clusterizer*, allows execution of implementations of the clusterizers based on the genetic algorithms described in chapter 3. The application provides basic options for setting parameters of the algorithms, loading graphs for clustering and collecting statistical data for performance evaluation.

For running the application, *Java Runtime Environment (JRE)* of version 1.6.0_21 or higher is required to be installed on the system. *JRE* can be downloaded from the following URL: `http://www.java.com`. Source code of the application (and the application itself) can be downloaded from the SourceForge page of the project: `http://sourceforge.net/projects/gclusterizer/`.

The application is contained on the attached CD.

4.1 Basic architecture

The *Graph Clusterizer* application is implemented in *Java SE*. It is designed as a server application with no graphical user interface. *Graph Clusterizer* runs on the server machine and receives text commands from the clients sent over network using TCP connection. The commands are used for loading graphs for clustering, setting configuration parameters of the algorithms used for clustering and for withdrawing the solutions when the computation is done. The current version of the application does not allow multiple clients to be connected concurrently, but adding this functionality can be one of the aims for further development (which is planned). At the client side, almost any Telnet application for network communication can be used, for example standard Unix or Windows Telnet clients or *The Java Telnet Application* (`http://sourceforge.net/projects/jta/`). It is also possible to create a client program with GUI which converts the actions of the user into text commands and sends them to the server (and, of course, receives the responds and shows them). Complete list of all possible commands is presented further in this chapter.

When a clustering task is started (i.e., one of the clusterizers is called to compute a clustering solution on previously loaded graph), the algorithm is executed in its own thread. This allows the server to accept another commands during the clustering, for example, querying the state of the computation. In the current version of the application, parallel execution of more than one clusterizer (in multiple threads) is not possible, but it should not be very complicated to extend the functionality to allow this. By a request for execution of any clusterizer while another one is running, the old one is discarded and new thread for executing the new clusterizer is created.

The application also takes into account the possibility of graphical presentation of the computed clustering solutions. Because of the distributed and network related nature of the application, *Graph Clusterizer* does not perform the drawing by itself but instead of it, it relies on any other program which is reachable via the network and has ability to show the solution. Format of the data sent to the drawing program depends on that program. In the current version, one drawing

application is supported (for demonstrational purposes) called *AbuGraph*. This application is also developed in *Java* by Ignacio Labrador Pavón and is available for free at `http://abugraph.sourceforge.net/`.

4.2 Usage of the application

4.2.1 Starting the server

The application classes are packed into jar file called `glusterizer.jar`. This archive can be loaded and executed by the *JVM* by issuing following command from the command line in the application's directory:

```
java -jar gclusterizer.jar [-p port_number]
```

where `[-p port_number]` is an optional command line argument which can be used for specifying the port on which the server will start listening for commands. If the port parameter is omitted, default port number is used (which is defined as 3333 in the current version of the application).

After the successful execution, the server begins waiting for connection from a client. For example, standard Telnet application can be used as a client program for establishing connection:

```
o host_name port_number
```

When the connection is established, the server answers with a welcome message and after that, commands can be sent from the client to the server.

4.2.2 Configuring and executing clusterizers

When the server is running and listening for commands, following things can be configured and set:

- **Clusterizer** - The user can set which clustering algorithm will be used for next execution of the clustering task. the server holds this setting until it is changed by the user (or the server is restarted).
- **Graph** - An instance of graph must be loaded before the clustering task can be executed. When the graph is loaded (from file), it is stored on the server until it is replaced by loading a new one. It is not possible to store more than one graph concurrently on the server. The graph remains unchanged among multiple executions of clustering algorithms.
- **Clusterizer configuration properties** - For each clusterizer, several parameters can be set before its execution. All settings for all clusterizers are gathered in one set of configuration properties as pairs `property_name, property_value`. If some property is not explicitly set by the user, the default value is used (when a clusterizer which uses this property is executed). Properties which are set but not used by the executed clusterizer are ignored (but they remain stored in memory of the server for future usage). Configuration properties can be also deleted by the user.

When the proper clusterizer is selected, graph for clustering is loaded and all properties are set to required values, a clustering task can be started (by the `run` command, described further). The clustering task is executed in a separate thread, so the server can be queried for the status of the execution. When the computation process is done, the solution is kept and can be obtained by the user (when a new clustering task is executed and finished, the old solution is rewritten by the new one).

4.2.3 Commands

The current command set of the application contains following commands:

- `setc clusterizer_name` - Sets the clusterizer which will be used for next clustering task. The clusterizer is specified by its name via the parameter `clusterizer_name`. The name of the desired clusterizer must be the same as the name of the Java class containing implementation of that clusterizer. For currently implemented algorithms, the names are: *SimpleGC* for SGC, *TwoPhaseGC* for TPGC and *IterativeGC* for IGC.

- `loadg -f filename` - Loads a graph for clustering from file which is stored locally on the machine where the server application is running. The option `-f` indicates that the graph will be loaded form local file. This option must be always set to this value in the current version of the application, but it is planned to extend graph loading possibilities to allow graph loading from network input stream.

- `setp prop_name prop_value` - Sets the value of the configuration property `prop_name` to `prop_value`. The old value of the property is rewritten (if exists).

- `listp` - Lists all configuration properties with their actual values that were set by the user. Properties not listed will be replaced by their default values when needed.

- `loadp -f filename` - Loads configuration properties from file `filename` which is stored on the server machine. Same as with loading a graph, the option `-f` indicates, that the properties will be loaded from local file. It is planned to extend the functionality to allow loading of properties via network (but it is not implemented in the current version yet). The structure of the file with configuration properties is defined by the rules for Java properties file which is described in the Java language documentation[1].

- `deletep prop_name` - Deletes the configuration property `prop_name` from the list of user set properties (it means that if the value of this property will be needed later, the default value for this setting will be used). If the word `all` is used as the parameter `prop_name`, all properties are deleted.

- `listr` - Prints information which (if any) clusterizer is running.

[1] http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html

- `isr` - Prints information whether any clusterizer is running. If the answer is yes, `1` is printed to the output, otherwise, `0` is printed.

- `gets [-q|-t]` - Retrieves the computed clustering solution which is stored on the server. If no solution is known, the message with information about this state is returned. If the option `-q` is used (i.e., the whole command is `gets -q`), just the quality of the retrieved solution is printed (three numbers representing *Modularization Quality, Performance* and *Coverage* of the solution). If the option `-t` is used, the runtime of the last clusterizer execution is returned (in nanoseconds).

- `draws hostname port_number` - This command initiates drawing of the stored clustering solution by the application *AbuGraph* (mentioned earlier in this chapter) running on the machine `hostname` and listening on port `port_number` (the machine can be the same as the machine on which the clustering application is running - if so, `localhost` can be used as `hostname`). In the current version, the drawing process is designed to visualize only intra-cluster edges to allow easy identification of clusters in the solution. If some new applications for visualizing of the solution will be used, this command should be modified to allow the user to choose which application to use.

- `stat on|off` - Turns collecting of statistical data on or off. If the statistics collecting is on, data characterizing population development are collected and stored during the execution of any clusterizer. The collected data can be then saved in the CSV format. More about statistics will be explained later.

- `savestat -f filename sep` - Saves current collection of statistical data to file `filename` (on the server) in CSV format[2]. A character given in the parameter string `sep` will be used as a separator (typically comma or semicolon is used).

- `quit` - Ends the current session. Loaded graphs, computed solutions and configuration properties persist on the server between connections.

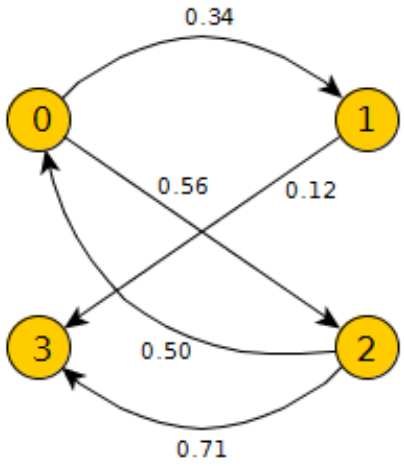
4.2.4 Graphs file format

In the current version of the application, graphs for clustering are loaded from files formatted similarly as in the case of *Trivial Graph Format*[3] files. The difference is that because the nodes are always identified by integer numbers (from 0 to $n-1$ where n is the total number of nodes of the graph), the list of nodes identifiers is omitted (which is a part of *TGF*). Instead of the list, just the total number of nodes is on the first line of the file. Then, a list of graph edges follows, each edge is written on a single line of the file. The structure of this line is following:

```
start_node end_node edge_weight
```

[2] http://en.wikipedia.org/wiki/Comma-separated_values
[3] http://en.wikipedia.org/wiki/Trivial_Graph_Format

Figure 4.1 A sample directed weighted graph.



The edge is interpreted as oriented and weighted, from the node identified by integer number `start_node` to node number `end_node`. The `edge_weight` values is a real number interpreted as the weight of the edge. In practice it can be often useful to have weights normalized within range $[0, 1]$, but it is not required by the loading routine. Records for one edge can be repeated. If so, the last one is used.

An example of file containing representation of a simple graph with four nodes and five edges (depicted in the figure 4.1) can look like this:

```
4
0 1 0.34
0 2 0.56
1 3 0.12
2 0 0.50
2 3 0.71
```

For further development of the application, it is planned to implement loading graphs from other formats, especially *TGF* or some more advanced widely used formats, such as *GraphML*.

