FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Alexander Ocheretyanyy

Balanced Partitioning of Graphs with Small Vertex Cover

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Dr. Andreas Emil Feldmann
Study programme:	Computer Science (B1801)
Study branch:	IOIA (1801R008)

Prague 2020

To my son Maxim whom I love with all my heart. Forgive me, my little boy, for not being with you.

Title: Balanced Partitioning of Graphs with Small Vertex Cover

Author: Alexander Ocheretyanyy

Department: Department of Applied Mathematics

Supervisor: Dr. Andreas Emil Feldmann, Department of Applied Mathematics

Abstract: The Balanced Partitioning problem is an important problem from the NP-class of problems, that finds its practical applications in parallel computing, data mining, VLSI design, and others. In the thesis we try to solve the problem practically using two kinds of algorithms - a Fixed-Parameter Tractable Algorithm, and an Evolutionary Algorithm, providing the results of many experiments performed.

Keywords: Balanced Partitioning, Vertex Cover, Genetic Algorithm, FPT algorithm

Contents

1. Introduction

The purpose of this bachelor thesis is to implement a fixed-parameter algorithm for the Balanced Partitioning problem parameterized by the vertex cover, and compare it with an Evolutionary Algorithm.

The Balanced Partitioning problem is a clustering-type problem in which the clusters must be of almost equal size. At the same time, the desired number d of clusters is given, and we want to minimize number of cross-edges between different clusters. That is, given a graph $G = (V, E)$ and a number of clusters d, we want to partition the graph in such a way, that every cluster has at most k vertices from the vertex set V, where $k = \lceil \frac{|V|}{d} \rceil$, and we want the partition to cut as few edges from the set E as possible, where an edge $e = \{u, v\}$ is said to be *cut* if both endpoints u and v lie in different clusters of the partition. The *cost of a solution* is the number of edges the partition cuts.

The Balanced Partitioning problem finds practical applications and, in particular, is extremely important in parallel computing (e.g. assigning data evenly to many processors). Also, it is used in circuit design [20], load balancing [22], biological networks, on-line social network analysis and graph databases.[7] A famous IT company Google uses balanced graph partitioning for Google Maps driving directions.[5]

Andreev and Racke [3] proved that the Balanced Partitioning problem is NP-hard, that is there does not exist an algorithm that solves the problem in polynomial time, unless NP=P. Thus, to solve the problem in this thesis we will use two different approaches to NP-hard problems.

The first approach is to partition the set of all possible graphs according to some unique property they have, and then to solve the problem only for some of the equivalence classes, that is, we restrict the problem to some subset of possible inputs for which we can efficiently solve the problem using a so-called Fixed-Parameter Tractable (FPT) algorithm.

Definition 1 ([8, p. 12–13])**.** *A parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the parameter.*

A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called fixed-parameter tractable (FPT) if there exists an algorithm $\mathcal{A}$ (called a fixed-parameter algorithm), a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

The biggest advantage of FPT algorithms is the fact that they always provide us with optimal solutions of problems they solve.

Definition 2. *Let $G =$ (V, E) be a graph. A set $C \subseteq V$ is a vertex cover of G if for all $\{u, v\} \in E$ either $u \in C$ or $v \in C$.*

The algorithm we will use in this thesis was described by van Bevern et al. [23], and it successfully solves the problem for graphs with small vertex covers. Of course, the word "small" is ambiguous and one cannot definitely say what is considered to be "small". Nevertheless, we can use an empirical approach and, given a particular hardware domain, decide what size of the vertex cover a graph should have in order for the Balanced Partitioning Problem to be efficiently solvable by the FPT algorithm. However, using this approach we solve the problem only partially because we cover only a small subset of the set of all possible graphs (which is infinite), and we are still not able to solve the problem for general graphs.

The second way to solve the problem is a heuristic approach where we will employ an Evolutionary Algorithm (EA) that is nowadays widely used in artificial intelligence (AI).

Definition 3 ([25, p. 6])**.** *Evolutionary Algorithms (EAs) are algorithms that perform optimization or learning tasks with the ability to evolve. They have three main characteristics:*

- *Population-based. EAs maintain a group of solutions, called a population, to optimize or learn the problem in a parallel way. The population is a basic principle of the evolutionary process.*
- *Fitness-oriented. Every solution in a population is called an individual. Every individual has its gene representation, called its code, and performance evaluation, called its fitness value. EAs prefer fitter individuals, which is the foundation of the optimization and convergence of the algorithms.*
- *Variation-driven. Individuals will undergo a number of variation operations to mimic genetic gene changes, which is fundamental to searching the solution space.*

In this thesis we will use so-called Genetic Algorithm (GA) as the most popular type of EA. We expect it to be quite efficient in terms of computational time and to be able to solve the problem not only for graphs with small vertex cover, but for some wider range of inputs. Nevertheless, this approach has several drawbacks as well. First of all, as any heuristic approach, it does not guarantee a solution to be optimal. More precisely speaking, in contrast to approximation algorithms, we even cannot estimate the quality of a solution where by the term "quality" we implicate the distance of a solution found by the GA from an optimal one. Second of all, there does not exist an optimal setting of the GA and to find some well behaving parameters we still need a lot of physical and computational time to deduce them empirically. Notwithstanding, having the FPT algorithm, we can find an optimal solution that we can use to assess the quality of the GA's output.

Since we also want to compare computational time of the algorithms, for their implementation we will use C++: primarily due to its computational efficiency (unlike Java or C#, C++ does not use a virtual machine and does not have a garbage collector that could interrupt, and thus slow down computations), and secondarily for its object oriented facilities that we will use during implementation of the GA.

The program and the test data can be found in the author's repository in [17] and also on the DVD attached to this thesis.

1.1 Related work

Kernighan and Lin [11] used a heuristic approach to the problem - so-called ”Kernighan-Lin algorithm”. Van Bevern et al. [23] developed an FPT algorithm.

Rahimian et al. [19] provided a distributed balanced graph partitioning algorithm that they called ”Ja-BE-Ja” (”swap” in Persian). Kim, H. and Kim, Y. [13] gave an overview of evolutionary computation application towards the problem. Kim et al. [12] gave an overview of different approaches to genetic operator design and encountered problems that may arise during a GA implementation. Pope et al. [18] tried to solve so-called Bisection problem which is in fact the Balanced Graph Partitioning problem where the number of clusters is equal to 2. Sanders and Schulz [21] used a multilevel partitioner that splits input graphs into smaller ones and then works with them in parallel. This approach allows to partition huge graphs (the authors of the article even used graphs with 51M vertices). Finally, Buluc et al. [7] gave an overview of many techniques that are used for the problem solving.

The reader also may be interested in [4] that presents a collection of topic related articles.

As for implementations of some graph partitioning algorithms, the reader might be interested in METIS project, which is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices.[1]

To the best of the author’s knowledge no one tried to compare different settings of GA’s and to figure out whether they find an optimal solution for the Balanced Graph Partitioning problem or not.

2. FPT algorithm

The FPT algorithm for the Balanced Partitioning problem we will use is described in [23]. This algorithm has the time complexity $\mathcal{O}(\tau^\tau \cdot n^3)$, where τ is the *vertex cover number* (i.e. the size of a smallest vertex cover). In our case τ is the size of the vertex cover that we provide as the input to the algorithm, i.e. we assume that we have already known the size of a smallest vertex cover of the input graph.

2.1 The algorithm

To describe the algorithm we will need the following definition:

Definition 4. *Let $G = (V, E)$ be a graph. We say that a set $S \subseteq V$ is an independent set of G if no two vertices in S are adjacent.*

Let us also prove the following useful theorem:

Theorem 1. *Let $G = (V, E)$ be a graph and let C be a vertex cover in G. Then $I = V \setminus C$ is an independent set in G.*

Proof. Let us assume that I is not an independent set i.e. it does not satisfy Definition 4. Then there exist $u, v \in I$ s.t. $u \neq v$ and $e = \{u, v\} \in E$, that is both u and v are adjacent in G. That implies that both of the endpoints of e do not belong to C and that contradicts to the fact that C is a vertex cover, since by Definition 2 either u or v must be in C. □

The essence of the algorithm is the following:

Given a graph $G = (V, E)$ and positive numbers τ (size of the vertex cover) and d (number of clusters of a partition). Let $n = |V|$ be the size of the vertex set of G.

1. Find a vertex cover C of size τ of G.
2. We consider all partitions of C into d sets (clusters). For each partition of C we check whether some cluster has more than $k = \lceil \frac{n}{d} \rceil$ vertices. If this is the case, the partition is discarded. Otherwise we need to partition the independent set $I = V \setminus C$ (Theorem 1) not belonging to the vertex cover.
3. We need to compute an assignment of the vertices in the independent set I to the clusters such that the number of vertices in each cluster does not exceed k and the total introduced cost is minimal.
4. By going through the above steps and picking the best solution among all partitions of C that are not discarded, the minimum cut size can be computed.

2.2 Graph representation

We will work with a graph G and a number of clusters d. Each cluster must contain at most $\lceil \frac{|V|}{d} \rceil$ vertices.

The graph G can be represented in memory either as an adjacency matrix, or as an adjacency list. In case of adjacency matrices we get $\Theta(1)$ time complexity for checking, whether there is an edge between two vertices or not, but its space complexity is $\Theta(n^2)$. In case of adjacency lists we have that time complexity for checking whether there is an edge between two vertices or not is $\mathcal{O}(m)$, and space complexity is $\Theta(m+n)$, where m is the number of edges in the graph. In both cases each vertex of G has a unique index.

In our experiments we will have small size of the vertex cover due to the time complexity of the FPT algorithm, and that implies that we will deal with sparse graphs. Usage of the latter graph representation is more space efficient for such type of graphs, and thus we will use adjacency list as a basic structure.

2.3 Vertex Cover algorithm

To find a vertex cover of a given graph $G = (V, E)$ we will use the so-called *Search Tree* technique from [10]. The only adjustment will be usage of an accumulator that will collect vertices of a vertex cover.

Lemma 2. *Let $G = (V, E)$ be a graph, and let $k \in \mathbb{N}$ be the size of a vertex cover. The recursive function $VertexCover$, described in Algorithm 1 and called with the arguments $(V, E, \emptyset, k)$ finishes, and when it finishes it returns a vertex cover of size $\leq k$ in G, if it exists.*

Proof. If G does not contain any edges, then the function finishes and returns the empty set, which is, indeed, a vertex cover of G.

Otherwise, if G has at least one edge, then at the initial step the set T is empty. The function picks a random edge $e = \{u, v\}$ from E. Definitely, in any vertex cover C of G either $u \in C$ or $v \in C$. Thus, the function makes recursive calls for $G - u$ as the left child, and for $G - v$ as the right child. For the left child the function adds u to the set T, for the right child it adds v to the set T. For both of the calls it sets $k = k - 1$.

Steps 9 and 10 of the algorithm guarantee that k always decreases. If at some point $k = 0$ but $E \neq \emptyset$ then the function returns *null*. That means that the accumulator T does not contain a vertex cover of G yet, and if we continue its construction we will exceed the required size of the vertex cover. Thus, we do not need to continue this branch of the recursion tree and may immediately terminate it.

We may notice that steps 7 and 8 of the algorithm ensure that the sets E' and E'' are not empty since both of them contain at least one edge - e that was picked at step 6. At steps 9 and 10 we always extract one vertex from the vertex set V and delete at least one edge from the edge set E and thus the set of unprocessed edges decreases at each call of the function. Since the edge set of G is finite, at some point the algorithm must exhaust it. It follows that at some point of execution of the algorithm it will call $VertexCover(V', \emptyset, T', k')$ for some vertex set $V' \subset V$ and $k' \in \mathbb{N} \cup \{0\}$, which returns T'. Thus, the algorithm always finishes.

Also, we notice that T accumulates vertices that are in some vertex cover of G. When the algorithm finishes it returns some vertex set T' of accumulated vertices which is in fact a vertex cover of G by the construction - steps 9 and 10 guarantee that for any $e \in E$ s.t. $e = u, v$ either u or v has been added to set T.

If the initial k is bigger than the smallest vertex cover of G, then we may observe that when left and right children are returned and contain some vertex sets T' and T'' accordingly, the algorithm returns the smallest set among those two. Thus, when the algorithm stops its execution it returns the smallest vertex cover which is constructed by the shortest branch of the recursion tree. ☐

Algorithm 1: Vertex Cover Algorithm

Input : Graph G represented as an ordered pair of its vertex and edge set (V, E), set $T \subseteq V$, k - size of a vertex cover

Output: A vertex cover of size $\leq k$ if it exists, *null* otherwise

```
Vertex Cover($V, E, T, k$):
1   if $k = 0$ and $E \neq \emptyset$ then
2       return null;
3   if $E = \emptyset$ then
4       return $T$;
5   else
6       Pick $e = \{u, v\} \in E$ at random;
7       Let $E' \subseteq E$ s.t. for all $\{a, b\} \in E$ if $a = u$ or $b = u$ then $e \in E'$
          (i.e. $E'$ is the set of all edges of $G$ incident to $u$);
8       Let $E'' \subseteq E$ s.t. for all $\{a, b\} \in E$ if $a = v$ or $b = v$ then $e \in E''$
          (i.e. $E''$ is the set of all edges of $G$ incident to $v$);
9       $leftChild$ = Vertex Cover($V \setminus \{u\}$, $E \setminus E'$, $T \cup \{u\}$, $k-1$);
10      $rightChild$ = Vertex Cover($V \setminus \{v\}$, $E \setminus E''$, $T \cup \{v\}$, $k-1$);
11      if $leftChild = null$ and $rightChild = null$ then
12          return null;
13      else if $leftChild = null$ and $rightChild != null$ then
14          return $rightChild$;
15      else if $rightChild = null$ and $leftChild != null$ then
16          return $leftChild$;
17      else if $|leftChild| <= |rightChild|$ then
18          return $leftChild$;
19      else
20          return $rightChild$;
```

Let us assume that the input graph G has a vertex cover of size at most k. From Algorithm 1 we can see that at each call it makes exactly two recursive calls. Thus, we can describe it with the following reccurence formula:

$$T(|E|, k) \leq 2 \cdot T(|E|, k-1) + \mathcal{O}(m).$$

Here, $\mathcal{O}(m)$ represents the time needed to delete a vertex and its neighbours from G and ignores the fact that at every call of the function the size of G decreases. Solving the inequality we reveal that the time complexity of this algorithm is $\mathcal{O}(2^k \cdot |E|)$, where k is the height of the recursion tree.

As we can notice, Algorithm 1 always exhausts all possible branches of the recursion tree before it stops. It may be helpful in case we do not know the parameter k in advance and we just want to find a smallest vertex cover (in this case we set up $k = n$ as the maximum depth of the recursion tree). However, if we know the size of a smallest vertex cover, then we actually do not need to traverse the whole recursion tree - we only need to find the first vertex cover of size k. Thus, when implementing Algorithm 1, we will use a shared variable that, if set up to some vertex cover, will indicate that we can stop the algorithm since we have already found a vertex cover of size k.

Also, when k is not known, we do not need to explore the recursion tree to any depth bigger than the size of a smallest vertex cover. Thus, instead of going to the maximum depth of each branch of the recursion tree we could traverse the recursion tree as a breadth first search tree, i.e. layer by layer.

2.4 Enumeration of partitions

As it follows from the Step 4 of the algorithm from Section 2.1, we need to try all possible partitions of a given graph.

When we partition a vertex set we create clusters and assign indices to them. Thus, there are many partitions that differ only by indices chosen for each cluster, i.e. they are isomorphic to each other. If we go through all such partitions we waste computational time because we check partitions identical by their nature. To avoid such a situation we want to index clusters in such a way that we never encounter isomorphic partitions.