A simple utility which is a part of this work can be used for converting graph representations from *TGF* to the format used by the *Graph Clusterizer* application (described above). The utility *TGFConvert* works from command line, takes a *TGF* file and produces a file containing representation of the same graph in the *Graph Clusterizer* format as the output file. The converting application is also developed in *Java*, so the usage is following:

```
java -jar tgfconvert.jar input_file output_file [default_weight]
```

The parameter `input_file` specifies the name of the *TGF* file to be converted, `output_file` specifies the name of the file which will contain the representation of the graph in `Graph Clusterizer` format. The parameter `default_weight` is optional and if it is not omitted, it specifies default edge weight that will be as-

signed to each edge of the graph if it can not be determined from the *TGF* file. The *TGF* file allows to specify a label for each edge (which can be an arbitrary string). Because the *Graph Clusterizer* works always with weighted graphs, the converter tries to interpret the label as a weight of the edge. If it is not possible (the label is not present or is not in a form of a real number), the default weight is used for that edge (specified by the parameter `default_weight`). If this parameter is omitted, `1.0` is used as the default edge weight.

4.3 Clusterizers configuration

Each clusterizer implementation has some parameters for its configuration. These parameters are set via configuration properties before the execution of the clusterizer. If any parameter is set but not used in the currently executed clusterizer, its value is ignored, so it is possible to use one list of configuration properties for all clusterizers, because each algorithm uses only those properties that are needed. On the other hand, some properties can be shared between several clusterizers.

Lists of configuration parameters for each clusterizer are following:

Shared parameters:

- `PopSize` - Size of the population (i.e., number of individuals) of the genetic algorithm used (default value is 150).
- `EvolutionCycles` - Maximal number of iterations of the genetic algorithm used (the exact meaning of this setting can depend on the concrete clusterizer, default value is 500).
- `StatPeriod` - Period (expressed as the number of iterations of the genetic algorithm) between two consecutive samples of statistical data (default value is 10).
- `MaxClusterCount` - Maximal number of clusters in the computed clustering solution (can be used to limit the search space if it is known a priori, default value is 2). If this value is higher than the number of nodes of the graph for clustering, the limit is automatically changed to the number of nodes.
- `SimpleMutProb` - Probability of the *Simple* mutation in the *Master* mutation.
- `SplittingMutProb` - Probability of the *Splitting* mutation in the *Master* mutation.
- `JoiningMutProb` - Probability of the *Joining* mutation in the *Master* mutation.

SGC parameters: This clusterizer has no extra parameters, only the shared ones listed above.

TPGC parameters:

- `Autostop` - Can be `true` or `false`. If `true`, the *Autostop* function is turned on which means that the genetic algorithm can be stopped before the maximal number of iterations is reached (more about this feature can be found in the section 3.2).
- `AutostopTolerance` - Tolerance of the *Autostop* feature (default value is 0.01).
- `MaxNotImprovingCycles` - Also a parameter of the *Autostop* feature, indicates how many iterations can be performed in a row without significant improvement of the best individual (see section 3.2 for details, default value is 3).
- `ExploringCycles` - Length of the exploring phase of the genetic algorithm (number of iterations, default value is 10)
- `RefiningCycles` - Length of the refining phase of the genetic algorithm (number of iterations, default value is 20)
- `HowManyToRefine` - Number of individuals that will be refined by the refining procedure in refining phases (see section 3.2 for details, default value is 3).
- `ClusterRefiningMutProb` - Probability of the *Cluster refining* mutation in the *Master* mutation (in the refining phase).
- `MinDensitySplittingMutProb` - Probability of the *MinDensity splitting* mutation in the *Master* mutation (in the refining phase).
- `MaxCutJoiningMutProb` - Probability of the *MaxCut joining* mutation in the *Master* mutation (in the refining phase).

Probabilities of the mutations used in the exploring phase are set in the same way as in SGC. The sets of mutation operators used in SGC and TPGC were chosen according to the results of experiments (see chapter 5). Another sets of mutations can be set by modifying the source code of the application and rebuilding the project.

IGC parameters:

- `QualityJudges` - List of names of functions that will be used for judging quality of the solution found. The solution will be evaluated by each listed function and depending on values of other parameters, the solution will (or not) be proclaimed as improving. The list of functions must be in form "`function1,function2,...`". Names of the quality measuring functions must be the same as the names of the classes that implement them. So, in the current version of the applications, following functions are available: `ModularizationQuality`, `Performance` and `Coverage`.

- `MaxNotImprovingLevels` - Maximal number of levels (i.e.,repeated executions of the genetic algorithm with increasing limit for number of clusters in solution) when no improving solution is found that have to be tried before stopping the algorithm (default value is 3).

- `QualityDominatingNumber` - This parameter specifies how many votes from the quality judges used are needed to proclaim the new solution best solution found so far (see the detailed characteristics of IGC in section 3.3, default value is 1).

4.3.1 Example of usage

Here we present a scenario illustrating typical usage of the application. Conversation between a client and the server is demonstrated (lines beginning with `>` are input typed by the user while the other lines are ansewers received from the server). The user connects to the server and wants to compute a clustering of a graph stored in the file `graphfile.txt` on the server. The user decides to use TPGC for this job:

```
Hello, this is clustering server!  Type your commands...
> loadg -f graphfile.txt
Graph loaded from file (25 nodes, 130 edges)
> setc TwoPhaseGC
> loadp -f propfile.properties
> setp EvolutionCycles 120
> listp
PopSize=60
EvolutionCycles=120
ExploringCycles=10
RefiningCycles=2
HowManyToRefine=2
> run
> listr
Clusterizer "TwoPhaseGC" is running
> listr
No clusterizer is running
> gets
Number of clusters:  3
Cluster 1 (7 nodes):  1 4 6 7 8 9 12
Cluster 2 (10 nodes):  2 3 5 13 14 15 18 19 20 24
Cluster 3 (8 nodes):  10 11 16 17 21 22 23 25
> gets -q
0.6460578694 0.983657374 0.653786
> draws localhost 3456
> quit
Good bye!
```

After the connection was established, the server sent a welcome message to the user and then, the user started to configure the application. First, the graph

for clustering and the set of properties are loaded from files (`loadg` and `loadp`) and the desired clusterizer is selected (`setc`). Property for setting the number of iterations of the genetic algorithm was modified with the command `setp`. Then, a complete list of configuration properties and their values that were set by the user was listed (`listp`). After that, the clusterizer was executed (`run`). The clustering process took some time, so the user tested whether it is completed (`listr`). When the clustering was done, the user obtained the solution and its quality (`gets` and `gets -q`). Finally the user issued the command for drawing the clustering solution by the *AbuGraph* application running on the same machine.

4.3.2 Automated executing of clustering tasks

If a graph for clustering is too large or if it is needed to run the computation multiple times, it can be time consuming for a human user to do it. To make this process easier, a utility called *GCScriptWorker* has been developed which works as an automatic client program communicating with the *Graph clusterizer* server. The utility also simplifies repeated executions of a clusterizer on one graph with the same settings to obtain average results. When the whole bundle of executions is done, the average values of quality measures are outputted. The utility is configured by the command line arguments when it is started:

```
java -jar GCScriptWorker.jar addr port clusterizer [repeats properties graph]+
```

where `addr` is an address of the machine where *Graph clusterizer* is running, `port` is the port number on which the clustering server is listening (default is 3333), `clusterizer` is the name of the clusterizer that will be used (e.g., `TwoPhaseGC` for TPGC). `[repeats properties graph]` is a configuration of one bundle of executions - `repeats` is the number of reruns, `properties` is the path to the file with configuration properties and `graph` is the path to graph file containing the graph for clustering. The number of bundles is not limited (i.e., the triplet `[repeats properties graph]` can be repeated multiple times to specify more bundles of executions that will be run). An example of running *GCScriptWorker* can be following:

```
java -jar GCScriptWorker.jar localhost 3333 TwoPhaseGC 10 props1.txt g1.txt 5 props2.txt g2.txt
```

This configuration will execute TPGC ten times on graph loaded from file `g1.txt` with properties from file `props1.txt` (first bundle of executions) and then, TPGC will be executed five times on the graph from file `g2.txt` with properties from file `props2.txt` (second bundle of executions). After ending each bundle of executions, average values of quality measures of the returned solutions will be printed.

4.4 Statistics

All implemented clusterizers allow collecting of statistical data reflecting the evolution of the population. Following values are periodically (the length of the period in evolution cycles can be set by the user) saved:

- **Minimal value** of the fitness function.
- **Maximal value** of the fitness function (i.e., fitness function value of the currently best individual).
- **Mean value** of the fitness function among the population.
- **Variance** of the fitness function.

The statistical data are kept in a form of table which can be exported and saved in a file in *comma separated values* format. Each row of the outputted file represents a quaternary of the data listed above. Separator of the columns in the file can be specified by the user (typically, `,` or `;` is used). The collected data can be then used, for example, for producing graphs that can help with analyzing algorithms convergence and for tuning their parameters.

Collecting statistical data can be switched on or off. The saved data are stored on the server until new computation of any clusterizer is executed. When a clusterizer is started, the old data are dropped.

4.5 Implementation

The application is implemented in *Java* programming language, version 1.6.0_21. The source code is organized under the project for *NetBeans IDE*[4] by the *Apache Maven*[5] tool. The design of the application is very modular to allow easy extensions and adding implementations of new algorithms. The project is composed of following packages:

- `cz.cuni.mff.kohoj7am.graphclusterizer` - Main package containing the class with the program entry point (the `main` method) and class `ArgsManager` used for managing command line arguments.
- `cz.cuni.mff.kohoj7am.graphclusterizer.exceptions` - Classes implementing special exceptions that can be thrown in the application should be contained in this package.
- `cz.cuni.mff.kohoj7am.graphclusterizer.functions` - This package aggregates classes implementing functions that can be used as clustering quality measures (and, thus, as fitness functions in clusterizers, for example).
- `cz.cuni.mff.kohoj7am.graphclusterizer.ga` - Contains interfaces for any genetic clusterizer and solution decoder (solution decoder decodes clustering solutions from chromosomes).
- `cz.cuni.mff.kohoj7am.graphclusterizer.ga.impl` - This package contains implementations of genetic clusterizers - cores of the clusterizers, genetic operators used etc.
- `cz.cuni.mff.kohoj7am.graphclusterizer.graph` - Gathers classes and interfaces for representing graphs in the application.

[4] http://netbeans.org/
[5] http://maven.apache.org/

- `cz.cuni.mff.kohoj7am.graphclusterizer.graph.impl` - Contains implementations of concrete types of graphs.
- `cz.cuni.mff.kohoj7am.graphclusterizer.gui` - In this package there should be gathered all classes and interfaces that have anything to do with graphical output and graphical user interface of the application. Also implementations of solution drawing mechanisms should be placed here.
- `cz.cuni.mff.kohoj7am.graphclusterizer.launcher` - Implemented clusterizers can be executed in separate threads, the execution of any clusterizer is maintained by launcher - a class extending the `Thread` class. Launchers can implement some extended functionality like measuring runtime etc. All launchers should be aggregated in this package.
- `cz.cuni.mff.kohoj7am.graphclusterizer.server` - This package contains everything that has something to do with the server interface of the application. Typically, classes involved in serving the network commands will be put in this package.
- `cz.cuni.mff.kohoj7am.graphclusterizer.statistics` - Package for classes used for collecting, saving and exporting statistical data during running of clusterizers.
- `cz.cuni.mff.kohoj7am.graphclusterizer.util` - Contains utility classes serving various purposes, such as generating random graphs etc.

The application is based on *JGAP* library [5] which provides implementations of basic routines for genetic algorithms.

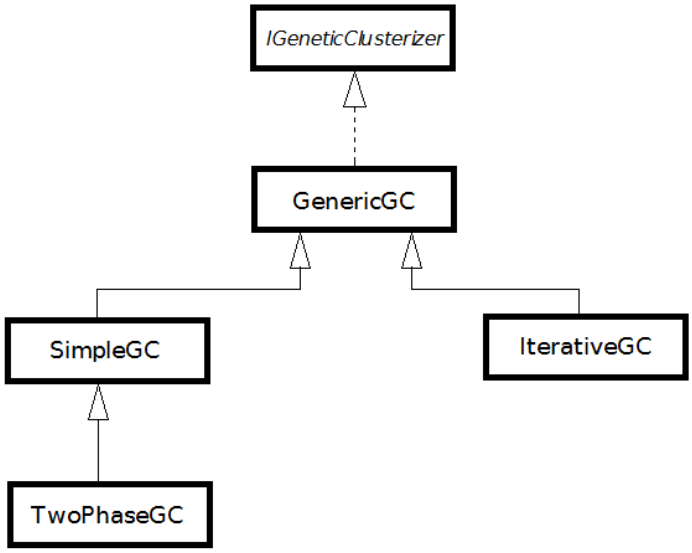
More detailed programmer's documentation can be found in HTML form (generated by the *Javadoc* tool) on the CD attached to this work as well as in SVN repository of the project[6]. The documentation contains descriptions of all classes and interfaces used in the project.

4.5.1 Clusterizers design

The core of the application was designed with emphasis on modularity and easy possibility of adding new implementations of clustering algorithms. Each clustering algorithm (called a *genetic clusterizer*) is implemented in its own class. The algorithm's class must implement `IGeneticClusterizer` interface which contains signature of the method `clusterizeGraph(Graph)`. The method takes a graph for clustering and encapsulates the computation process of the clusterizer. The interface contains also signatures of other methods to ensure compatibility with the rest of the application (such as getting description of the algorithm, setting configuration properties etc.). There are many routines that are common for almost each genetic clusterizer, for example, gathering data for statistics. To make code for these routines reusable, an abstract class `GenericGC` is provided. This class contains implementations of methods that are expected to be the same in most clusterizers, but leaves the method `clusterizeGraph` not implemented. The easy way how to start implementing new clusterizer can be extending the

[6] https://gclusterizer.svn.sourceforge.net/svnroot/gclusterizer/

Figure 4.2 Hierarchy of the implemented clusterizers.



class `GenericGC` (and so it is in the case of clusterizers implemented in this work). The hierarchy of implemented clusterizers is shown in the figure 4.2.

Genetic operators are also implemented in separate classes, according to the architecture of the *JGAP* library (details can be found in its documentation [5]).

As mentioned earlier, clusterizers are configured via configuration properties. In implementation, properties are managed by the `Properties` class which is a part of *J2SE*. The current instance of properties for a clusterizer is set by the method `setConfProperties(Properties)`. One instance of properties can be shared by more clusterizers. The typical course is following: The server creates an instance of the `Properties` and edits it as the commands for setting properties' values are coming from the user. Then, when the clustering is started, the server set the instance with properties to be used by the executed clusterizer. For each property its default value should have been defined. The preferred way how to specify default values of properties is using `static final` variables in clusterizers' classes. The naming convention for an identifier of such a variable is `PROP_DEFAULT_XXX` where `XXX` stands for the name of the property (examples can be found in the source code).

4.5.2 Solutions representation

Representation of computed clustering solutions should be independent on any concrete clusterizer. The interface `IClusteringSolution` is designed for this purpose. Any class which is used for representing a solution should implement this interface. One simple implementation is provided by the `ClusteringSolution` class. Also the implementations of clustering quality measures should be independent on individuals representation (because the quality measures can be used in more situations than only in fitness functions). To enable conversions between solution encodings used in the genetic algorithm and their stand-alone representations, the concept of *solution decoders* is introduced. A *solution decoder* is any class implementing the `ISolutionDecoder` interface, suited to the concrete genetic algorithm and the encoding of individuals used. By calling the method

`decode(IChromosome)`, the solution encoded into the given individual (chromosome in the terminology of the *JGAP* library) is decoded and returned. For each clusterizer that uses its own encoding of individuals there should be a proper solution decoder provided to enable obtaining of clustering solutions representations that are independent on the clusterizer. In the current version, one decoder is implemented which decodes solutions from vectors of integer numbers.

4.5.3 Graphs representation

The modular design of the application applies also to storing graphs for clustering. Each graph that can passed to a clusterizer for clustering must extend the abstract class `Graph`. The way how the information about the graph are stored depends on implementation of the class is used. In the current version of the application (which is attached to this work) the representation by a weight matrix is implemented (see the class `DirectedMatrixGraph`).

Graphs are loaded from text files (structure of these files is described in section 4.2.4), but the functionality can be easily extended to allow loading also from network streams thanks to the design of data streams in *Java*. This loading feature could be useful in the future. The way how the loaded data are parsed depends on the type of representation of the graph, so it is not implemented in the common ancestor class `Graph`.

5. Experiments

In this chapter we present several experiments performed with the implemented clusterizers to assess their performance and estimate proper values for their settings.

All experiments were run on a personal computer with CPU Intel Core 2 Duo, 2.4 GHz, 2 GB DDR3 RAM and OS Microsoft Windows 7 Professional (32-bit). The version of *JVM* was 1.6.0_21. For these experimental purposes, six random weighted oriented graphs $G_1, ..., G_6$ were generated, each with 50 nodes but with different number of edges. The weights of all edges were randomly chosen from range [0,1]. The number of edges in the graphs were following:

Graph	**Nr. of edges**
G_1	293
G_2	225
G_3	316
G_4	172
G_5	331
G_6	251

All graphs used in experiments are stored on the CD attached to this thesis (in *Graph clusterizer* proprietary format).

5.1 SGC experiments

This section covers experiments made with the basic version of the genetic clusterizer - Simple Genetic Clusterizer (described in section 3.1).

5.1.1 Crossover operators test

In this experiment, we have evaluated quality of three different crossover operators: *Clusterwise* crossover, *Uniform* crossover and *Twopoint* crossover (see sections 2.3 and 3.1 for details) that can be used in SGC. For better assessment of the gain of the crossover operators, also results from an execution when there was no crossover operator used are published. The rest of the algorithm's settings were equal in all three cases. Other operators and settings used were:

- **Population size:** 50
- **Evolution cycles:** 100
- **Max. number of clusters in solution:** 50
- **Fitness:** *Performance*
- **Mutations (and their probabilities in *Master* mutation):** *Basic* $p = 0.2$, *Joining* $p = 0.4$, *Splitting* $p = 0.4$

- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation

Note that the *Basic* mutation was set to modify 5% of genes on average. With these settings, the clusterizer was executed ten times on each graph $G_1, ..., G_6$ for each of the tested crossover operators. In the tables below, there are average values computed from these ten executions on each graph. The results of this experiment are summarized in tables 5.1, 5.2, 5.3 and 5.4.

From the results we can see that there is definitely a nontrivial benefit from usage of *Clusterwise* and *Uniform* crossover operators. On the other hand, there is some time tradeoff when these operators are used. *Twopoint* crossover operator does not look like very promising in this case, because it does not speed up convergence of the algorithm so significantly as the previous two operators (while consuming almost the same amount of time as the other crossovers). If the runtime is really crucial, one can consider to use none crossover operator, because it can save quite a lot time (about 50% in case of the experimental graphs used) but lower quality of produced solutions should be probably expected. If the clustered graphs are of some reasonable size (and, thus, the runtime does not play critical role), the best choice is probably to use *Clusterwise* or *Uniform* crossover. On the experimental graphs, we can not uniquely determine which one is definitely better. So, one approach how to deal with it can be swapping between these two operators during the process of evolution.