To achieve this goal we will employ an algorithm that enumerates restricted growth strings in lexicographic order, which is described in [14].

Definition 5. *A sequence of integers* $(a_0, a_1, ..., a_{n-1})$ *is a restricted growth string of a partition on n-sets if* $a_0 = 0$ *and for all* $i > 0$ *it holds that* $0 \le a_i \le \max\{a_0, a_1, ..., a_{i-1}\} + 1$.

Given $n \ge 2$, Algorithm 2 generates all partitions of $\{1, 2, ..., n\}$ by visiting all strings $a_1 a_2 ... a_n$ that satisfy the restricted growth condition: $a_1 = 0$ and $a_j \le 1 + \max(a_1, ..., a_j)$ for $1 \le j < n$ and then returns the set of such strings. We exploit an auxiliary array $b_1 b_2 ... b_n$, where $b_{j+1} = 1 + \max(a_1, ..., a_j)$.

The analysis of the algorithm's time complexity is done in [14]. We just mention that it is $\mathcal{O}(B_{|C|} \cdot \frac{\ln |C|}{|C|})$, where B_n is the n-th Bell number, and C is the vertex cover size.

Finally, now we should vividly see why the FPT algorithm requires a minimum vertex cover to be small. Definitely, the number of possible partitions of a vertex cover $|C|$ into d clusters is equal to $d^{|C|}$. This value grows exponentially with growth of the vertex cover size and thus to guarantee that the algorithm stops after some reasonable time we have to make sure that $|C|$ is small.

Algorithm 2: Restricted growth strings

Input : $n \in \mathbb{N}$ - the length of a string
Output: The set of restricted growth strings of size n

```
RGS(n):
    $S = \{\emptyset\}$ ; // The output set
    for $i \leftarrow 1$ to $n$ do
        $a_i = 0$;
        $b_i = 1$;
    $S = S \cup \{\{a_1, a_2, ...\ a_n\}\}$;
    if $a_n == b_n$ then
        go to 10;
    $a_n = a_n + 1$;
    go to 5;
    $j = n - 1$;
    while $a_j == b_j$ do
        $j = j - 1$;
    if $j == 1$ then
        return $S$;
    else
        $a_j = a_j + 1$;
    $b_n = 1 + b_j$;
    $j = j + 1$;
    while $j < n$ do
        $a_j = 0$;
        $b_j = b_n$;
        $j = j + 1$;
    $a_n = 0$;
    go to 5;
```

2.5 Matching

Definition 6. *Let $G = (V, E)$ be a graph. A set of edges $M \subseteq E$ is a matching in G if no two edges $e_1, e_2 \in M$, $e_1 \neq e_2$ share a common vertex.*

M is said to be a maximum matching in G, if it has the maximal cardinality among all matchings of G.

M is a perfect matching in G if it is maximal, and moreover for all $v \in V$ there exists $e = \{a, b\} \in M$ s.t. either $a = v$ or $b = v$, i.e. every vertex of G is matched by M.

Let $c : E \to \mathbb{R}$ be a function defined on the edges of G (cost function). A matching M is a minimum cost maximum matching in G, if M is maximum, and moreover the sum of all costs of edges in M is minimal among all maximum matchings of G.

When we have found a vertex cover and partitioned it, we are left with an independent set. According to Step 3 of the algorithm in Section 2.1 we need to find a correct assignment of the vertices from the independent set to the clusters in such a way that the total cost of the solution is minimal. To achieve this goal we can use a minimum cost maximum matching in an auxiliary graph which we construct next.

2.5.1 Auxiliary graph

Let us assume that for a given graph $G = (V, E)$ we have a vertex cover and some partition of the vertex cover into k clusters. Since the partition was not discarded by the algorithm at Step 2 we know that each cluster has less than or exactly $\lceil \frac{|V|}{d} \rceil$ vertices.

From the formulation of the Balanced Partitioning problem in Chapter 1 we know that the algorithm must produce only those partitions in which every cluster has less than or exactly $\lceil \frac{|V|}{d} \rceil$ vertices. Thus, assigning the vertices of the independent set to the clusters, we want to satisfy this constraint. That means that we need to identify those clusters that have *vacant places* in them (i.e. clusters that have less than $\lceil \frac{|V|}{d} \rceil$ vertices and the number of vacant places in a cluster is just the difference between $\lceil \frac{|V|}{d} \rceil$ and the factual number of vertices that are already placed in this cluster) and try to assign the vertices of the independent set to those vacant places.

Definition 7. *A graph $G = (V, E)$ is bipartitie if the vertex set V can be represented as a union of two disjoint sets A and B, i.e. $V = A \dot\cup B$, and every edge $e \in E$ has one endpoint in A and the second endpoint in B.*

A bipartite graph $G = (A \dot\cup B, E)$ is complete if every vertex from A is connected to every vertex in B by an edge from E.

We build a bipartite graph $G = (A \dot\cup B, E)$ where A is the independent set and B consists of vacant places in the clusters of a partition. The number of vacant places is always bigger than or equal to the size of the independent set, otherwise some cluster would have hold more than $\lceil \frac{|V|}{d} \rceil$ vertices by the Pigeonhole principle stated below:

Definition 8. *(The Pigeonhole principle) If n objects are put into m containers, and $n > m$, then at least one container must contain more than one object.*

Now, connect each vertex of A to each vertex of B and define a cost function $q : E \to \mathbb{N}$ in such a way that for all $u \in A$ and for all $v \in B$, $q(u) =$ *the number of cut edges*, i.e. the number of edges a partition cuts if the vertex u is assigned to the vacant place v of the cluster it belongs to.

We are almost done. To complete this construction, in case $|A| < |B|$ we add new vertices (*artificial vertices*) to set A to make $|A| = |B|$ and connect every new vertex to every vertex in B thus making the graph complete. We also assign cost 1 to newly created edges.

Now it should be vividly seen that if we find a minimum cost perfect matching in the auxiliary graph, it corresponds to an assignment of each vertex of A to a vertex of B, i.e. an assignment of the vertices of the independent set to the clusters of the partition. If we eliminate all the artificial vertices that we created earlier to make both sides of the auxiliary graph of equal size, then this partition satisfies the constraint that all clusters must contain less than or equal to $\lceil\frac{|V|}{d}\rceil$ vertices. Also, such a partition has minimum cut size for the fixed partition of the vertex cover, since the costs of the edges in the auxiliary graph represent the arisen cut edges due to the partition of the independent set.

The only problem is to decide whether we can always find a perfect matching in the auxiliary graph. Fortunately, Philip Hall in 1935 proved the so-called Hall's marriage theorem. To introduce the theorem we need the following definition:

Definition 9. *Let $G = (V, E)$ be a graph, and let $v \in V$ be a vertex. The neighbourhood of v, denoted by $N(v)$, is the set of vertices adjacent to v, i.e. $N(v) = \{u \in V | \{u, v\} \in E\}$.*

If $S \subseteq V$, then $N(S)$ denotes the union of the neighbourhoods of all vertices in S.

Now we introduce the Hall's theorem [6]:

Theorem 3. *(Hall's Marriage Theorem) Let $G = (A \dot\cup B, E)$ be a bipartite graph. A can be matched into B if and only if $|N(S)| \leq |S|$ for all subsets S of A.*

However, we actually need a corollary from the Hall's theorem that can be stated as the following:

Corollary 1. *If a graph $G = (V, E)$ is bipartite and every vertex $v \in V$ has the same degree (i.e. G is regular), then G contains a perfect matching.*

The Corollary 1 guarantees that we can always find a perfect matching in the auxiliary graph $G = (A \dot\cup B, E)$ constructed above, since it is bipartite and every vertex of A is connected to every vertex of B, i.e. all vertices in G have the same degree. The only concern must be how to find a perfect matching with minimum cost.

For any graph it holds that a perfect matching is also maximum since all the vertices of such a graph are matched and that implies the maximal cardinality of the matching. In case of the auxiliary graph $G = (A \dot\cup B, E)$ a maximum matching is always perfect, and we will prove this fact.

Theorem 4. *Let $G = (V, E)$, s.t $V = A \dot\cup B$ be the auxiliary graph from Section 2.5.1, and let $M \subseteq E$ be a maximum matching. Then M is perfect.*

Proof. Suppose M is not perfect. From Definition 2.5 it follows that there exists $v \in V$ which is not matched by M. Without loss of generality suppose $v \in A$.

Let M' be a perfect matching in G, which exists by Corollary 1. Since a perfect matching is also maximum it follows that $|M| = |M'|$. However, in the auxiliary graph $|M'| = |A| = |B|$, but in M at most $|A| - 1$ vertices of A are matched and it follows that $|M| < |M'|$. That implies that M is not maximum and that contradicts our assumption. □

2.5.2 Successive shortest path algorithm

To find a minimum cost maximum matching in the auxiliary graph presented in Section 2.5.1 we can use the algorithm that is described in [2, p. 321] and was implemented in the C++ library "Boost" - Successive shortest path algorithm. The algorithm is based on minimum cost flow computations.

The details of this algorithm are out of the scope of this thesis, so we will take it as a "black box". A curious reader can find the algorithm itself alongside with its analysis in [2]. We will just mention that in our case the time complexity of this algorithm is $\mathcal{O}(n \cdot T)$, where T is the time taken to solve the shortest path problem. For instance, if we use Dijkstra's algorithm described in [9], then this subroutine will have $T = \mathcal{O}(n^2)$ and thus the complexity of the Successive shortest path algorithm is $\mathcal{O}(n^3)$.

As we will see later in Chapter 5, the implementation of this algorithm is impractical, so we will never use it.

2.5.3 Linear program

Let $G = (A \dot\cup B, E)$ be an auxiliary graph defined in Section 2.5.1. We want to find a minimum cost maximum matching in it using linear programming. This problem can be solved in polynomial time since we have that $|A| = |B|$ and if we drop the integrality constraint.

We solve the following linear program (LP):

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} w_e x_e \\ \text{subject to} \quad & \sum_{e \in \delta(v)} x_e = 1, \quad \forall v \in V \\ & x_e \geq 0 \quad \forall e \in E \end{aligned}$$

where for all $e = \{u, v\} \in E$, s.t. $u \in A$ and $v \in B$, $\delta(v)$ is the neighbourhood of the vertex v, $w : E \to \mathbb{R}$ is a weight function such that $w_e =$ *the number of cut edges in the original graph containing* u *as an endpoint* for the vertices of the independent set; $w_e = 1$ otherwise (i.e. for the artificial vertices created to balance the sizes of A and B).

Since the program described above is not integer (it is a relaxation of the integer program), the variables can attain fractional values that do not correspond to any matching in the auxiliary graph. Nevertheless, such a solution will be feasible, and the set of all feasible solutions of the program constitutes a polytope. However, we are interested in matchings only, and those are provided by the optimum solutions of the linear program exclusively, that can be found by optimization over the polytope. The optimum solutions are always at the extreme points of the polytope, and the variables at these points are integral. We refer the reader to [15] where a proof of this fact can be found.

Thus, solving the linear program described above, we guarantee that all vertices of the vertex cover will be assigned to some cluster of the partition of the original graph in the cheapest way possible.

In fact, the theoretical time complexity of this algorithm is not better than the time complexity of the algorithm from Section 2.5.2, i.e. $\mathcal{O}(n^3)$. However, as we will see, it is much better in practice.

3. Genetic Algorithm

3.1 Biological terminology

Since genetic algorithms are inspired by the nature, they use biological terms, but significantly simplify entities. Here, to give the reader some understanding of evolutionary computations, we briefly go through some terms we will use in the thesis [16].

Every living matter or organism consists of a great amount of cells. Each cell has its own set of DNA (deoxyribonucleic acid) strings that together define the whole organism – its traits, features and behavior. These strings are called *chromosomes*.

Chromosomes themselves are not atomic and can be split into *genes* where each gene represents a single trait of the organism. A possible "value" of a single gene is called *allele*. On the other hand, grouping all chromosomes of a single organism we get its *genome*.

In genetic algorithms each organism is called *an individual*. During sexual reproduction two individuals produce *offspring* – new individuals with genomes that are represented by combinations of its parents genomes. This process is called *crossover*.

It may happen that during crossover some of *nucleotides*, atomic elements of DNA, change their allele randomly. This process is called *mutation*.

Each individual has its own *fitness* – a value that basically says how good the organism is, meaning that individuals with higher fitness will reproduce more often than those with low fitness. A rule of fitness assignment is called *a fitness function*.

A group of currently living individuals that are available for reproduction form *a population*. One step of evolution after which a new population is emerged is called *a generation*.

All the organisms in a population are subject to so-called *selection* – a mechanism that chooses individuals from a population for further sexual reproduction.

Finally, selection, crossover and mutation together are called *genetic operators*.

We call the space of all possible individuals with assigned fitness a *fitness landscape*. We can imagine this space as a mathematical surface such that for every point of the surface its "height" is proportional to the fitness of the individual this point represents. Each peak of such a surface is called the *local optimum* with respect to its neighbourhood; a highest peak of the fitness landscape is called a *global optimum*. Thus, when the algorithm evolves a generation, it traverses the fitness landscape. The algorithm finds an optimal solution of a problem when it finds a global optimum on the fitness landscape.

There are several approaches to formation of new generations. One of them is *exploitation*. That means that the algorithm uses individuals that have been already explored so as to evolve better individuals based on them.

The second approach is *exploration*. Basically that means that the algorithm tries to find new individuals on the fitness landscape that have different traits from those individuals in the population.

Out of what has just been said, we can conclude that if a GA uses the first approach only, it gets stuck at some local optimum without any hope to escape it. On the other hand, an algorithm using the second approach only, just blindly traverses the fitness landscape with a tiny probability to eventually hit a global optimum. That leads to a conclusion that a good Genetic Algorithm must be well-balanced in terms of usage of both approaches.

In a Genetic Algorithm crossover is responsible for exploitation. Oppositely, mutation takes care of exploration.

Evolution is an infinite process. However, an algorithm must terminate at some point of its execution. How do we know when to stop? For that purpose we define a *stop condition* that, when occurs, stops the algorithm. In a Genetic Algorithm we can use either a sole stop condition or several ones; for instance, we can stop execution when we have already evolved some number of generations given as an input, or stop when the algorithm cannot evolve a better individual within some given number of generations (i.e. we got stuck at a local optimum).

3.2 The algorithm

The Genetic Algorithm can be viewed as follows:

1. Create an initial population G that has n random individuals in it.
2. Assign to each individual in the initial population its fitness.
3. Create a new empty population P.
4. (Selection) Choose individuals from the population G for sexual reproduction (usually, selection based on the fitness values of the individuals in a population).
5. (Crossover) With probability p_x (*probability of crossover*) exchange some parts of the individuals to produce offspring.
6. (Mutation) In the offspring, with probability p_m (*probability of mutation*) randomly change some of the genes.
7. Assign fitness to the offspring and put individuals chosen from the offspring and their parents to the population P.
8. Repeat steps 4 - 7 until the population P has the size n.
9. Eliminate the population G and let $G = P$.
10. Repeat steps 3 - 9 until a stop condition occurs thus evolving the initial population. Each iteration of steps 3 - 7 of the algorithm makes a new generation.