5.1.2 Mutation operators test

The second experiment is similar to the first one with the difference that several combinations of mutation operators were evaluated. The mutation operators can be combined via encapsulation by the *Master* mutation (as described in the section 3.1), so it is not practically possible to test all possible combinations of them. Because of this huge quantity of variants, we have chosen just five combinations that look like as promising for effective searching for good clustering solutions and these five sets of mutation operators were experimentally evaluated. The experiment was performed on the same set of six graphs $G_1, ..., G_6$ as the first one. For each configuration of the mutation operators tested, the genetic clusterizer was executed ten times on each of the graphs. By this methodology, average values computed from these ten repetitions were obtained and summarized in tables 5.5, 5.6, 5.7, 5.8, 5.9. The evaluated combinations of mutations and their probabilities (stated in parenthesis) were:

1. *Basic* (0.2) + *Swapping* (0.2) + *Joining* (0.3) + *Splitting* (0.3)
2. *Basic* (0.5) + *Swapping* (0.5)
3. *Joining* (0.5) + *Splitting* (0.5)
4. *Basic* (0.2) + *Joining* (0.4) + *Splitting* (0.4)
5. *Swapping* (0.2) + *Joining* (0.4) + *Splitting* (0.4)

The rest of SGC configuration was set in the following way:

Graph	MQ	Performance	Coverage	Time (s)
G_1	0.198	0.912	0.655	1.041
G_2	0.208	0.940	0.640	1.028
G_3	0.184	0.901	0.455	1.049
G_4	0.159	0.948	0.448	1.042
G_5	0.217	0.882	0.310	1.062
G_6	0.164	0.912	0.319	1.056

Table 5.1: *Clusterwise* crossover results

Graph	MQ	Performance	Coverage	Time (s)
G1	0.221	0.913	0.612	1.006
G2	0.188	0.941	0.663	0.993
G3	0.171	0.899	0.443	1.015
G4	0.188	0.951	0.471	0.970
G5	0.223	0.883	0.316	0.994
G6	0.163	0.910	0.306	0.993

Table 5.2: *Uniform* crossover results

Graph	MQ	Performance	Coverage	Time (s)
G1	0.169	0.909	0.602	0.997
G2	0.151	0.938	0.639	0.960
G3	0.142	0.897	0.433	0.994
G4	0.151	0.948	0.435	0.978
G5	0.199	0.880	0.298	0.993
G6	0.132	0.908	0.292	0.985

Table 5.3: *Twopoint* crossover results

Graph	MQ	Performance	Coverage	Time (s)
G1	0.099	0.900	0.483	0.411
G2	0.089	0.928	0.486	0.397
G3	0.076	0.892	0.392	0.426
G4	0.126	0.942	0.340	0.426
G5	0.114	0.872	0.237	0.420
G6	0.108	0.906	0.264	0.406

Table 5.4: Results when no crossover operator was used.

- **Population size:** 50
- **Evolution cycles:** 100
- **Max. number of clusters in solution:** 50
- **Fitness:** *Performance*
- **Crossover:** *Clusterwise*
- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation

Again, the *Basic* mutation was configured to change about 5% of genes (if used) and the *Swapping* mutation was set to perform from 1 to 5 exchanges (the concrete number of exchanges was chosen randomly in each execution of the mutation).

From the results obtained we can see that differences among all evaluated combinations are not radical, but we can still observe that some variants are more likely to find better solutions. For example, the pair consisting of *Joining* and *Splitting* mutations looks useful, because the combination number 2 (which did not use this pair of mutations) reached (besides few exceptions) results of lower quality than the other combinations. Also the versions where the *Basic* mutation is involved tend to be more successful than in the case when the *Swapping* mutation is used too often. Probably, the *Swapping* mutation does not support extensive search for new solutions very much so the convergence of the algorithm during the early phase is slower. So, this mutation is probably more suitable for tuning solutions of some higher quality instead of rough searching from scratch.

5.1.3 Population size and number of evolution cycles test

The goal of this experiment was to estimate proper values for setting number of iterations and size of the population for SGC that are sufficient to find solutions with enough quality while preventing the algorithm from running too long without any gain. It is a very complex task to make an estimation of the proper population size and number of iterations that would suit well for any graph. In this experiment, we focused our attention only to the six experimental representatives of graphs with 50 nodes $(G_1, ..., G_6)$, so the estimations are made according to these graphs. Our aim was to estimate both parameters - number of individuals in the population and number of evolution cycles - so the method of this test was following: for each pair of values of population size and number of evolution cycles, SGC with these settings was executed ten times on each of the graphs $G_1, ..., G_6$. The average values from the solutions obtained were computed and stored for further evaluation. The range of values for the size of the population was from 5 to 70, step by 5 and the range of values tried for the number of evolution cycles was from 10 to 160 (step by 10), so there were totally 224 pairs of values tested on each graph. The rest of algorithm's parameters were set similarly as in the cases of the preceding experiments:

- **Max. number of clusters in solution:** 50

Graph	MQ	Performance	Coverage	Time (s)
G_1	0.166	0.911	0.652	1.099
G_2	0.190	0.941	0.658	1.062
G_3	0.181	0.899	0.444	1.054
G_4	0.175	0.949	0.446	1.077
G_5	0.218	0.879	0.308	1.123
G_6	0.159	0.911	0.308	1.066

Table 5.5: Results of mutations combination nr. 1

Graph	MQ	Performance	Coverage	Time (s)
G_1	0.188	0.887	0.311	1.100
G_2	0.152	0.918	0.358	1.047
G_3	0.149	0.881	0.290	1.064
G_4	0.159	0.943	0.321	1.055
G_5	0.175	0.871	0.204	1.071
G_6	0.158	0.905	0.232	1.051

Table 5.6: Results of mutations combination nr. 2

Graph	MQ	Performance	Coverage	Time (s)
G_1	0.225	0.912	0.629	1.056
G_2	0.196	0.941	0.680	1.031
G_3	0.184	0.898	0.449	1.047
G_4	0.191	0.952	0.488	1.036
G_5	0.220	0.883	0.335	1.054
G_6	0.168	0.912	0.322	1.046

Table 5.7: Results of mutations combination nr. 3

Graph	MQ	Performance	Coverage	Time (s)
G_1	0.199	0.913	0.617	1.066
G_2	0.196	0.939	0.659	1.041
G_3	0.190	0.899	0.432	1.066
G_4	0.202	0.952	0.480	1.041
G_5	0.226	0.883	0.309	1.063
G_6	0.176	0.913	0.334	1.056

Table 5.8: Results of mutations combination nr. 4

Graph	MQ	Performance	Coverage	Time (s)
G_1	0.216	0.911	0.629	1.046
G_2	0.217	0.941	0.672	1.038
G_3	0.180	0.898	0.433	1.056
G_4	0.183	0.950	0.463	1.046
G_5	0.225	0.882	0.323	1.060
G_6	0.174	0.912	0.315	1.057

Table 5.9: Results of mutations combination nr. 5

- **Fitness:** *Performance*
- **Crossover:** *Clusterwise*
- **Mutations (and their probabilities in *Master* mutation):** *Basic* $p = 0.2$, *Joining* $p = 0.4$, *Splitting* $p = 0.4$
- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation

The discovered solutions were evaluated by clustering quality measures defined in the section 2.2.1. The measures used were *Modularization Quality (MQ)*, *Performance* and *Coverage*. The results are shown in the figures 5.1 and 5.2 (the graphs for the *Coverage* function have similar shape to the other two measures, so they are omitted here). The graphs for *Performance* are much smoother than for *Modularization Quality*, because the latter one was not used as the fitness function for the genetic algorithm, so it was not directly optimized by the search process. On the other hand, in both sets of graphs we can see the tendency to stabilize values when the population size reaches values about 50 individuals and the number of cycles is around 100. A positive finding can be that *MQ* of the solutions rises and stabilizes in a similar way as *Performance* thought it is not used in the fitness function.

5.2 TPGC experiments

This section is focused on presenting results of experiments performed with the second implemented genetic clusterizer, which is TPGC. The designed experiments should provide some information about the behavior of the algorithm when the swapping between two different searching approaches is involved (as described in the section 4). Next aim of these experiments is to enable comparison of SGC and TPGC algorithms.

5.2.1 Phases lengths test

This experiment was testing the influence of lengths of exploring and refining phases on the quality of the final solution. The basic question is whether it is better to swap between the two phases quickly (i.e., to set short periods of the phases) or conversely let each phase to run longer. Similarly as in the population size and evolution cycles test for SGC, several pairs of settings for exploring phase length and refining phase length were tested on the graphs $G_1, ..., G_6$ by ten executions on each of them. The range of values tested for the exploring phase length was from 5 to 17, stepping by 4 while the set of values for refining phase length was $1, 2, 3, 4, 5$ (it means that 20 pairs were tested totally). TPGC can use the same genetic operators as SGC and so it is in this case, except the mutation operators. The configuration of the algorithm for this test was following:

- **Population size:** 50
- **Evolution cycles:** 50

Figure 5.1 *Modularization Quality* of solutions obtained by SGC depending on the preset values of population size and number of evolution cycles.