Thus, the algorithm traverses fitness landscape trying to find an optimum. The illustration of this process is given in Fig.3.1:

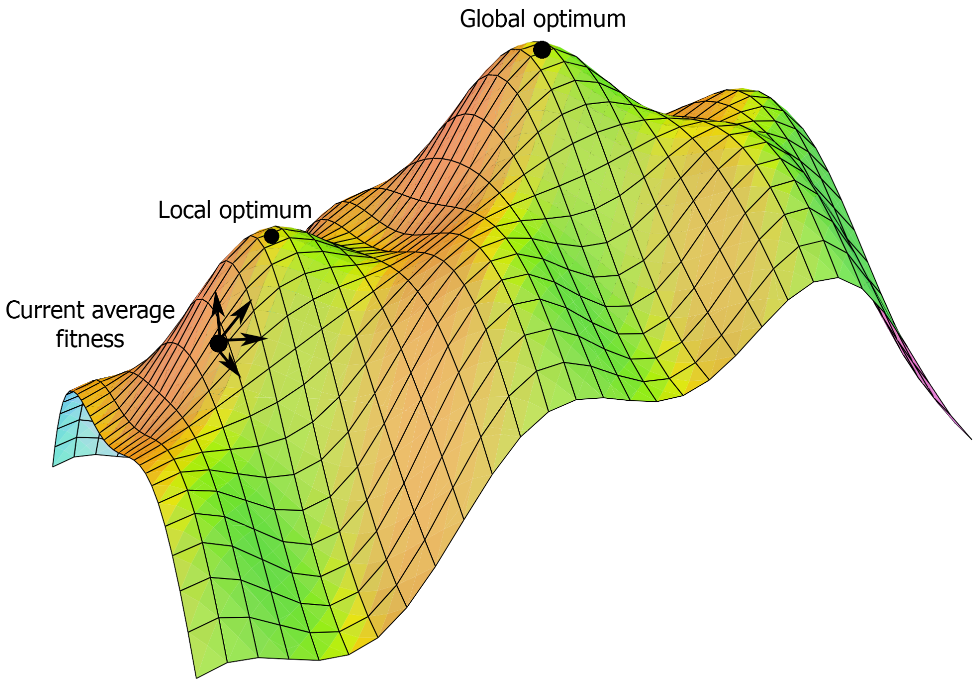


Figure 3.1: Example of a fitness landscape

3.3 Genetic operators

As it can be seen from the description of the GA, the algorithm is not rigorous – it does not prescribe any particular ways to do its steps, and thus we are free to choose its parameters and, moreover, we can omit some of its steps - for instance, the mutation step.

As it was previously mentioned, to find a good setting for the algorithm we must use an empirical approach since different settings have completely different outcomes for each input. That means that the algorithm highly depends on the size of input, probabilities of crossover and mutation, types of mutation, selection, crossover and many other aspects, and that gives us ample opportunity to experiment with all of these parameters.

In order to be able to work with the GA we will require a *framework* (software that provides generic functionality and can be changed by a user to create application-specific software) that can handle all necessary steps of the algorithm. To build the framework, first of all, we need to decide on the representation of individuals. Second of all, we must implement selection for step 3 of the algorithm. Finally, we also need to implement genetic operators such as crossover and mutation.

3.3.1 Graph representation

We will use the same graph representation as described in Section 2.2 because in the adjacency list representation of a graph for each vertex we can easily iterate through the list of its neighbours and thus quickly check correctness of individuals. If some cluster of the solution an individual represents contains more than $\lceil \frac{|V|}{d} \rceil$ vertices such an individual must be discarded.

Also, we still deal with sparse graphs and such a representation is more space efficient in this case.

3.3.2 Fitness function

It is stated in the formulation of the problem that given a partition, we want it to cut as few edges as possible. That means that it is quite natural if the fitness function represents the number of edges that a partition did not cut. Thus, to evaluate fitness, we use a simple formula: $|E| - |C|$, where E is the edge set of G, and $C \subseteq E$ is the set of edges crossed by the partition. This formula completely satisfies the original purpose – the higher fitness, the smaller a cut.

3.3.3 Individual representation

One possible representation of an individual is the following: an individual has the length equal to the number of clusters of a partition of G, i.e. each cell represents a unique cluster of the partition. Each cell of an individual (a cluster) holds a list of vertices it contains. An example is given in Fig.3.2.

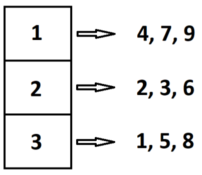


Figure 3.2: First individual representation. In the example, cluster no.1 contains vertices with indices 4, 7, 9, cluster no.2 contains vertices with indices 2, 3, 6 and cluster no.3 contains vertices with indices 1, 5, 8 of the graph G.

Looking ahead, we can notice that, having such representation, it is easy to mutate an individual merely exchanging vertices between two different clusters. However, there is a significant drawback of such a representation: since, after sexual reproduction, we want to get correct offspring, meaning that each offspring satisfies the property that each cluster of a partition has almost the same number of vertices and its size does not exceed a defined limit, it is not obvious how to implement fast crossover that would result in correct offspring each time it is executed.

However, we can develop another possible representation: each individual has the length equal to the number of vertices of the graph G, that is, each "bit" (gene) represents a vertex in the graph, and since every vertex in G has its own index, we can easily define a bijection between the adjacency list of G and such an individual. Thus, each gene of an individual keeps the index of a cluster this vertex belongs to, i.e. if the 4th position of an individual contains number 2 that means that the vertex with the index 4 lies in the cluster no.2 of the current partition. An illustration of this representation can be seen in Fig.3.3.

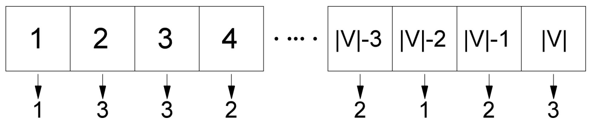


Figure 3.3: Second individual representation. Here the vertex indexed by 1 lies in cluster 1, the vertex indexed by 2 lies in cluster 3 and so on.

In this case, it is easy to check correctness of an individual – we go through the individual and count the number of vertices in each cluster. This operation requires $\Theta(n)$ time, where n is the number of vertices of G, and $\Theta(d)$ space for cluster counting, where d is the size of a partition.

Mutation is also simple – for instance, we can take two different places in an individual and exchange the numbers they hold. As we will reveal very soon, the problem with crossovers encountered earlier disappears as well.

Since the latter representation seems to be more advantageous than the previous one, we will use it in our framework.

3.3.4 Selection

We will implement two types of selection – *Roulette Wheel Selection* and *Tournament Selection*.

3.3.4.1 Roulette Wheel Selection

To introduce Roulette Wheel Selection, we can imagine a Roulette wheel in a casino that has as many sectors as the number of individuals in a single population. The sectors of the wheel have different width, and each sector of the wheel is assigned to an individual in the population in such a way, that the fitter an individual is, the wider its assigned sector is (in other words, the size of a sector is proportional to the fitness value of the individual it represents). For this assignment we divide the fitness of each individual by the total fitness of the whole population, that is, we normalize them to 1. When the wheel is set up we generate a pseudo random floating-point number between 0 and 1, thus choosing some sector (an individual) on the wheel for further reproduction.

With such a selection that is proportional to the fitness it is possible that individuals with small fitness will survive the selection step, since the probability of an individual being selected is different from 0. This allows the algorithm to keep diversity in new populations, and gives it time to explore the fitness landscape making convergence slower.

3.3.4.2 Tournament Selection

In Tournament Selection we choose two individuals in the population uniformly at random, and then, in accordance with the given probability of the best individual to be chosen, we either take the individual with the best fitness as a parent or the individual with the lower fitness. This type of selection is more randomized than the Roulette Wheel Selection and may produce interesting results.

3.3.5 Crossover

Evolution does not occur at all if every time during sexual reproduction we produce completely new individuals with a random set of traits that are not inherited from their parents. Designing a crossover operator one must be careful and think in advance of several crucial properties of this operator:

- The offspring produced after crossover must inherit as many original traits from their parents as possible (i.e. unmutated genes).
- The offspring must be consistent and satisfy all the properties of the individual representation.
- The operation must be as fast as possible.

To achieve these goals we can design the following crossover algorithm:

1. Take two parents for sexual reproduction provided by the selection step.
2. Choose a *cross point* (a point at which we split an individual into two parts) uniformly at random.
3. Split both parents at the cross point and decide which side is shorter (left-hand side or right-hand side).
4. Take the shorter part (if the sides are of equal size pick either of them) of each parent and compute *state vectors* s_1 and s_2 for both pieces, where a state vector is a vector whose length is equal to the size of the partition and each position holds the number of vertices that belong to the cluster of the partition with the corresponding index. Informally speaking, here, if we look at an individual as a partition and "mark" all vertices that lie in the shorter part of the individual after cutting, then for each cluster we count the number of these "marked" vertices in it.
5. Swap the parts from the previous step between the parents.
6. Based on the state vectors from Step 4 compute *difference vectors* d_1 and d_2 for both parents, where the difference vector for the first offspring is $d_1 = s_1 - s_2$ and for the second offspring $d_2 = s_2 - s_1$, and we define vector difference as difference of corresponding elements. Continuing the informal description in Step 4, at this point we just count the number of "marked" vertices that we need to add to each cluster so as to get the initial state of an individual, that is the state of the individual before cutting and swapping parts.
7. For both parents traverse the swapped part and for each gene:
    7.1. Take the value t it holds and let z be the value that is stored at the position t of the difference vector corresponding to the parent.

7.2.
- if $z < 0$: in the difference vector find a position p that holds a positive value, change the value of the gene to p, add 1 to z and subtract 1 from the value stored at the position p in the difference vector;
- if $z > 0$: in the difference vector find a position p that holds a negative value, change the value of the gene to p, subtract 1 from z and add 1 to the value stored at the position p in the difference vector;
- if $z = 0$: continue.

Before we judge the correctness of the algorithm let us prove the following theorem:

Lemma 5. *A difference vector always adds up to 0.*

Proof. Let t be the number of clusters of a graph partition, K and M be two individuals represented as described in Section 3.3.3. We suppose that the algorithm has already found a cross point, split the individuals and made a swap, so now it is at Step 7.

Let A and B be the swapped parts of the individuals K and M. Due to the algorithm we have that $|A| = |B|$.

Let $\mathbf{z} = (z_1, ..., z_t)$ be gotten from the difference of two state vectors - $\mathbf{q} = (q_1, ..., q_t)$ and $\mathbf{p} = (p_1, ..., p_t)$, where $\mathbf{q}$ corresponds to A and $\mathbf{p}$ corresponds to B.

Let us suppose for the sake of contradiction that $\sum\limits_{i=1}^{t} z_i \neq 0$. It follows that $0 \neq \sum\limits_{i=1}^{t} z_i = \sum\limits_{i=1}^{t} (q_i - p_i) = \sum\limits_{i=1}^{t} q_i - \sum\limits_{i=1}^{t} p_i$. Without loss of generality we assume that $\sum\limits_{i=1}^{t} q_i > \sum\limits_{i=1}^{t} p_i$ and that implies that the number of vertices counted in $\mathbf{q}$ is bigger than the number of vertices counted in $\mathbf{p}$. But that means that the number of vertices in A surpasses the number of vertices in B and that contradicts to the fact that $|A| = |B|$. It follows that $\sum\limits_{i=1}^{t} z_i = 0$.

To finish the proof we observe that when the algorithm makes Step 7 it always adds one vertex to the difference vector and subtracts one vertex, so the total contribution is always 0. This concludes the proof. □

Using a difference vector we aim the goal to count the number of vertices we add or remove from each cluster. Step 7 guarantees that, when it is made, the resulting distribution of vertices is the same as it was before this step, in terms of the number of vertices in each cluster. That is it only swaps vertices between clusters and never decreases or increases their number in any of them.

Thus we can see that when the algorithm finishes it produces a pair of correct individuals, since every cluster satisfies the constraint on the maximum number of vertices in it. Step 7 also ensures that we preserve as many genes at their original places as we can, since it never reorders the genes of an offspring; the time complexity of this process is $\mathcal{O}(n)$.

To speed up searching for positive (negative) values in a difference vector we can employ a stack, storing indices of positions that hold a positive (negative) value in amount equal to the magnitude of this value (of the absolute value). That is, such a stack will hold indices of partition clusters that must get additional vertices (eliminate extra vertices) in order to be consistent. We fill the stack in at Step 6 of the algorithm and never rebuild it. Also, to organize a random change for each incorrect gene, we fill the stack in using a random order, i.e. filling it with a random permutation of absent (extra) indices of partition clusters.

To better understand this algorithm, consider an example in Fig.3.4.

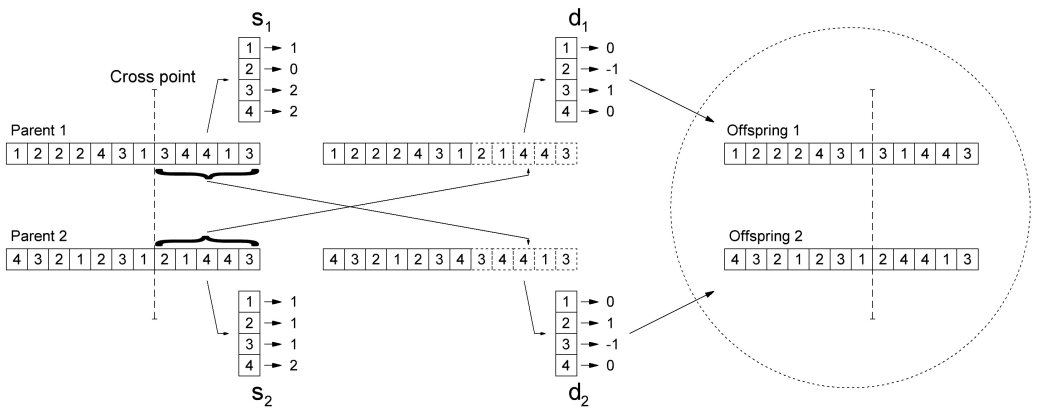


Figure 3.4: Crossover

In this example, the algorithm chooses the right-hand side parts of the parents, and computes the state vectors of these parts - s_1 and s_2. For instance, for the right-hand side of Parent 1 we have that two of its vertices lie in the cluster with index no.3, so the corresponding state vector s_1 holds value 2 at the 3^{rd} position. Then, the algorithm exchanges the right-hand side parts of the parents and produces two difference vectors - d_1 and d_2. As we can see, some of values of a difference vector may be negative. Thus, in this example, we know that for the second offspring we have to remove one vertex that lies in the cluster with index no.3 and put it into the cluster with index no.2. As we can easily check, both of the offspring satisfy the individual structure.

3.3.6 Mutation

There are many ways to mutate an individual, and we will create a few types of mutations. However, not all of them will be applicable since some operators can lead to unnecessarily high variance in a population.

3.3.6.1 Random mutation

For this type of mutation we, uniformly at random, choose two genes and exchange values they hold between them. In the scope of the original problem that means that we exchange two vertices of the input graph between the clusters they belong to.

3.3.6.2 Uniform mutation

In Uniform mutation for each gene we decide whether we want to mutate it or not. If yes, then we find another gene in the individual different from the one previously chosen, and exchange the values between these genes. For this type of mutation we need an additional parameter - *probability of bit-mutation.*

3.3.6.3 Shift mutation

Here we shift genes of an individual to some randomly chosen distance. This mutation most probably will be impractical due to huge changes in genome it causes.

3.3.6.4 Big bang mutation

Another extreme type of mutation - we completely ruin the genome structure of an individual and replace it with a new, randomly generated genome. In the view of the original problem that means that we create a new random partition and use it for evolution purposes. As in the previous case, we expect this operator to be highly impractical for the same reason.

3.3.7 GA strategies

After the algorithm produces offspring, it needs to define which individuals will be transferred to the next generation.