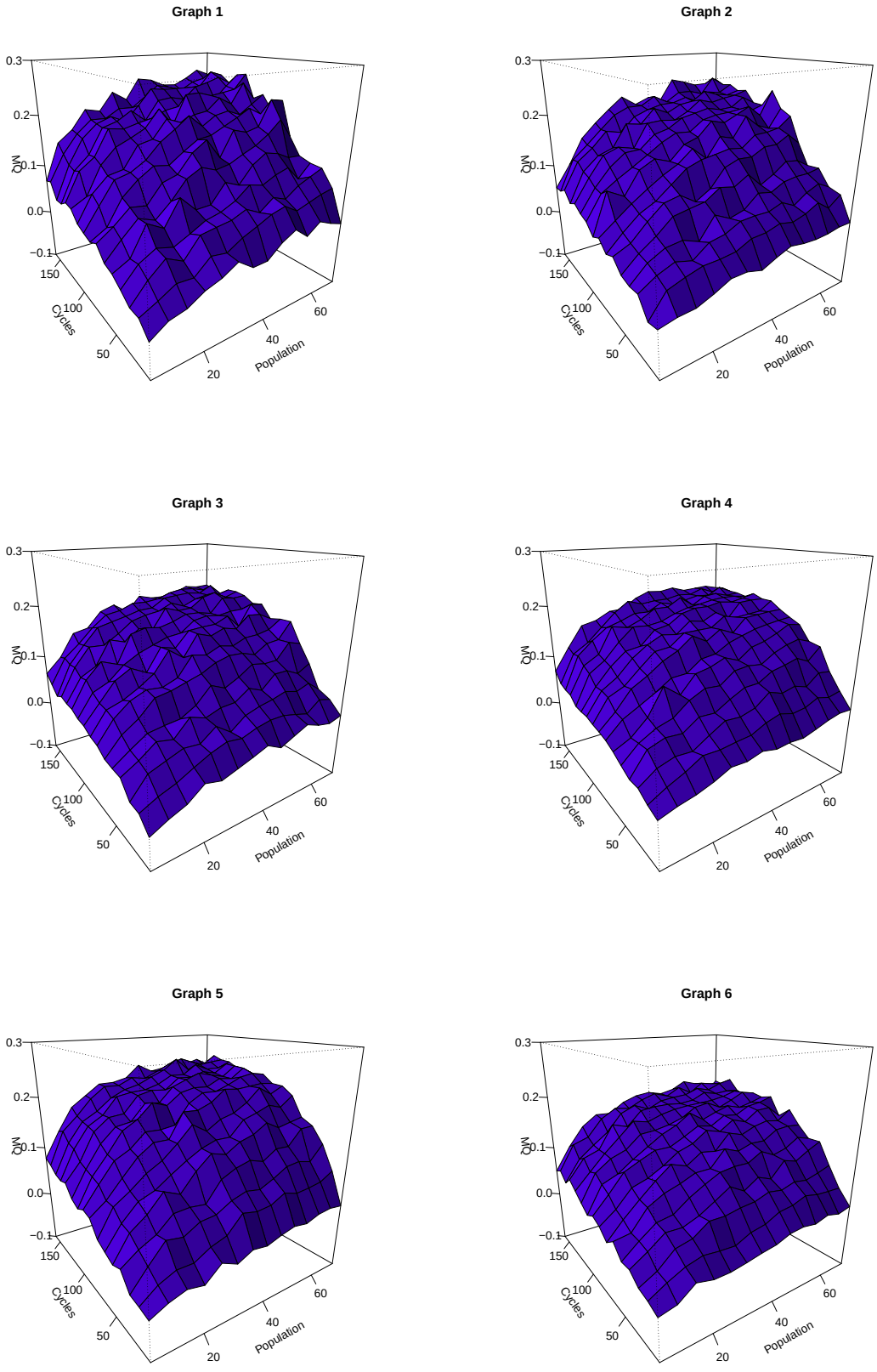
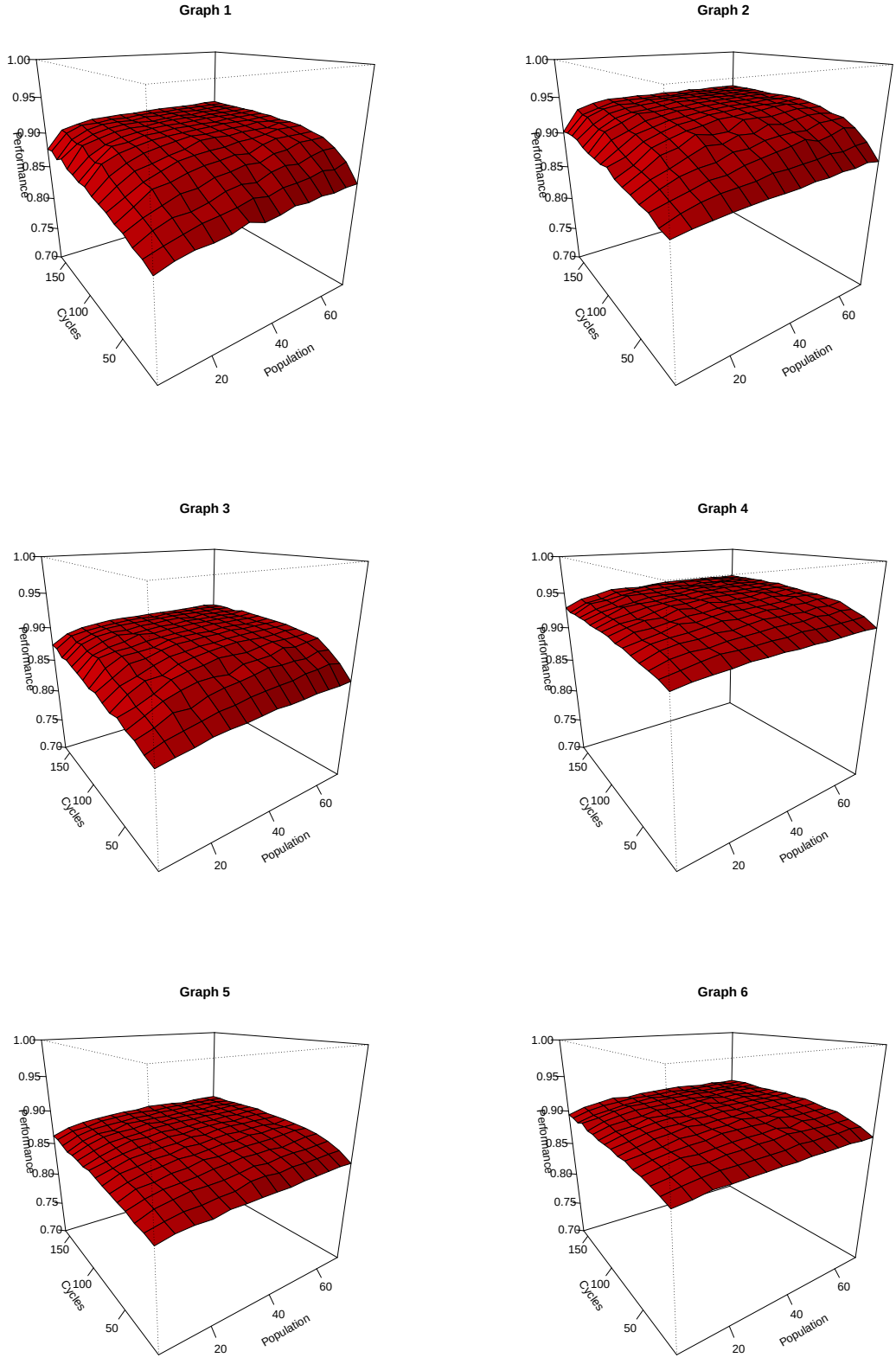


Figure 5.2 *Performance* of solutions obtained by SGC depending on the preset values of population size and number of evolution cycles.



- **Max. number of clusters in solution:** 50
- **Fitness:** *Performance*
- **Crossover:** Swapping randomly between *Clusterwise* and *Uniform* in expl. phase, no crossover in refining phase
- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation
- **Expl. phase mutations:** *Basic*[1] $p = 0.2$, *Joining* $p = 0.4$, *Splitting* $p = 0.4$
- **Refining phase mutations:** *Cluster Refining* $p = 0.4$, *MinDensity Splitting* $p = 0.2$, *MaxCut Joining* $p = 0.2$
- **Nr. of refined individuals:** 5

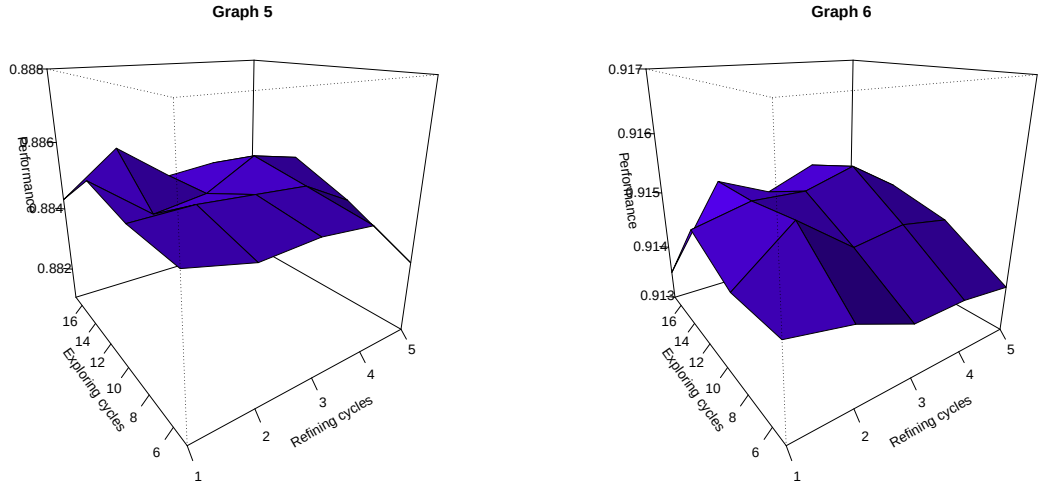
The tested lengths of the refining phase are generally shorter than for the exploring phase because there is a higher risk that the search process will get stuck in a local optimum is the exploiting tendencies last too long. More sophisticated mutations and usage of the refining procedure make running TPGC more time consuming than running SGC (if we measure average time needed for one evolution cycle). On the other hand, significantly lower number of cycles is needed to obtain solutions of equal or even higher quality than in the case of SGC.

The results of this experiment show that on the one hand, exact lengths of both phases do not have dramatic impact on the quality of the final solution but, on the other hand, it looks like that smaller lengths of the refining and exploring phase have a little bit greater chance to find better solution. This can be demonstrated by the graphs in the figure 5.3. The figure shows the fitness function value (*Performance*) of solutions obtained on graphs G_5 and G_6. Despite the fact that differences are not very strong, we can observe that the combinations of settings when the exploring phase length is around 10 cycles and the refining phase length is set around 2 cycles are a little bit more successful. For the graphs the situation is similar, definitely there is no graph for which the longer phases would be significantly more appropriate than the shorter ones.

The explanation of this phenomenon could be following: If the exploring phase is too long then there are too many random changes in the individuals that mostly don't contribute positively to the quality of the solutions and the refining phase is not able to "repair" these changes by the local search improvement of the individuals. On the other hand, if the refining phase lasts too many cycles, the exploiting strategy leads the individuals to get stuck in a locally optimal solution (a solution that can't be improved by simple hill-climbing techniques) and, thus, the population diversity decreases. If the quality of this local optimum is quite good (but there can still exist much more better solutions in the search space), it is not easy to escape from this local extremum by random changes during the exploration phase because the mutated individuals are typically worse than those lying in the local optimum, so they are quickly eliminated from the population by the selection operator. The consequence is that the search process is prevented

[1] with increased amount of genes affected by the mutation to 20% (on average)

Figure 5.3 *Performance* of the returned solutions depending on the refining phase and the exploring phase lengths.



from leaving the local optimum and from finding more promising solutions that can exist far from that point.

5.2.2 Combined fitness function test

This experiment has two goals: to test usability of a fitness function where two clustering quality measures are combined together (instead of using just one quality judging function as in the cases before) and to compare results of SGC and TPGC when this new fitness function is used.

In the previous experiments, the fitness function was realized by *Performance* quality measure. Because the graph clustering problem is quite complex, there is not exactly one function which would be the best one to be used as a subject of optimization (i.e., as the fitness function in the case of the genetic algorithm), so, multi-objective optimization (that can be also realized by the genetic algorithm) can be involved. There are many more or less sophisticated ways how to handle multi-objective optimization in the genetic algorithms, presented in various works on this theme, for example in [9]. One of the most straightforward approaches is to combine more functions into one fitness function by adding them together. This technique was used also in this experiment when *Modularization Quality* and *Performance* were combined in the following formula which was used as the fitness function in SGC and TPGC. If we want to evaluate solution C, its fitness is given:

$$Fitness(C) = \frac{1 + MQ(C)}{2} + Performance(C)$$

Same as in the preceding tests, we summarize the rest of settings of both clusterizers here. SGC was configured in the following way:

- **Population size:** 50
- **Evolution cycles:** 190

- **Max. number of clusters in solution:** 50
- **Crossover:** *Uniform*
- **Mutations (and their probabilities in *Master* mutation):** *Basic* $p = 0.2$, *Joining* $p = 0.4$, *Splitting* $p = 0.4$
- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation

The configuration of TPGC was following:

- **Population size:** 50
- **Evolution cycles:** 70
- **Max. number of clusters in solution:** 50
- **Crossover:** Swapping randomly between *Clusterwise* and *Uniform* in expl. phase, no crossover in refining phase
- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation
- **Expl. phase mutations:** *Basic*[2] $p = 0.2$, *Joining* $p = 0.4$, *Splitting* $p = 0.4$
- **Refining phase mutations:** *Cluster Refining* $p = 0.4$, *MinDensity Splitting* $p = 0.3$, *MaxCut Joining* $p = 0.3$
- **Expl. phase length:** 10
- **Refining phase length:** 2
- **Nr. of refined individuals:** 1

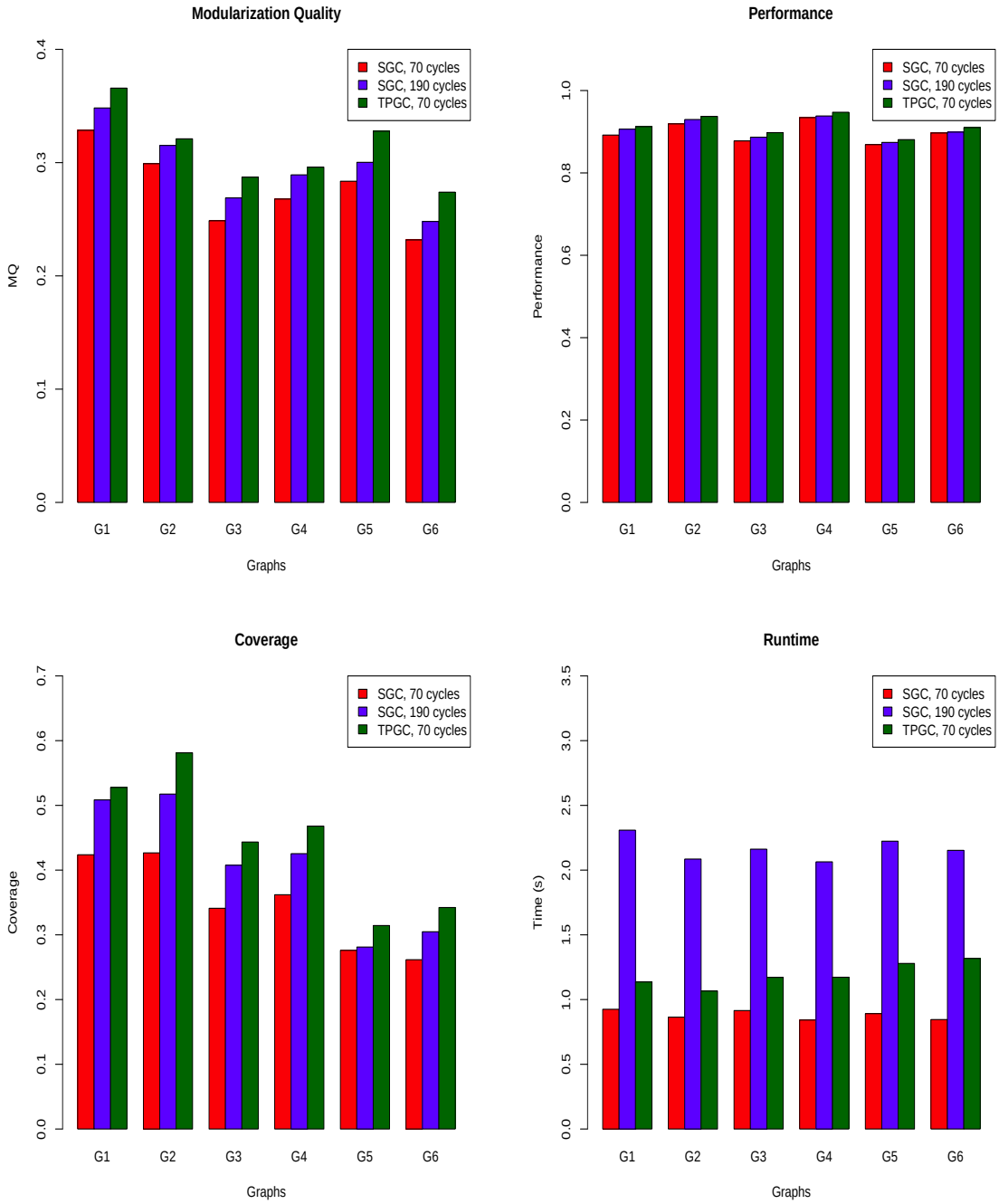
Both of the tested clusterizers were executed on all graphs $G_1, ..., G_6$ for fifty times to obtain average values for results that are presented in the table 5.10. In each cell of the table the first number is the result for TPGC and the second number (in parenthesis) is the result for SGC. SGC was set to perform 190 to make sure that the quality of the solution is stabilized and that the probability of finding remarkably better solution is low.

To enable better comparison of SGC and TPGC, we present four graphs in the figure 5.4. The graphs show average quality of solutions obtained from TPGC (run with the configuration listed above) and from SGC (also configured as presented earlier, but once with 190 evoluiton cycles and once with only 70 cycles).

As can be seen from the graphs, TPGC was more successful by means of all quality measures involved in all cases than both versions of SGC. The lower quality of solutions produced by SGC executed for 70 cycles was expected, but the results can be interesting when compared with the results of TPGC, which was executed for the same number of cycles.

[2]with increased amount of genes affected by the mutation to 20% (on average)

Figure 5.4 Comparison of SGC and TPGC (both used with combined fitness function), executed on experimental graphs $G_1, ..., G_6$



Graph	MQ	Performance	Coverage	Time (s)
G_1	0.366 (0.348)	0.912 (0.906)	0.528 (0.508)	1.14 (2.30)
G_2	0.321 (0.315)	0.937 (0.929)	0.581 (0.517)	1.07 (2.08)
G_3	0.287 (0.269)	0.898 (0.886)	0.443 (0.407)	1.17 (2.16)
G_4	0.296 (0.289)	0.947 (0.938)	0.468 (0.425)	1.17 (2.06)
G_5	0.328 (0.300)	0.881 (0.874)	0.314 (0.281)	1.28 (2.22)
G_6	0.274 (0.248)	0.910 (0.899)	0.342 (0.304)	1.32 (2.15)

Table 5.10: Comparison of TPGC and SGC with combined fitness function

5.3 IGC experiments

5.3.1 Comparison of TPGC and IGC - experiment I.

In this test, the effectiveness of TPGC and IGC was evaluated. Unlike in the previous experiments, new set of graphs was used here. Again, there were six graphs used, but these graphs differed not only in numbers of edges but also in numbers of nodes. The main goal of the experiment was to resolve whether it is better to use IGC which internally uses TPGC executed with smaller populations and for lower number of cycles or to use TPGC directly, executed only once but with increased evolution cycles count and greater population size.