We suppose that the algorithm went through the selection step and chose two different parents. The parents reproduced two children, each with its genome inherited from the parents and created during crossover. After that, the children's genomes mutated and the algorithm assigned some fitness values to them. Now we have a set of 4 different individuals (the parents and the offspring), and we need to decide on a general strategy of the algorithm, which can transfer to the next generation:

1. The children (offspring) only (*"Kill parents"* strategy);
2. Two fittest individuals out of this set of four (*"Best Only"* strategy);
3. The whole set of individuals;
4. Any subset of the set of individuals, that is not covered by the previous options.

Usually, in practice, the first two strategies are used. However, as we can deduce, if we choose the "Kill parents" strategy it may happen, that we will lose the fittest individuals from the currently considered population, owing to randomness of the selectors (the fittest individuals will not be chosen by the selectors for sexual reproduction during the whole evolution step). To prevent this, we can also force the algorithm to transmit part of the fittest individuals from the current population to the next one unchanged by crossover and mutation. Here we decide on a proportion of the fittest individuals we want to transmit, for instance 5%. That means that if a population has size 200, every generation will get 10 fittest individuals from the previous generation unchanged.

3.3.8 Re-evaluation of fitness

Following the aim to implement as time efficient GA as possible we should think of possible ways to speed it up.

We can notice, that after crossover and mutation offspring do not inherit fitness from their parents. That implies that we have to evaluate fitness for all offspring the algorithm produces during its execution.

We have already defined the fitness function - it counts the number of edges a graph partition, represented as an individual, preserves. That means that, if we take it as it is, in order to evaluate fitness, for each offspring we have to map a partition on the given graph and traverse it, counting the number of edges whose endpoints belong to the same cluster of the partition. Unfortunately, that leads to discouraging time complexity $\Theta(n^2)$ that significantly slows down the algorithm.

However, for the crossover that we invented earlier we can employ the fact that the structure of an offspring differs from the structure of its *dominant parent* (a parent that passes more of its genes to an offspring than the second one) in a slight manner, since it preserves most of the dominant parent genes. Thus, for the fitness value of an offspring we can use the dominant parent's fitness value as a basis, and then, traversing the genes inherited from the second parent, we check how the structure of these genes differs from the original ones, every time increasing or decreasing fitness value according to those changes. The same idea may be used for the Random mutation - the only thing we need to do is to consider two mutated genes and adjust the original fitness value of an individual according to the changes they made in the structure of the original partition.

In case of other mutation types, we cannot do better than to traverse the whole graph over and over, and that makes these operators rather slow.

3.4 Time complexity

Despite the fact that the GA does not have any certain bound on its running time, we still can evaluate time needed to make a new generation.

- **Initial step:** we create a population of size m with individuals of length n. Each individual exploits random assignment of values to its genes, that implies that the time complexity is $\Theta(n)$ for a full individual assignment that in total gives us $\Theta(mn)$ time complexity for a whole population;

- **Selection:**

  1. **Roulette Wheel Selector:** we compute the sum of all fitness values in time $\Theta(m)$ and split the wheel into sectors in time $\Theta(m)$. Since to find a "winner" we use binary search and the sequence of sectors is sorted by default, it gives us $\mathcal{O}(\log m)$ time complexity. In total, for this step we have $\Theta(2m + \log m)$ time complexity.
  2. **Tournament Selector:** this type of selection is the fastest and gives us a constant time complexity $\Theta(1)$, owing to random selection of two individuals and returning one of them.

- **Crossover:** Crossover, as it was discussed in Section 3.3.5, has time complexity $\mathcal{O}(n)$.
- **Mutation:**
  1. Random mutation has a constant time complexity $\Theta(1)$.
  2. Uniform mutation is bounded by $\Theta(n)$.
  3. Big Bang mutation is performed within $\Theta(n)$ time.
  4. Shift mutation time complexity is also bounded by $\Theta(n)$.
- Both, Selection and Mutation steps get an additional factor of $\mathcal{O}(t \cdot n)$ for fitness re-evaluation, where t is the number of genes changed at the respective step.

4. Description of the framework

4.1 Structure

To build a framework we need to follow the next steps:

1. First of all, we need to think of graph representation and implement a data structure for graphs and tools for them. We define a class ”Data” which will contain all information about a graph.
2. To get an input graph we use either a graph generator or we read it from a file. Thus we will need to implement both of them. For those purposes we define two classes – ”Reader” and ”Generator”.
3. FPT algorithm:
    (a) Create a function which produces a vertex cover of a given graph using the FPT algorithm. Thus, we will introduce a class ”VertexCover”.
    (b) Create a function which produces restricted growth strings in lexicographic order to go through all possible partitions of a given graph. For that purpose we will use class ”Partition”.
    (c) Create a function which computes a minimum cost maximum matching in a given bipartite graph. We will use 3rd party software – library ”Boost” which contains the algorithm mentioned in Section 2.5.2.
    (d) Create a function which solves the linear program for the Minimum Weight Perfect Matching problem in a bipartite graph, which we described earlier in Section 2.5.3. We will compare its effectiveness with the function from the previous item. For the function we will use the CPLEX library (IBM’s LP solver) written in C.
4. Genetic algorithm:
    (a) Create classes for individuals and populations.
    (b) Create a virtual class for selectors; implement Roulette Wheel and Tournament selectors.
    (c) Create virtual classes for mutation and crossover operations; implement the genetic operators discussed in Section 3.3.
    (d) Create a class that implements the GA discussed in Section 3.2.

4.2 Graph representation

As was discussed in Section 2.2 and Section 3.3.1, we will use adjacency list for graph representation. An input graph $G = (V, E)$ will be given as a vector of size $|V|$ with a linked list for each vertex. In C++ the data structure looks as follows:

```
std::vector<std::list<size_t>> graph.
```

We would like to organize reading of a graph from a file. So, first of all, we will need a function which reads a file and stores its data into an adjacency list. Also, each file contains a parameter d – number of partitions of a graph.

4.3 Input file structure

Let the structure of a graph file be as the following:

- 1^{st} line – parameter d
- 2^{nd} line – the cardinality of the vertex set (number of vertices)
- 3^{rd} – nth lines – edges of the graph

All the vertices of a graph must have a positive index.
Here is an example of an input file:

```
5
20
1 2
3 6
5 9
5 12
5 18
6 20
```

This can be interpreted as a graph with 20 vertices and 6 edges in which we want to partition the vertex set into 5 clusters.

4.4 Classes

4.4.1 Class ”Data”

The class is used to keep data of a graph such as its size (the vertex set size), vertex cover, independent set, number of partitions, maximum number of vertices in each part. When an instance of the class has been created it contains the graph in a form which ”Boost” library can accept as its input. The class also contains a function for storing the graph as a ”GraphViz” file with ”.dot” extension to easily visualize it in GraphViz program.

4.4.2 Class ”Reader”

When a user wants to read data from a file then an instance of the class ”Reader” is created. The class contains fields for a graph itself, number of partitions and number of vertices in the graph. Also, it has a set of getters to retrieve data from outside functions.

4.4.3 Class "Generator"

An instance of the class is created when the user decides to generate a graph and the class simply generates a random graph using input data such as:

- Number of vertices in a graph
- Size of a smallest vertex cover
- Probability of an edge presence

The constructor of the class calls the function "generate" which randomly adds edges into the vertex cover and then edges between the vertices of the independent set and the vertex cover. The generator creates two files in a file system:

- File "Generated_N.txt" - the generated graph itself;
- File "Generated_N_VC_IS.txt" - a smallest vertex cover and the corresponding independent set,

where N is a first free index in the file system (starts from 0). If we start either of the algorithms immediately after a graph has been generated, the algorithms work with this graph. Otherwise, if we want to work with the generated graph after we restart the program, we need to rename the files "Generated_N.txt" and "Generated_N_VC_IS.txt" into "Input.txt" and "Input_VC_IS.txt" respectively.

4.4.4 Class "VertexCover"

The class is used for providing a vertex cover for a given graph. It is an implementation of the algorithm described in Section 2.3.

4.4.5 Class "Partition"

Class "Partition" is used for the purpose to provide the program with restricted growth strings in lexicographic order using the algorithm described in Section 2.4. Those strings are used to partition the vertex cover of a given graph and ensure that we do not use a partition isomorphic to a previously used one.

4.4.6 Class "FPT"

Class "FPT" contains an implementation of the FPT algorithm. When the method "startFPT" is called, the program asks the user whether they want to use the LP solver or not. In either case the program generates a partition of the vertex cover for a given graph, and then it assigns the vertices of the independent set to the sets of the partition in the cheapest way possible.

If the user chooses not to use the LP solver, then the program uses the minimum cost maximum flow algorithm of the "Boost" library, which was described in Section 2.5.2. Otherwise, it solves the LP that was described in Section 2.5.3. After that, the program calculates the total cost of the iteration and compares it with the current best partition, storing the current partition in case it is better than the previous best solution.

After going through all possible partitions of the vertex cover the program creates the file ”Graph.doc” in the working directory which can be opened with GraphViz program. Also, it prints out the best partition alongside with its cost.

4.4.7 Library ”Evolution.h”

This library contains an implementation of the GA with all of its genetic operators.

The library relies on object-oriented programming (OOP) and extensively uses polymorphism that allows us to keep the code short. It contains interfaces for all genetic operators, several realizations of the operators discussed previously, and a class for the GA itself. This class provides the user with an input dialog that allows them to adjust the algorithm choosing genetic operators and their parameters.

The Roulette Wheel Selector uses sequential access to the individuals in a population that exploits an iterator and provides us with the fastest access possible. Also, to find the ”winner” on the Roulette wheel we employ binary search algorithm, since the whole wheel can be seen as a scale from 0.0 to 1.0 and thus all the values of the scale are naturally ordered, i.e. we have a sorted sequence of floating-point numbers.

The implementation of the crossover operator uses a stack, as it was mentioned earlier in Section 3.3.8. To provide random order of its elements we randomly permute the values it holds.

4.4.8 Software Project.cpp

File ”Software Project.cpp” contains the function ”main”. First of all, it gets an input. Then it checks whether the number of partitions is equal to the number of vertices. If so, it prints out ”You are separating all vertices, thus the minimum cost is equal to the number of edges in the graph” and finishes. Otherwise, if the graph does not contain any edges at all, the program returns any random partition. If the vertex cover contains an edge, the program asks the user whether they want to start the GA algorithm and/or the FPT algorithm.

4.5 Logging

In order to collect data for further analysis we need to register information about an optimal solution found and the time taken by the FPT algorithm, and also record information such that the best fitness value in a population, the average fitness value within a population, the cost of the best solution of the original problem found so far and the time spent by a generation in case of the GA. We define the following log file patterns:

- FPT log:

$$\text{OPT} = x$$
$$\text{Time: } t \text{ s.}$$

where x is the cost of an optimal solution, t - time the FPT algorithm took, in seconds.

- GA log:

Header

Generation p:
$->$ Best fitness: b
$->$ The average fitness: f
$->$ Cost of best solution: c
Time: t s.

...
—— RESULTS OF EVA ——
Fitness of the best individual found: F
The best partition found: $\mathbf{v}$
Cost of the solution: C

END OF EVA

Time taken by EVA: T s.

where p is the index of a current generation, b - the highest fitness value among the individuals present in the current population, f - the average fitness value in the current population, c - the cost of the best solution of the original problem, t - time taken by the algorithm so far, F - fitness value or the best individual found by the algorithm, $\mathbf{v}$ - a vector representing the best partition found, C - the cost of the best solution of the original problem, T - total running time taken by the GA and, finally, the header section is a summary of the GA settings.

4.6 Parsing of log files

To have a convenient form of log files to be used further for plotting, we will use a parser written in C#. It reads a log file of the GA and produces another file of the form:

$$p \mid b \mid f \mid c \mid t$$

where all variables have the same meaning as was described in the previous section.

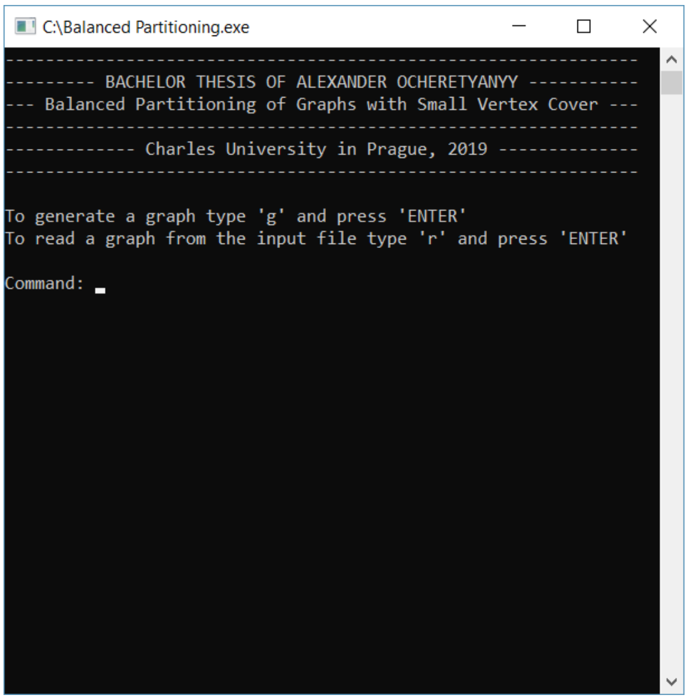
4.7 Automation

To automate the experiments we will use scripts written for PowerShell - a task-based command-line shell equipped with a scripting language. It will allow us to not start the program and the parser manually and efficiently use CPU time.

4.8 Program usage

To work with the program we must use a 64-bit version of Windows OS and start "Balanced Partitioning.exe".

After startup, the program outputs a dialog shown in Fig.4.1.



C:\Balanced Partitioning.exe

```
----------------------------------------------------------------
--------- BACHELOR THESIS OF ALEXANDER OCHERETYANYY -----------
--- Balanced Partitioning of Graphs with Small Vertex Cover ---
----------------------------------------------------------------
------------- Charles University in Prague, 2019 --------------
----------------------------------------------------------------

To generate a graph type 'g' and press 'ENTER'
To read a graph from the input file type 'r' and press 'ENTER'

Command:
```

Figure 4.1: Input dialog of the program

We can put the key 'g' in case we want to generate a graph. The program will ask:

- Number of vertices – a positive integer;
- Size of a vertex cover – a positive integer, smaller or equal to the number of vertices;
- Probability of an edge – a real number between 0 and 1;
- Number of partitions – a positive integer smaller or equal to the number of vertices.

In case we want to read data from a file, we must put the file into the working directory and call it "Input.txt". The file must have the structure described in

Section 4.3. We choose the key 'r' for reading the file and follow the steps of the dialog to work with the algorithms.

After the program finishes the FPT algorithm, we can find the result in the working directory in the file "Graph.dot" which can be opened with GraphViz program; one such a graph is shown in Fig.4.2.

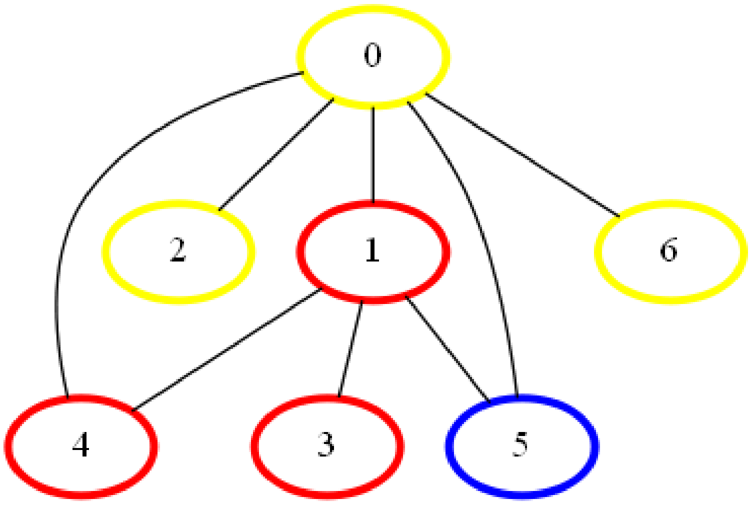


Figure 4.2: A partition with the cost 4 found by the FPT algorithm.