For this experiment, new collection of testing graphs was created. Firstly, we have compared the clusterizers on two small graphs (S_1 and S_2) and two medium graphs (M_1 and M_2). The main features of these graphs are summarized in the table 5.11.

Graph	Nr. of nodes	Nr. of edges
S_1	25	157
S_2	25	190
M_1	75	1902
M_2	75	2146

Table 5.11: Sizes of experimental graphs S_1, S_2, M_1 and M_2

TPGC was executed with following configuration (again, ten times on each of the four graphs, average values are published):

- **Population size:** 20 for graphs S_1 and S_2, 50 for graphs M_1 and M_2
- **Evolution cycles:** 30 for graphs S_1 and S_2, 80 for graphs M_1 and M_2
- **Max. number of clusters in solution:** 25 for graphs S_1 and S_2, 75 for graphs M_1 and M_2
- **Fitness function:** Combined fitness function (same as in the experiment 5.2.2)
- **Crossover:** Swapping randomly between *Clusterwise* and *Uniform* in expl. phase, no crossover in refining phase

- **Selection:** *Tournament* with tournament size 8, $p = 0.75$ and the best individual preservation
- **Expl. phase mutations:** *Basic*[3] $p = 0.2$, *Joining* $p = 0.4$, *Splitting* $p = 0.4$
- **Refining phase mutations:** *Cluster Refining* $p = 0.4$, *MinDensity Splitting* $p = 0.3$, *MaxCut Joining* $p = 0.3$
- **Expl. phase length:** 10
- **Refining phase length:** 2
- **Nr. of refined individuals:** 1

IGC in this test was configured to use repeated executions of TPGC with these settings:

- **Population size:** 10 for graphs S_1 and S_2, 20 for graphs M_1 and M_2
- **Evolution cycles:** 15 for graphs S_1 and S_2, 30 for graphs M_1 and M_2

The rest of TPGC settings were the same as in the stand-alone execution of TPGC presented above. Additional configuration properties of IGC were set in the following way (see section 3.3 for explanations):

- **Quality judges:** *Modularization Quality, Performance*
- **Quality dominating number:** 1
- **Max. number of iterations without improvement:** 3

For the small graphs S_1 and S_2, there was also one more configuration of IGC evaluated with lower settings for population size in TPGC executed by IGC. The population size was decreased to only 5 individuals and tested on both small graphs (this configuration is further referred as IGC2 in this experiment, while the first configuration is referred as IGC1).

The results for graphs S_1 and S_2 are shown in tables 5.12, 5.13 and 5.14.

Graph	MQ	Performance	Coverage	Time (s)
S_1	0.518	0.999	0.876	0.088
S_2	0.521	0.908	0.568	0.099

Table 5.12: Results of TPGC on graphs S_1 and S_2

Graph	MQ	Performance	Coverage	Time (s)
S_1	0.525	1.001	0.885	0.119
S_2	0.519	0.908	0.573	0.247

Table 5.13: Results of IGC1 on graphs S_1 and S_2

Similarly we present results for medium graphs in tables 5.15 and 5.16. On these graphs, IGC2 was not evaluated (i.e., only results for IGC1 are published).

Graph	MQ	Performance	Coverage	Time (s)
S_1	0.525	1.001	0.885	0.084
S_2	0.518	0.906	0.571	0.146

Table 5.14: Results of IGC2 on graphs S_1 and S_2

Graph	MQ	Performance	Coverage	Time (s)
M_1	0.646	1.079	0.878	4.32
M_2	0.535	0.845	0.583	5.90

Table 5.15: Results of TPGC on graphs M_1 and M_2

As it can be seen from the results, none of the tested clusterizers works better in all cases. Sometimes, IGC is able to produce solutions of the same or even higher quality than TPGC in shorter runtime, but on another graph it can consume more runtime than TPGC without any gain in solution quality. To investigate this more, we can analyze the average number of clusters in the solutions produced by TPGC for each of the graphs S_1, S_2, M_1 and M_2. This observation is presented in the table 5.17.

If we put the information from the the table 5.17 and from the results of TPGC and IGC on all four graphs together, we can see that for graphs where the number of clusters in good solutions (or even in the optimal solution) is low, IGC can perform better than TPGC, because it can found a solution of similar quality faster than TPGC. On the other hand, if the expected number of clusters in promising solutions is higher, IGC needs to perform too many iterations (i.e., repeated executions of internal TPGC) to detect this and, thus, is slower than stand-alone TPGC.

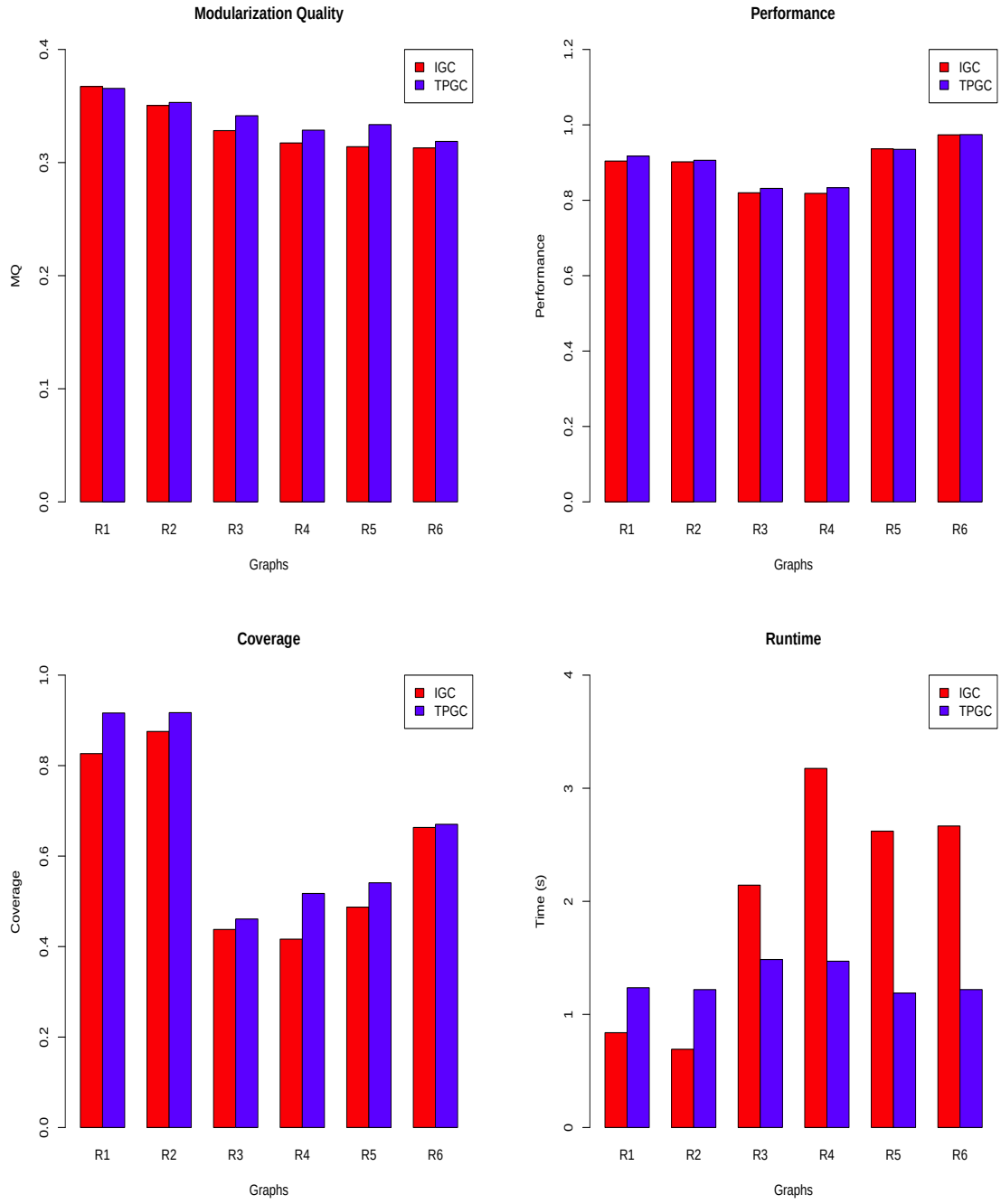
5.3.2 Comparison of TPGC and IGC - experiment II.

To verify the hypothesis from the previous experiment, the second test comparing IGC and TPGC was arranged. New set of six graphs $R_1, ..., R_6$ was generated for this purpose, each of the graphs with 50 nodes. The graphs differed not only in the number of edges but also in the distribution of intra-cluster and inter-cluster edges. To build these graphs we have used a simple generator implemented within the *Graph clusterizer* application in the class `RandomGraphGenerator`. This generator allows the user to set desired number of clusters in the graphs and also the ratio between intra-cluster and inter-cluster edges. However, because it is difficult to characterize good clustering by any objective function, the really optimal solution could not always be the same as the desired solution preset in the generator when specifying the graph. For details about the generator and the way how the graphs are produced, see the source code and its documentation (both can be found on the attached CD). The graphs were generated with the properties summarized in the table 5.18.