5. Experiments

5.1 Hardware domain

The aims of the experiments, as was previously mentioned, are:

1. To test the FPT algorithm and, for a given hardware domain, decide what is considered to be a small vertex cover.
2. To test the Genetic Algorithm and, if possible, to find an optimal setting for the Balanced Partitioning problem.
3. To compare the running time of both algorithms.

The last aim requires the experiments to be consistent in terms of an environment they are performed in, so we will use a single computer with the following characteristics:

- Processor: Intel(R) Core(TM) i7-8750 CPU @ 2.20GHz 2.21 GHz.
- Memory: 16 GB RAM.
- Operating System: Windows 10 Pro, version 1803, build 17134.765.
- System type: 64 bit operating system, x64-based processor.

5.2 Settings of the Genetic Algorithm

Evidently, to make the algorithm work reasonably well we need to find good balance between exploitation and exploration, and thus our experiments will be aimed towards finding good settings of the algorithm where the algorithm with "good settings" possesses some of the following features:

1. Population diversity (a population has a wide range of individuals with different traits).
2. Populations converge (the algorithm uses exploitation often enough), the algorithm finds an optimum (local or global).
3. The algorithm has enough time for exploration (not too rapid convergence).
4. It finds the global optimum quickly.

Before experimenting with parameters of the algorithm we can make several observations.

First note we can make is that the last feature is hardly achievable in general, since we deal with an NP-hard problem. At the same time, we can achieve the 3^{rd} feature giving the algorithm enough time to evolve an initial population.

When we transfer best individuals out of the current population to a new one without any modifications, we limit exploration since there are less "free slots" in the new population for mutated individuals and it causes inbreeding. That leads to faster convergence of the average fitness but, as a result, we easily get stuck at a local optimum due to degeneracy of the whole population to the local best individual. The more best individuals we transfer, the faster convergence we have.

The Roulette Wheel selection will provide us with individuals having fitness ”above average” more often than in case of the Tournament selection, and that leads to faster convergence and more extensive exploitation in the first case, than in the second one. The Tournament selection, conversely, takes an advantage of exploration and, with some settings (where we set up a low probability of the best individual, out of two picked uniformly at random, to be selected), may even diverge.

Also, the more homogeneous population is, the less variety of individuals we get. This leads to a conclusion that, again, we will faster end up at a local optimum with no chances to escape. Why? Because usually the probability of mutation is quite low in comparison with the probability of crossover. That means that we usually get offspring having the same traits as their parents do, without any modifications. We do not add to the population any new interesting individuals that could lead to some other set of alleles, but instead we try to evolve the best individual from only those that we have. Again, that means inbreeding.

We immediately can filter out some of genetic operators as impractical; the most intuitive choice is the Big Bang and Shift mutation, because those operators completely destroy the inner structure (genome) of an individual. Indeed, as we can see in Fig.5.1 for a random graph with the optimal fitness value equal to 152, they do not allow the algorithm to approach the optimal fitness value and the algorithm gets stuck at a local optimum due to crossover only.

We will not use Uniform mutation either - it is just a generalization of Random mutation.

Since we have only one machine to run the tests on, we should choose some settings of the GA that we want to compare. We will split the whole ”experiment field” into branching levels. Thus, we will distinguish possible inputs of the algorithm according to the following parameters:

1. Graph size: 50, 100, 200 vertices.
2. Size of the vertex cover and probability of an edge: 30 vertices and probability 0.7, 30 vertices and probability 0.3, 10 vertices and probability 0.7, 10 vertices and probability 0.3.
3. Size of a partition: 10, 5, 2 clusters.
4. Selector type: Roulette Wheel Selector, Tournament Selector.
5. Probability of the best individual to be chosen (for the Tournament Selector): 0.9.
6. Probability of crossover: 0.9, 0.8.
7. Probability of mutation: 0.4, 0.2.
8. Population size: 200, 100, 60 individuals.

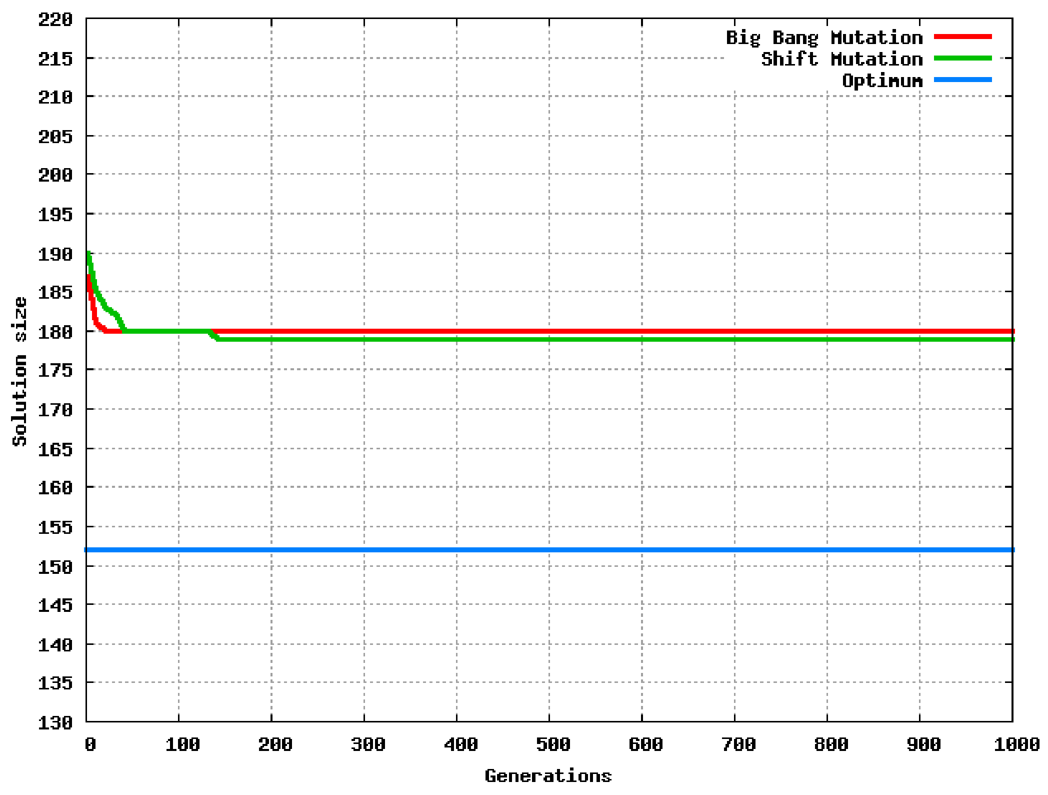


Figure 5.1: Big bang and Shift mutations.

9. Strategy:
    - Kill parents and transfer 10% of the best individuals to the next generation.
    - Kill parents and transfer 5% of the best individuals to the next generation.
    - Kill parents with no transfer of individuals from the original population to the next generation.
    - Best only and transfer 10% of the best individuals to the next generation.
    - Best only and transfer 5% of the best individuals to the next generation.
    - Best only and no transfer of individuals from the original population to the next generation.

In either case we will produce 1000 generations to give the GA enough time for evolution.

The choice of such settings is not random and is based on the author's experience with usage of a Genetic Algorithm for solving similar type of problem - the Bin Packing problem, which is described in [24]. The reason for such settings is the following - the probability of crossover should be high in order to force frequent sexual reproduction which leads to exploitation and allows the algorithm to find a local optimum (that, of course, can be a global optimum). On the other hand, we do not want to mutate offspring too often since in the case of frequent mutation we lose the traits inherited from parents and it leads to blind traversal of the fitness landscape. However, we still should use mutation often enough to allow the algorithm to find new species with exotic, with respect to the original population, genes that could possibly lead to another local optimum in the search field which might be a global one. Finally, the diversity of the GA settings will allow us to find some reasonable balance between exploitation and exploration.

For the Tournament Selector we set up very high probability (0.9) of the best individual to be chosen and we aim two goals - we want the average fitness among all generations to converge and we also want it to converge relatively quickly. If we set the probability less than 0.5 then the selector will be choosing worse individuals for sexual reproduction most of the time and thus we do not achieve convergence of the average fitness. On the other hand, if we set this probability to be slightly above 0.5 then it may happen (although, with small probability) that the selector still will be choosing worse individuals for sexual reproduction, so to be sure that it will not happen (or, more precisely, will almost never happen) we set up such a high probability.

The choice of the graph sizes is not random. Since we want to compare the GA with the FPT algorithm we want the FPT algorithm to find optimal solutions in reasonable time. At the same time we do not want to consider graphs with just a few vertices, because in real applications we usually encounter graphs with hundreds, thousands, millions and even billions of vertices (VLSI, for instance). Thus the choice of graphs with 50 vertices as the smallest instances is reasonable and then we double it twice, getting graphs with 100 and 200 vertices. Bigger graph sizes are difficult to deal with - quick experiments with the program can easily prove it.

The same logic is used for the choice of the vertex cover sizes. We start with the size 2 and then we use slightly more than twice of this size - 5 vertices which we then double thus getting 10 vertices. However, initial tests performed by the author showed that for the FPT algorithm even the size of 10 vertices is difficult to work with. At the same time, it is not a problem for the GA - the size of the vertex cover does not play any role for the GA owing to its nature.

The final experiment tree is presented in Fig.5.2.

As we can see from the picture, graphs emerge on the selector level at which we will also use the FPT algorithm, where the FPT algorithm requires reasonable computation time. For graph construction we will employ the graph generator which we described earlier.

As it follows from the tree, we will perform about 5,184 experiments for the GA, in total producing 5,184,000 generations.

5.3 Predictions

As it was stated in Chapter 1, the FPT algorithm always produces an optimal solution, and thus we expect the FPT implementation to produce correct and optimal results always. We will also use the results of the FPT algorithm to compare them with the GA's results so as to evaluate precision of the GA.

We also expect that the LP part described in Section 2.5.3 of the FPT algorithm will be much more time efficient than Successive shortest path algorithm described in Section 2.5.2.

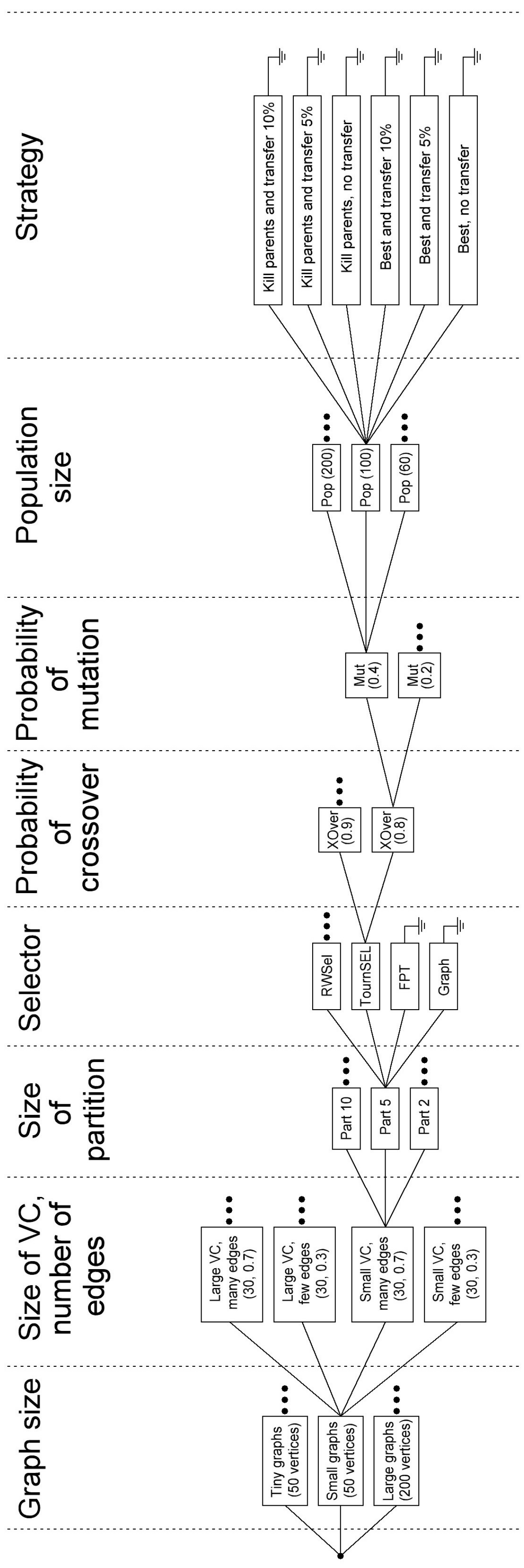

Graph size
Size of VC, number of edges
Size of partition
Selector
Probability of crossover
Probability of mutation
Population size
Strategy
Tiny graphs (50 vertices)
Small graphs (50 vertices)
Large graphs (200 vertices)
Large VC, many edges (30, 0.7)
Large VC, few edges (30, 0.3)
Small VC, many edges (30, 0.7)
Small VC, few edges (30, 0.3)
Part 10
Part 5
Part 2
RWSel
TournSEL
FPT
Graph
XOver (0.9)
XOver (0.8)
Mut (0.4)
Mut (0.2)
Pop (200)
Pop (100)
Pop (60)
Kill parents and transfer 10%
Kill parents and transfer 5%
Kill parents, no transfer
Best and transfer 10%
Best and transfer 5%
Best, no transfer

Figure 5.2: Experiment tree. The dots indicate that an element has an identical subtree as another element at the same layer presented on the picture.

Speaking of the GA, in general, we expect the Tournament Selector to work faster than the Roulette Wheel Selection due to its simplicity.

As for the strategies, if we choose the *"Kill parents"* strategy without taking some part of best individuals from previous generations, and if we use the Roulette Wheel Selector with this strategy, the algorithm will blindly search for feasible solutions (individuals), traversing the fitness landscape not following any specific direction. This happens because with this approach, for example, it may happen that all the offspring we produced are worse than their parents and thus we do not evolve population but deteriorate it. Also, it may happen that the selector never chooses best individuals for sexual reproduction and thus they will be lost. Oppositely, always taking some part of best individuals out of previous generations we increase their chances to be chosen for further reproduction. Thus, transferring some part of best individuals out of the previous population to a new population we increase their chances to be chosen at some generation for reproduction. Definitely, the more best individuals we transfer, the more chances for reproduction they have. That is why we do not expect to see blind traversal of the fitness landscape when we use the Tournament Selector - it uses best individuals out of a sample with a given probability and we never set up this probability to be equal to 0.

Since new generations are built by crossovers and mutations of already explored individuals, we do not expect any significant "jumps" across the fitness landscape, i.e. plots for such setting will look like a random function within some narrow (with respect to the whole fitness landscape) corridor whose average value depends on the initial population.

For the strategy *"Best only"* with keeping 10% of the best individuals from previous generation we expect it to converge quickly with a high probability of not finding an optimal solution, i.e. such setting should lead to some local optimum. However, having mutations, it can sometimes find an optimal solution, although we expect such an event be rather accidental than expected. Nevertheless, such setting should find its application in real life when we want to find some more or less optimal solution (local optimum) within a short period of time. To get better results, intuitively speaking, we could increase the probability of mutations to some high value to allow the algorithm to explore the fitness landscape and use other mutation types (e.g. uniform mutation) to widen the search range.

Having said that, the same setting with keeping only 5% of the best individuals gives a bit more time to the algorithm for exploration and such a setting should find a global optimum more often than the previous one, however in this case we, of course, sacrifice time.

The last setting that exploits the *"Best only"* strategy without keeping the best individuals from previous generation is a classical one; we expect it to show better result than the previous two since it has enough time to explore the fitness landscape and also more time for mutations that can eventually lead to a global optimum.

5.4 FPT algorithm

Before we start, let us notice that for the experiments that will follow in this section we never execute the algorithm described in Section 2.3, since for every generated graph we are given a smallest vertex cover in advance (see Section 4.4.3 for details).

5.4.1 Matching algorithms

First of all, let us compare the speeds of two approaches to the FPT algorithm, mentioned in Section 2.5 - solving an LP or finding a minimum cost maximum flow in a bipartite graph.

We run the algorithm on the same instance of an input graph that has 50 vertices, size of a vertex cover is 5, it has 130 edges and the number of partitions is 5. The running time of the FPT algorithm:

- Max-flow computation: 59.143 s.
- LP: 1.203 s.

For a slightly different instance of an input graph where the number of edges is 180, and the number of partitions is 10, we get the following result:

- Max-flow computation: 57.782 s.
- LP: 1.487 s.

For a bigger instance that has 100 vertices, size of a vertex cover is 5, it has 331 edges and the number of partitions equals to 10, the difference between running times is dramatic:

- Max-flow computation: 446.822 s.
- LP: 7.153 s.

That means that there are no reasons to use max-flow computation in further experiments; we will use the fastest approach among those two - LP.

However, as monitoring of running by the operating system processes showed, the FPT algorithm with usage of LP is highly space consuming. For graphs with 500 vertices, in average case it uses around 5 GB RAM, while 1000 vertices graphs crash the program due to lack of memory needed (the program crashes near 10 GB RAM of used memory and probably does not use the whole available space due to some inner restrictions of the operating system).

5.4.2 Partition size influence

For this test we generate a graph with 100 vertices. The graph has 336 edges, the vertex cover has size 5. We consequently set up the partition size to the values in the range from 2 to 99 to check whether the size of a partition influences running time of the algorithm. For each partition size we run the program 5 times.

The result of the experiment is given in Fig.5.3.

Despite the fact the plot appears to be unpredictable, we can explain why it behaves in this manner.

Let us consider the series of samples at Fig.5.3. For the series we create a table that shows us the following data:

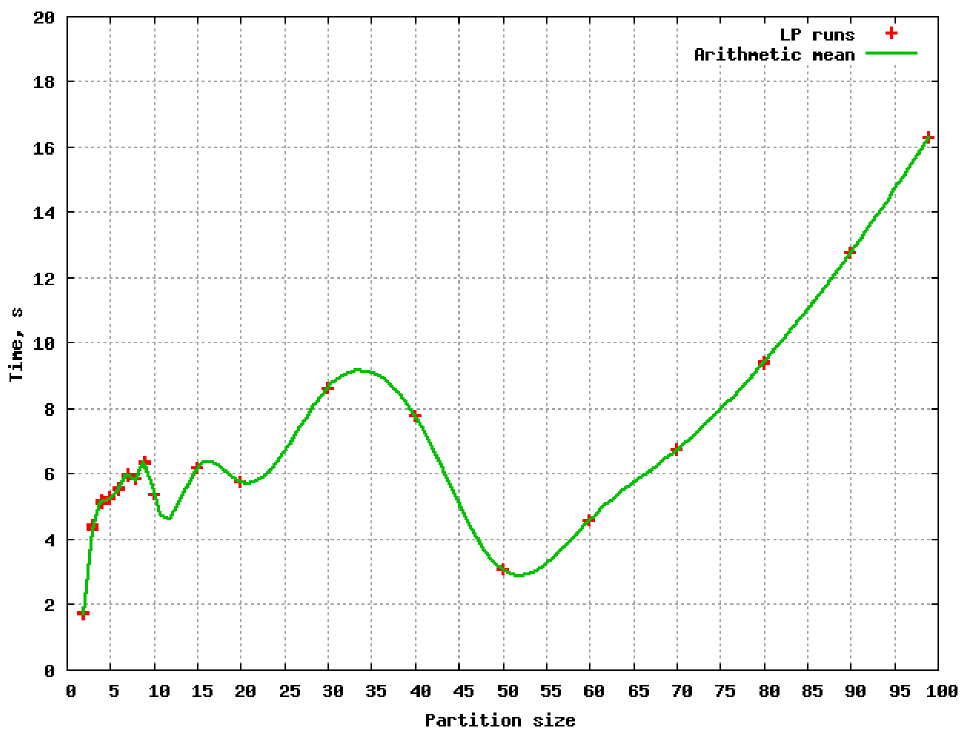


Figure 5.3: Partition size influence.

1. Partition size;
2. Cluster limit - the maximum number of vertices each cluster of a partition can have, i.e. $\lceil\frac{|V|}{d}\rceil$, where $|V|$ is the size of the vertex set of the input graph and d is the partition size;
3. Total partition capacity - the maximum number of vertices all clusters of a partition can keep simultaneously, i.e. $(\lceil\frac{|V|}{d}\rceil \cdot d)$;
4. Number of occupied places - the number of places that are already occupied by the vertices in the vertex cover right before the moment we create an auxiliary graph (see Section 2.5.1), which in our case always equals 5, i.e. the size of the vertex cover;
5. Size of the independent set - in our case it is always equal to 95, i.e. $(|V| - \textit{the size of the vertex cover})$;
6. Number of vacant places - see Section 2.5.1 where we gave the definition of a vacant place;
7. Number of artificial vertices - see Section 2.5.1 for the definition of an artificial vertex.

The data is given in Fig.5.4.

Partition size	Cluster limit	Total partition capacity	Number of occupied places	Size of the independent set	Number of vacant places	Number of artificial vertices
2	50	100	5	95	95	0
3	34	102	5	95	97	2
4	25	100	5	95	95	0
5	20	100	5	95	95	0
6	17	102	5	95	97	2
7	15	105	5	95	100	5
8	13	104	5	95	99	4
9	12	108	5	95	103	8
10	10	100	5	95	95	0
15	7	105	5	95	100	5
20	5	100	5	95	95	0
30	4	120	5	95	115	20
40	3	120	5	95	115	20
50	2	100	5	95	95	0
60	2	120	5	95	115	20
70	2	140	5	95	135	40
80	2	160	5	95	155	60
90	2	180	5	95	175	80
99	2	198	5	95	193	98

Figure 5.4: Data for partition sizes in the range between 2 and 99.

The dependency of the number of artificial vertices on the partition size is given in Fig.5.5.

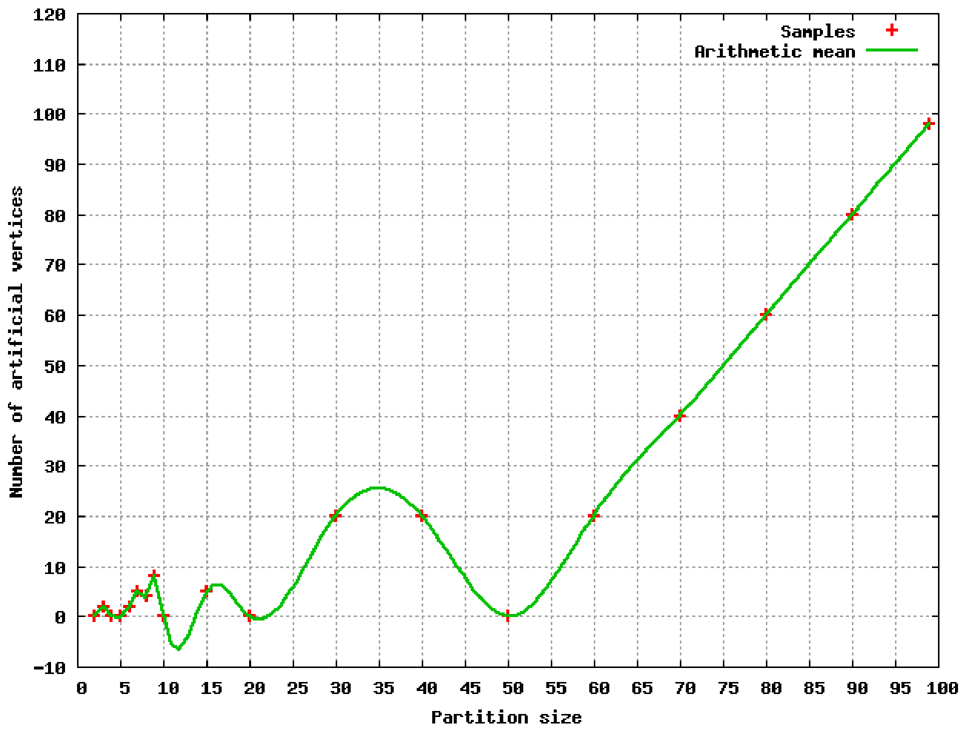


Figure 5.5: Dependency of the number of artificial vertices on the partition size.

If we now compare Fig.5.3 and Fig.5.5 then we will see that both plots have the same envelope. This demonstrates why the behavior of the plot in Fig.5.3 is so peculiar. In fact, the size of a partition influences the running time of the FPT algorithm in such a way, that it significantly grows when the number of clusters approaches the number of vertices in the vertex set of the input graph. Thus, the running time of the FPT algorithm highly depends on the number of artificial vertices it creates during its execution.

5.4.3 Influence of the vertex cover size

In Fig.5.6, Fig.5.7 and Fig.5.8 we can see the running time of the FPT algorithm for vertex covers of different sizes in graphs with 50, 100 and 200 vertices respectively and the average number of edges (we used probability of an edge = 0.5 in the generator). For each vertex cover size we ran 5 consecutive test.

The dependency is quadratic; according to the data used for the plot, each increase of the vertex cover size by 1 requires approximately 4 times more of computational time than without this vertex.

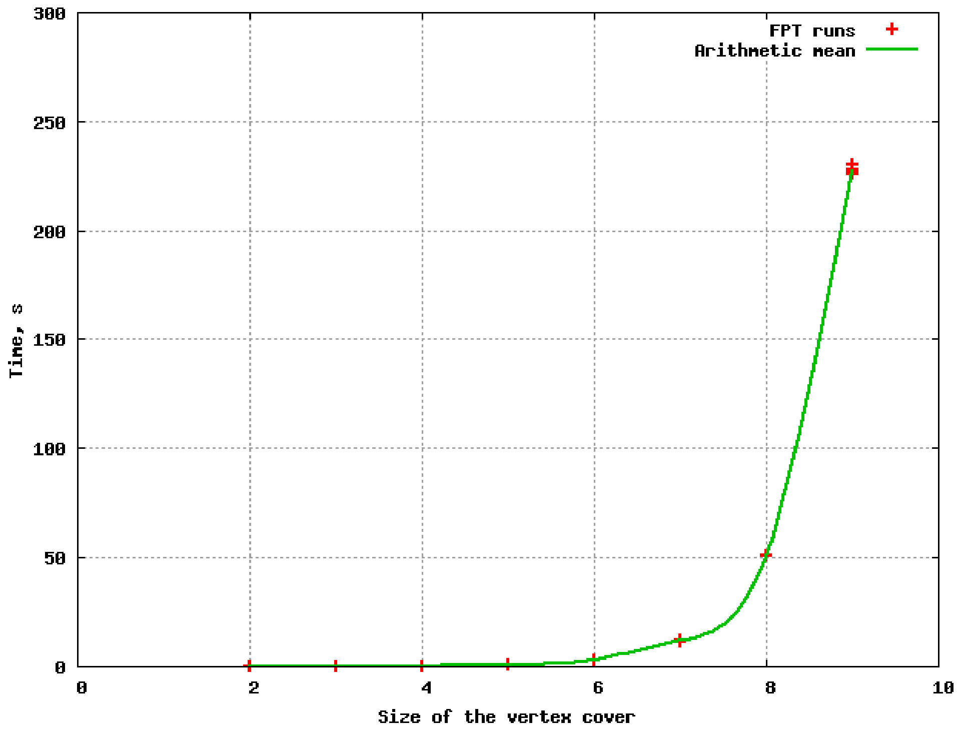


Figure 5.6: FPT running time for a graph with 50 vertices.

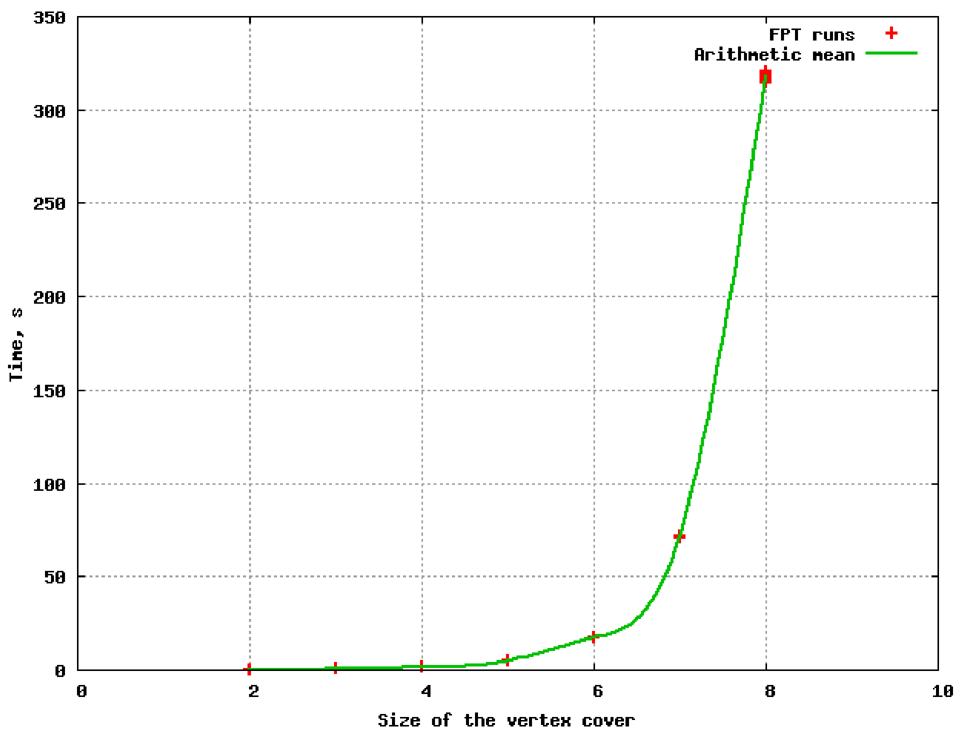


Figure 5.7: FPT running time for a graph with 100 vertices.

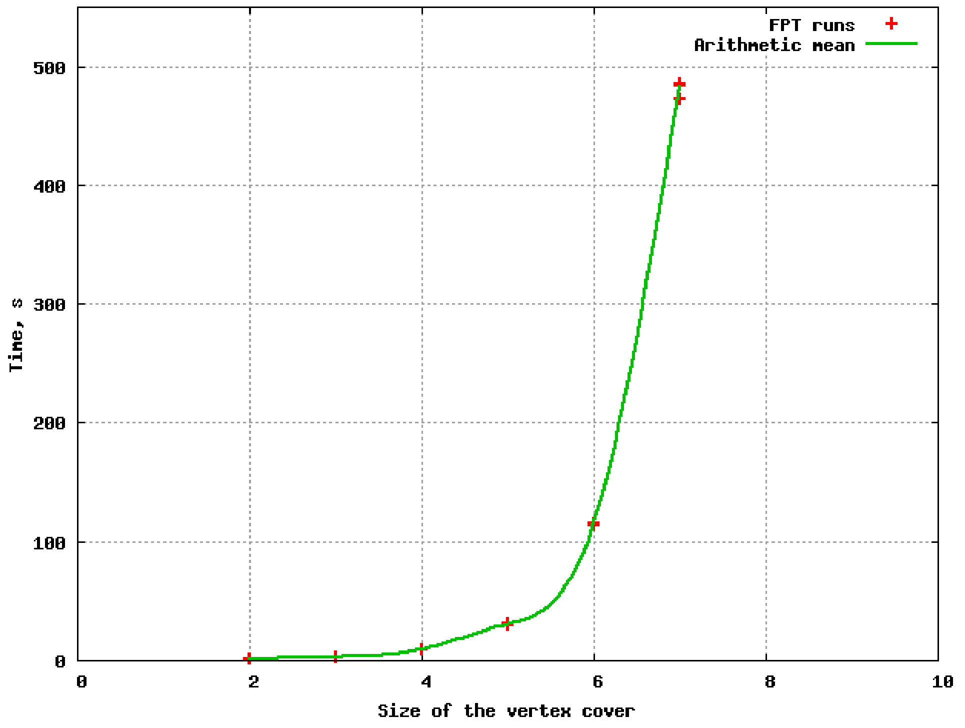


Figure 5.8: FPT running time for a graph with 200 vertices.