The graphs from the experimental set are of three types: with low desired number of clusters that should be very obvious (high modularization ratio) - represented by graphs R_1 and R_2, with low number of clusters again and also with low

[3] with increased amount of genes affected by the mutation to 20% (on average)

Figure 5.5 Comparison of IGC and TPGC on random experimental graphs $R_1, ..., R_6$



Graph	MQ	Performance	Coverage	Time (s)
M_1	0.646	1.079	0.878	2.88
M_2	0.539	0.855	0.562	6.15

Table 5.16: Results of IGC1 on graphs M_1 and M_2

Graph	Avg. number of clusters
S_1	3.2
S_2	5.7
M_1	3.0
M_2	5.2

Table 5.17: Cluster counts in solutions produced by TPGC

modularization (i.e., the clusters are harder to discover) - represented by graphs R_3 and R_4 and graphs with higher number of clusters and high modularization ratio (R_5 and R_6).

According to the results of the previous experiment, IGC should be more suitable for graphs R_1 and R_2 while TPGC can be better for the rest of the experimental graphs. This expectation is supported by the bar plots in the figure 5.5. The figure shows average results from ten executions of both clusterizers on graphs $R_1, ..., R_6$. The configurations of the clusterizers were the same as in the previous experiment except the number of cycles and the population size. For stand-alone execution of TPGC, these settings were following:

- **Population size:** 50
- **Evolution cycles:** 80

While in the case of IGC, the configuration was:

- **Population size:** 15
- **Evolution cycles:** 20

The results of both clusterizers are presented in the figure 5.5. For graphs R_1 and R_2, IGC reached almost the same or even higher quality of solutions as TPGC while sparing non-trivial amount of runtime, so it can be consider more appropriate clusterizer for this type of graphs. On the other hand, for the other graphs, TPGC is definitely faster without loss of quality (or the solutions produced by TPGC are even better). The fitness landscape for graphs R_3 and R_4 is probably more complicated because of more uniform distribution of edges, so IGC is not able to discover good solutions in short iterations. In the case of graphs R_5 and R_6, IGC has to perform too many iterations to discover good solutions (because of the higher number of clusters), so the runtime rises too much. For the graphs R_1 and R_2, the convergence of the algorithm is faster, because the clusters are more apparent and the fitness landscape is probably more simple, so fast iterations performed by IGC are able to discover promising solutions in a short time.

Graph	Nr. of edges	Modularization ratio	Desired nr. of clusters
R_1	491	0.85	3
R_2	473	0.85	3
R_3	523	0.65	3
R_4	539	0.65	3
R_5	209	0.85	5
R_6	109	0.85	10

Table 5.18: Properties of graphs $R_1, ..., R_6$

Conclusion

The aim of this work was to analyze usage of genetic algorithms for clustering in the domain of directed weighted graphs. The problem of clustering is very complex and the basic genetic algorithm enables many modifications and upgrades, so it is not possible to perform complete analysis of this task. Wide range of possibilities offers the way of testing new genetic operators suited for graph clustering. In this work, several clustering operators were proposed, implemented and their usage in different algorithms was evaluated. The operators can be basically divided into two groups: the exploring ones and the exploiting ones. While the exploring operators are typically faster, simpler and modify individuals without any additional information about the problem, the exploiting operators can use more detailed knowledge about the nature of the task solved and, thus, can perform more sophisticated modifications of solutions. Both types of operators bring some drawbacks - slow convergence towards optimal solutions and high risk that the optimal solutions will be missed on the one hand and possibility of reaching just locally optimal solutions and ignoring too large areas of the search space on the other hand. Whether to prefer the first type of operators or the letter one is one of the main dilemmas in applications of evolutionary algorithms. In this thesis, we proposed several operators of both types and evaluated some configurations where these operators were combined together. Two clusterizers were designed - one which exactly copies the scheme of the basic genetic algorithm (this clusterizer is referred as SGC in the text, see section 3.1) and one which deals with the problem of exploration vs. exploitation by alternating two different sets of operators during the evolution (referred as TPGC. see section 3.2).

Another problem in a clustering task is how to deal with the situation when the desired number of clusters in the solution is not known a priori. If the number of clusters is unlimited, the search space can become much larger and it can be too complex to find good solutions. In this work, we have designed so called Iterative Genetic Clusterizer (IGC, see section 3.3) that tries to limit the search space by increasing the limit for the number of clusters stepwise and by repeated executions of some another clusterizer with this new limit applied.

Finally, an application that contains implementations of proposed clusterizers was developed and published via the SourceForge[4] portal. This application has a client-server architecture which makes it suitable for usage in distributed network environments. Modular design of the application makes it easily extendible with new implementations of clusterizers. Implementation in *Java* programming language makes it independent on concrete operating system.

Results

The results of the experiments show that the performance of a genetic algorithm based clusterizer can be improved by combining both types of operators and alternating between them in short periods during the computational process. The results indicate that short lengths of periods of using one set of operators

[4] http://sourceforge.net/projects/gclusterizer/

are more promising for finding better solutions. Shorter phases probably help the algorithm not to stay in locally optimal solutions while still protecting it from destroying good solutions by too much exploring search.

The tests that were performed show that concept of IGC can be useful, but not in all cases. If the number of clusters in good solutions is low, IGC needs just few iterations to find them and these iterations do not consume too much time because the searching runs in very limited space, so, IGC is faster than other clusterizers without loosing much quality of the solutions in such cases. So, if the clusters are obvious enough, IGC can be faster than other implemented clusterizers. There are also other methods how to tackle this problem of unknown optimal number of clusters. For example, the concept of hierarchical clustering can be adopted, as described in section 2.1.1.

Future work

There are many ways how to implement new genetic operators intended for graph clustering, so further research on this topic could be surely made. Especially new smart exploring operators can be designed. The successful exploitive operators should take into account existing clusters in the modified individuals and use some heuristic to improve them. Smart splitting and joining of clusters looks promising, or slower rebuilding of clusters by including or excluding just single nodes. The research can be in the area of suitable heuristic functions to be used in such operators. Also the options of informed crossover operators could be analyzed more, for example crossovers based on greedy merging of clusters from parental solutions.

Implementation of the idea of hierarchical clustering using the genetic algorithm and comparison with the other genetic clusterizers could be a straightforward extension of this work. New genetic operators suited for hierarchical splitting of the set of graph nodes could be designed, the operators that use some fast approximations of minimal cut could be promising for this purpose.

The developed application could become a core of some service, for example exposed via web interface as a public web service. Many various client programs can be developed and used thanks to the defined communication protocol between client and server which uses text commands. One of the main goals in further development of the application should be allowing parallel executions of more than one clusterizer, issued by multiple users (especially if the application would be available as a public web service). The program is well prepared for this, because the clusterizers are executed in separate threads even in the current version. Another extension of the application could be the implementation of routines for loading graphs from widely used formats such as *TGF* or *GraphML*. This functionality could be useful especially in the case of exposing the service for public use when the graphs would be passed to the application by the user using a web form or some similar type of input.

Another way of improving performance of the application can be parallelization of computation. The evaluation of individuals (i.e., computing their fitness function values) typically needs read-only access to the individuals, so sharing the instances among multiple threads should be easy. If we use an additive fitness function, its evaluation can be than divided among parallel threads. Because the

evaluation of fitness function often needs to cycle over all edges of the graph, the fitness function computation can become a bottleneck of the algorithm. In this case, parallelization could significantly help. For example, in [11] the authors propose parallelization of genetic algorithms according to the *MapReduce* paradigm[3] using *Hadoop*[5] framework which could be a promising approach also for this work.

[5]http://hadoop.apache.org/

CD

A CD is attached to this thesis which contains the *Graph clusterizer* application and other software referred in the text and also graph files used for the experiments. The complete list of the contents is:

- *Graph clusterizer* application, including source code and documentation (the entire project for *NetBeans* IDE, managed by the *Maven* tool) - in the `applications` directory
- *GCScriptWorker* application, including its source code (*NetBeans* project) - in the `applications` directory
- *AbuGraph* application (the complete package that can be downloaded from the application's homepage) - in the `applications` directory
- *TGFConvert* - a simple utility for converting graph files from *TGF* to *Graph clusterizer's* proprietary format (complete *NetBeans* project) - in the `applications` directory
- *The Java Telnet Application* - a program that can be used as a client for connection with the clustering server - in the `applications` directory
- Graph files with graphs $G_1, ..., G_6$ (50 nodes) used for experiments - in the `graphs/50_graphs` directory
- Graph files with graphs S_1, S_2, M_1 and M_2 from the IGC experiment - in the `graphs` directory
- Graph files with graphs $R_1, ..., R_6$ used for the last experiment with IGC
- Text of this thesis in *PDF* file

Copy of the CD can be downloaded from the following URL:
`http://www.ms.mff.cuni.cz/~kohoj7am/download/jan_kohout_dpcd.zip`

Bibliography

[1] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, 1987.

[2] U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. *11th Europ. Symp. Algorithms*, 2003.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[4] C. R. Dias and L. S. Ochi. Efficient evolutionary algorithms for the clustering problems in directed graphs. *Proc. of the IEEE Congress on Evolutionary Computation, IEEE Computer Press*, pages 983–988, 2003.

[5] Klaus Meffert et al. Jgap - java genetic algorithms and genetic programming package. http://jgap.sf.net.

[6] M. Gaertler. Clustering. In Brandes and Erlebach, editors, *Netowrk Analysis*. Springer, 2005.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.

[8] E. Hruschka, R. Campello, A. Freitas, and A. de Carvalho. A survey of evolutionary algorithms for clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 2009.

[9] M. Pilát. Parallel evolutionary algorithms for multiobjective optimization. Master's thesis, MFF UK, Prague, 2009.

[10] J. Sima and S. E. Schaeffer. On the np-completeness of some graph cluster measures. In *SOFSEM 2006: Theory and Practice of Computer Science*. Springer Berlin / Heidelberg, 2006.

[11] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. In *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, ISDA '09, pages 13–18, Washington, DC, USA, 2009. IEEE Computer Society.

[12] M. D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. The MIT Press, 1999.