The consolidated plot for the experiments is given in Fig.5.9.

Generating a random graph with 500 vertices and a vertex cover of size 4, the FPT algorithm gives total execution time 118.925 s.

As the tests showed, the FPT algorithm works more or less quickly with the size of a vertex cover at most 10. Higher values still could be manageable, but would require much more of computational time. Values about 20-30 make the algorithm almost impractical, however they also could be handled with super-computers or/and with usage of parallel computations.

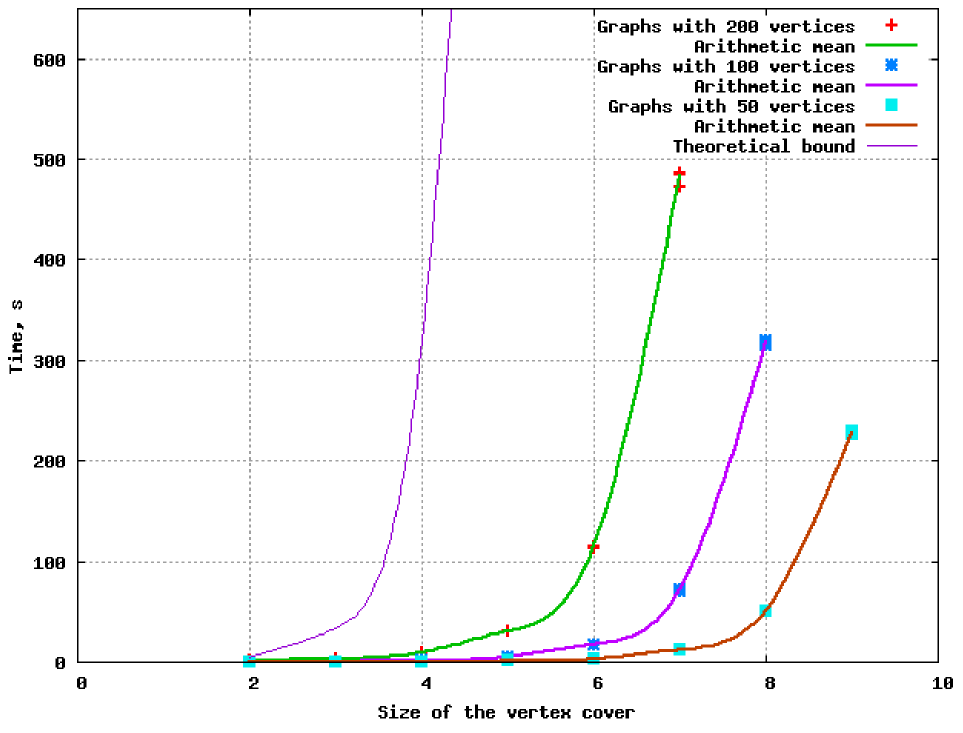


Figure 5.9: Consolidated plot for the experiments in Fig.5.6, Fig.5.7 and Fig.5.8

5.4.4 Influence of the graph size

For these tests we choose graphs with the following parameters:

- Probability of an edge = 0.3 (sparse graph).
- Size of a vertex cover = 5.
- Number of partitions: 5.

In the Fig.5.10 we can see the results of such tests (graph sizes = 50, 75, 100, 150, 200 vertices, the other points are interpolated).

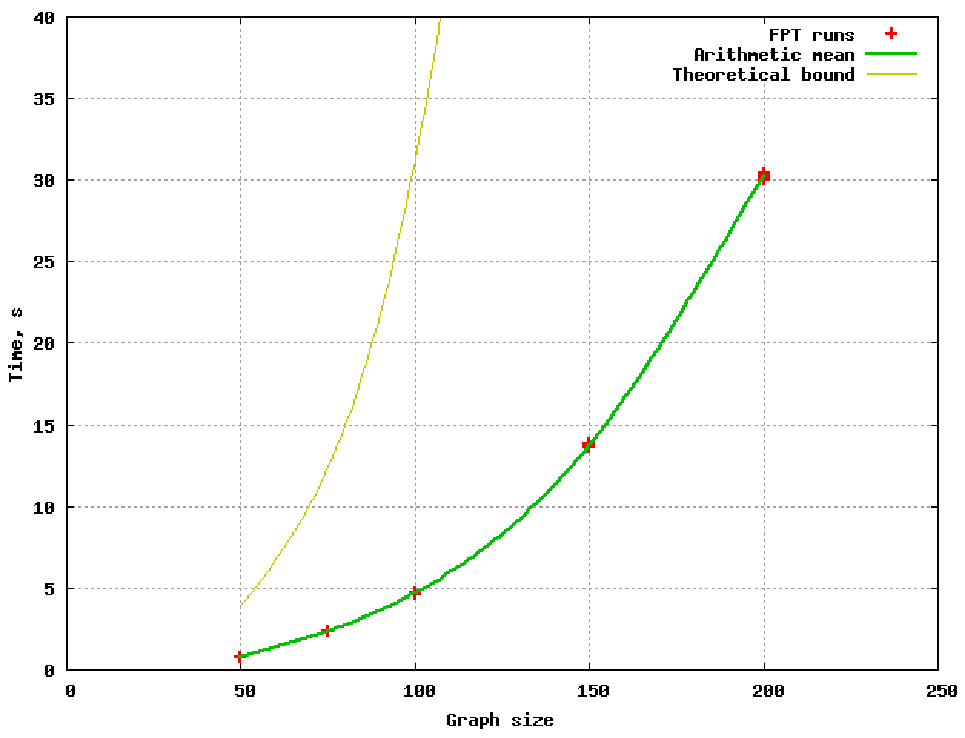


Figure 5.10: FPT running time for graphs of different sizes

It follows from the tests, that running time grows exponentially with growing of graphs. For a graph with 200 vertices and the vertex cover of size 10 the program took 60467.9 s, which is, approximately, 17 hours.

5.5 Genetic Algorithm

5.5.1 Speed of the selectors

First, we compare the speed of the selectors we chose in Section 3.3.4.

To perform these experiments we arbitrarily choose two graphs alongside with settings for the GA that differ only by the selector, and then we use them for 5 runs (in parallel) for each population size. The resulting plots of these tests are presented in Fig.5.11 and Fig.5.12.

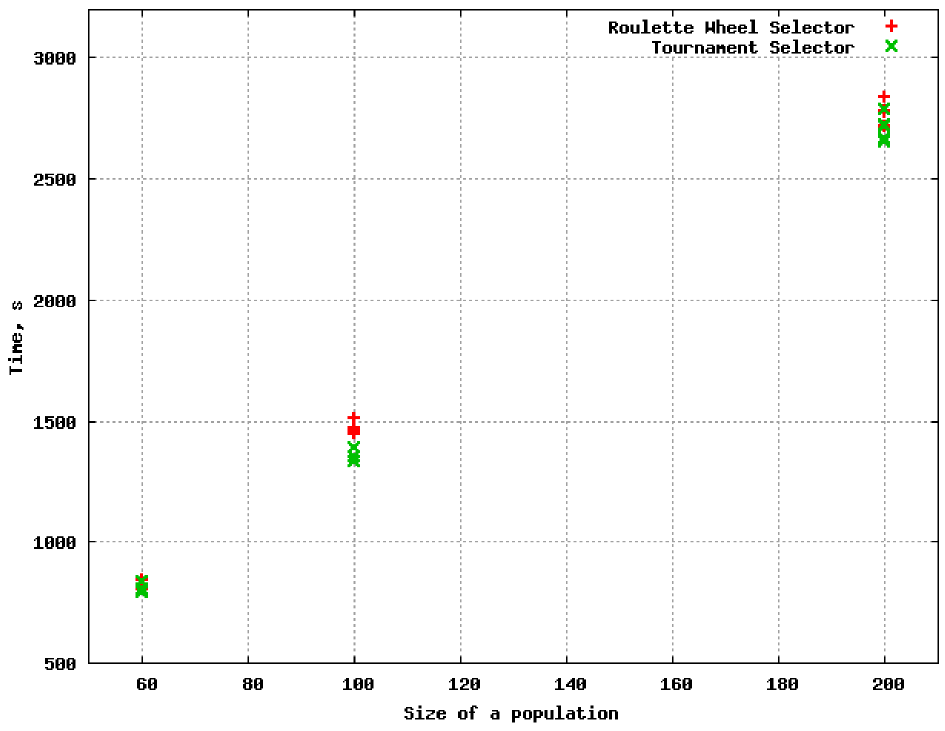


Figure 5.11: Time comparison for the Roulette Wheel and the Tournament selectors. Graph size: 50, VC size: 10, probability of an edge: 0.7, partition size: 5, XOver probability: 0.9, mutation probability: 0.4.

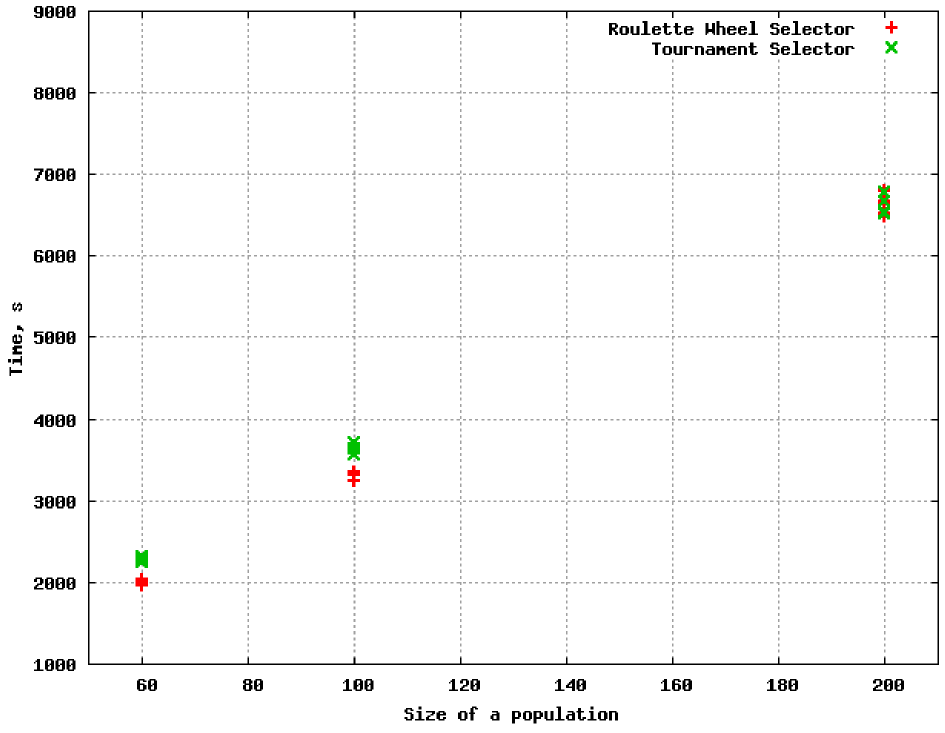


Figure 5.12: Time comparison for the Roulette Wheel and the Tournament selectors. Graph size: 200, VC size: 10, probability of an edge: 0.3, partition size: 10, XOver probability: 0.8, mutation probability: 0.4.

As we can see from Fig.5.11 and Fig.5.12, the Tournament Selector is not always faster than the Roulette Wheel Selector and thus, using these plots, we cannot conclude that usage of the Tournament Selector gives us any significant advantage in terms of the algorithm's speed. Oppositely, both selectors show almost the same speed and the little difference between their speeds can be explained by the current load of the processors during the tests by other processes.

The difference is vividly seen when a population has the size 1000. In that case the average difference is 80.9 s./generation (i.e. Tournament Selector is faster, indeed) for graphs with 100 vertices and the partition size - 4. However, we do not use such huge populations and we can say that the difference is insignificant for our experiments.

5.5.2 Population size influence

To reckon the influence of the population size on the speed of the algorithm we will fix some random setting of the GA.

Let us choose the following setting:

- Graph size: 50.
- Probability of an edge: 0.7.
- Size of the vertex cover: 10.
- Number of clusters: 5 .
- Selector: Roulette Wheel.
- Probability of crossover: 0.9.
- Probability of mutation: 0.2.
- Strategy: Best only, no transfer.

We run the GA 5 times for each population size thus getting the plot in Fig.5.13.

We can see a linear dependency of the GA's speed on the population size - quite natural result that we could expect. In Fig.5.13 the approximate equation of the arithmetic mean is $y = 10x + 150$.

5.5.3 Graph size influence

We will pick a random setting for the GA:

- Probability of an edge: 0.3.
- Size of the vertex cover: 10.
- Number of clusters: 10.
- Selector: Tournament Selector.

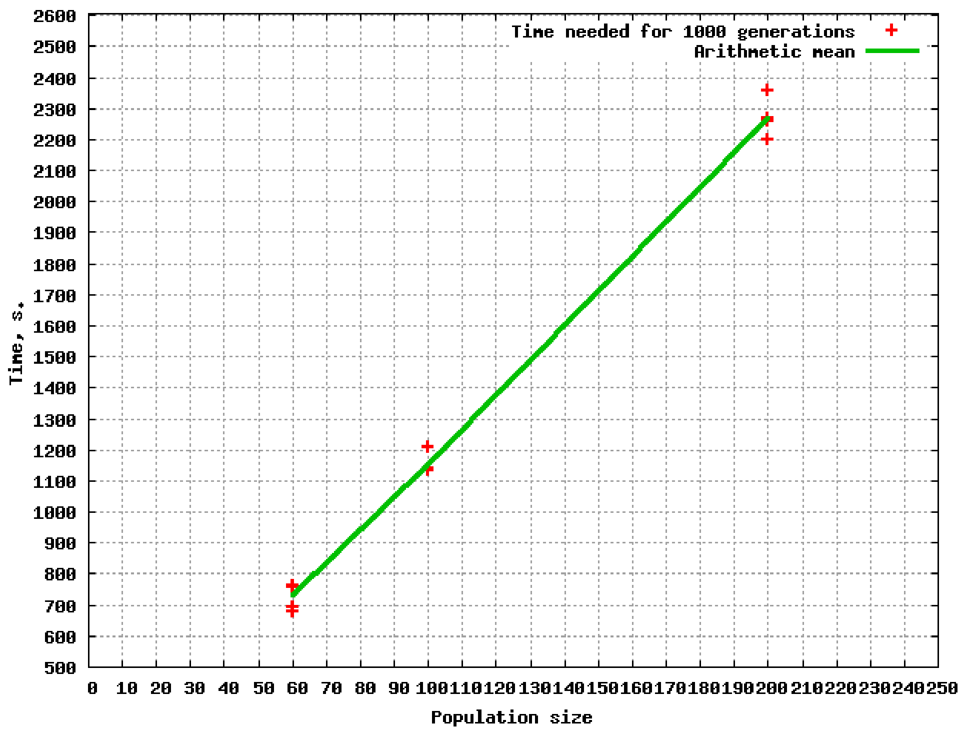


Figure 5.13: Population size influence. Graph size: 50, VC size: 10, probability of an edge: 0.7, partition size: 5, selector: Roulette Wheel, XOver probability: 0.9, mutation probability: 0.2, strategy: Best only, no transfer

- Probability of crossover: 0.8.
- Probability of mutation: 0.4.
- Population size: 100.

We do not specify a strategy here, instead we will use all possible strategies and will find the arithmetic mean of their running time. The plot is represented in Fig.5.14.

As we can see, this dependency is almost linear and the little deviation in case of the graph size equal to 200 can be explained by the high load of the processor that performed all these computations simultaneously in parallel.

5.5.4 Strategies

Hereinafter we will consider only a short series of plots so as to not overflow the text with them. However, if one wants to check the results of the experiments performed, they can use the data presented on the DVD which is attached to this thesis.

5.5.4.1 Kill parents without transferring of best individuals

As we predicted in Section 5.3, if we do not transfer any portion of the best individuals from one population to the next one, in case of the Roulette Wheel Selector the GA just blindly traverses the fitness landscape. This can be vividly

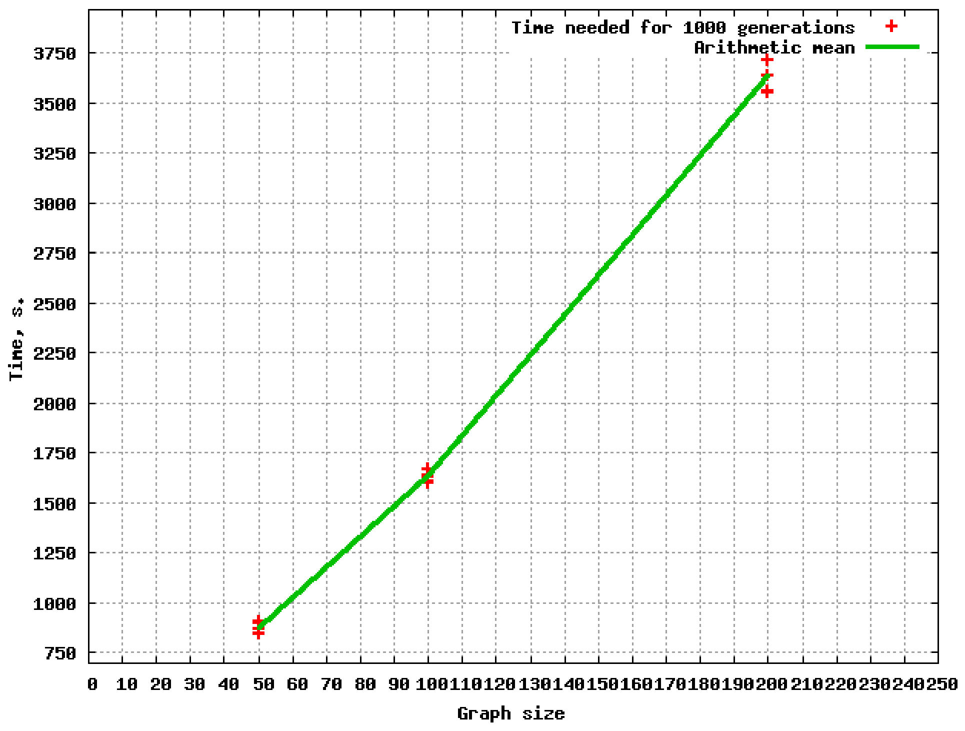


Figure 5.14: Graph size influence. VC size: 10, probability of an edge: 0.3, partition size: 10, selector: Tournament, XOver probability: 0.8, mutation probability: 0.4, population size: 100.

seen in Fig.5.15 and Fig.5.16, where the fitness function does not converge only for the "Kill parents, no transfer" strategy.

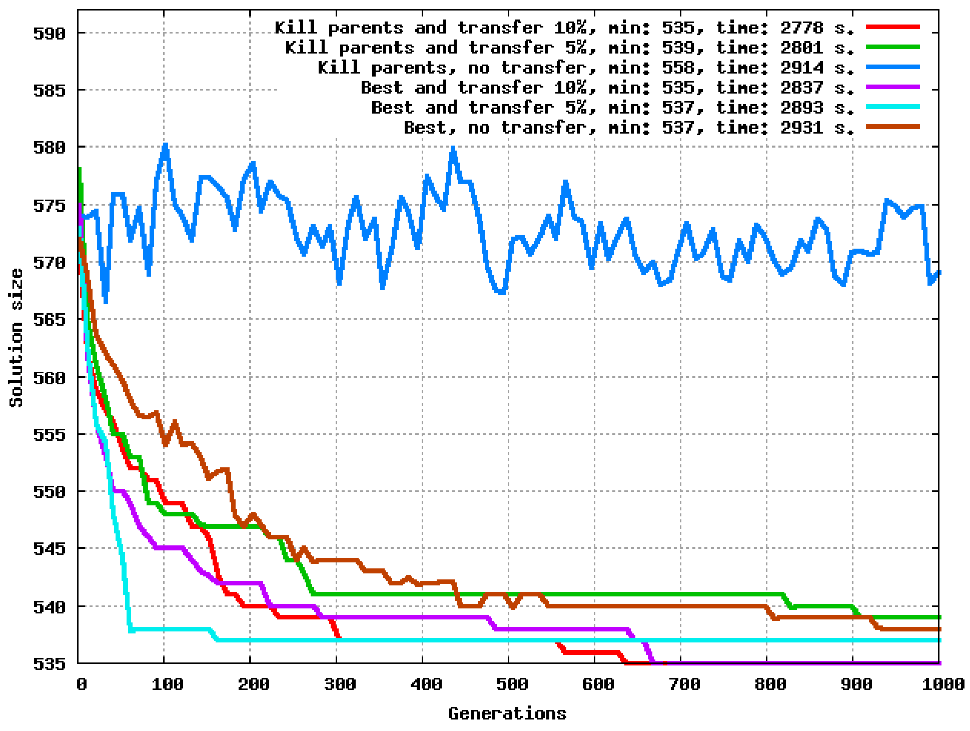


Figure 5.15: Strategies with the Roulette Wheel selector. Graph size: 50, VC size: 30, probability of an edge: 0.7, partition size: 5, XOver probability: 0.9, mutation probability: 0.2, population size: 60.

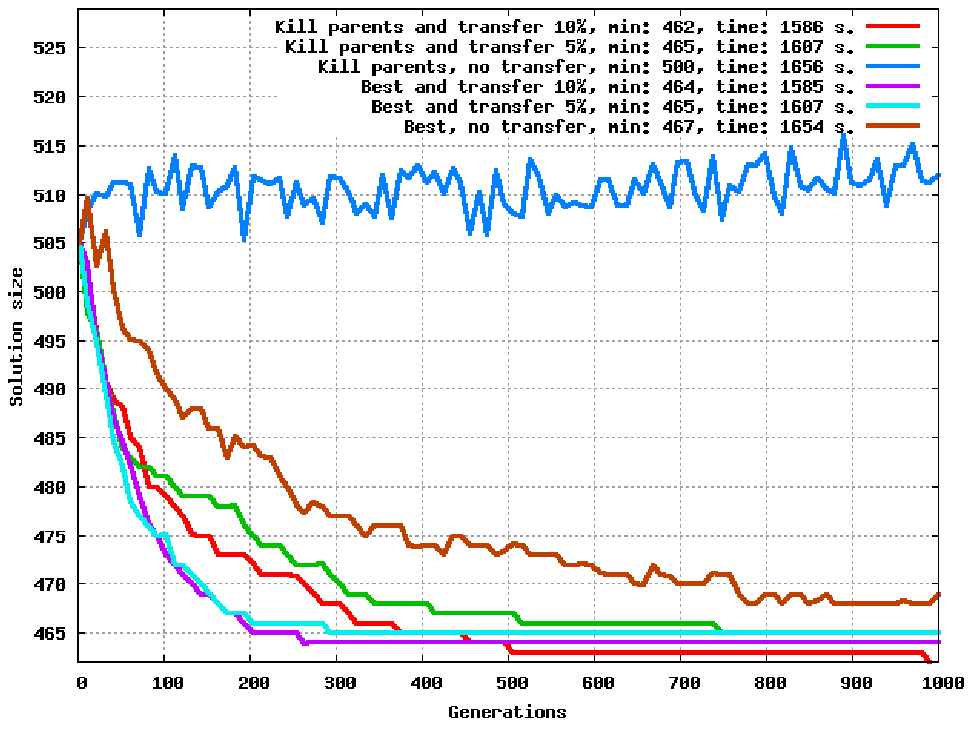


Figure 5.16: Strategies with the Roulette Wheel selector. Graph size: 100, VC size: 10, probability of an edge: 0.7, partition size: 5, XOver probability: 0.8, mutation probability: 0.4, population size: 60.

However, as we fairly noticed in Section 5.3, it is not the case if we use the Tournament selector (Fig.5.17).

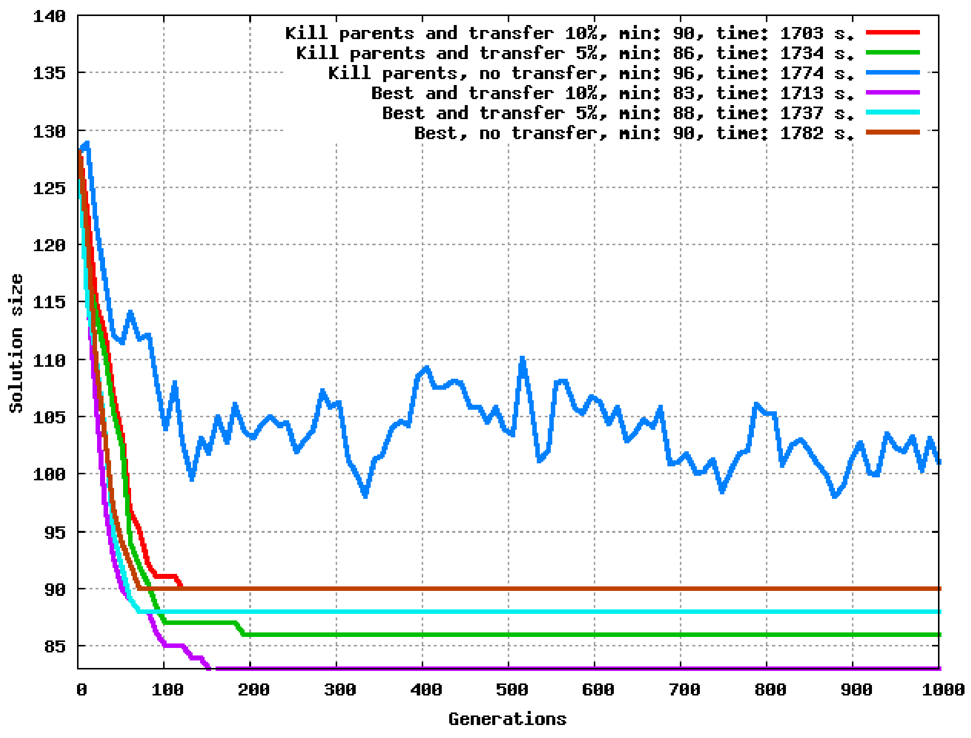


Figure 5.17: Strategies with the Tournament selector. Graph size: 100, VC size: 10, probability of an edge: 0.3, partition size: 2, XOver probability: 0.8, mutation probability: 0.4, population size: 100.

An interesting fact is that with this configuration of the GA the fitness function converges only partially. Thus, in Fig.5.17 we can observe that at some point it still looks like a random function but before this point the fitness function has converged. This happens because when the algorithm chooses two individuals for sexual reproduction it makes its choice arbitrarily. That implies that the algorithm can choose two equally bad individuals and they produce even worse offspring. However, having high probability of the best individual to be chosen, the algorithm sometimes may produce better offspring than their parents.

Anyway, this strategy does not appear to be reliable, so one should not use it.

5.5.4.2 Kill parents with partial transferring of best individuals

From Fig.5.15, Fig.5.16 and Fig.5.17 we can see that in all these cases the strategies when we kill parents but always transfer some portion of best individuals to a new population show convergence of the fitness function.

Also, as it was predicted earlier, the fitness function in case of the strategy where we transfer 10% of best individuals converges quicker than the fitness function of the strategy where we take only 5% of best individuals. However, in some cases it does not happen, but that is because sometimes we start with a good random initial population (with high average fitness) and the algorithm takes good individuals for reproduction thus giving us quick convergence of the fitness function.

5.5.4.3 Best only strategies

Let us consider a series of plots.

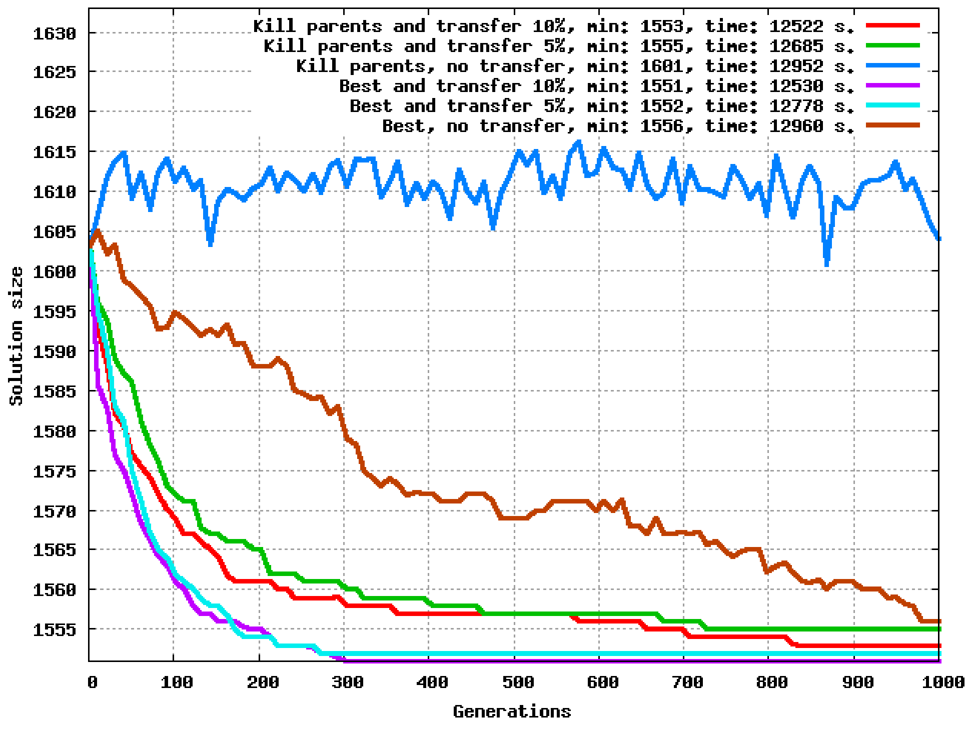


Figure 5.18: Strategies. Graph size: 100, VC size: 30, probability of an edge: 0.7, partition size: 10, selector: Roulette Wheel, XOver probability: 0.8, mutation probability: 0.2, population size: 200.

From Fig.5.18 we can notice that in the Best Only strategies the fitness function converges quicker than in cases when we exploit the ”Kill Parents” approach, as it was predicted in Section 5.3.

In Fig.5.19 we can see that all the strategies we are testing behave in the same manner - quick convergence and further degeneracy of the fitness function.

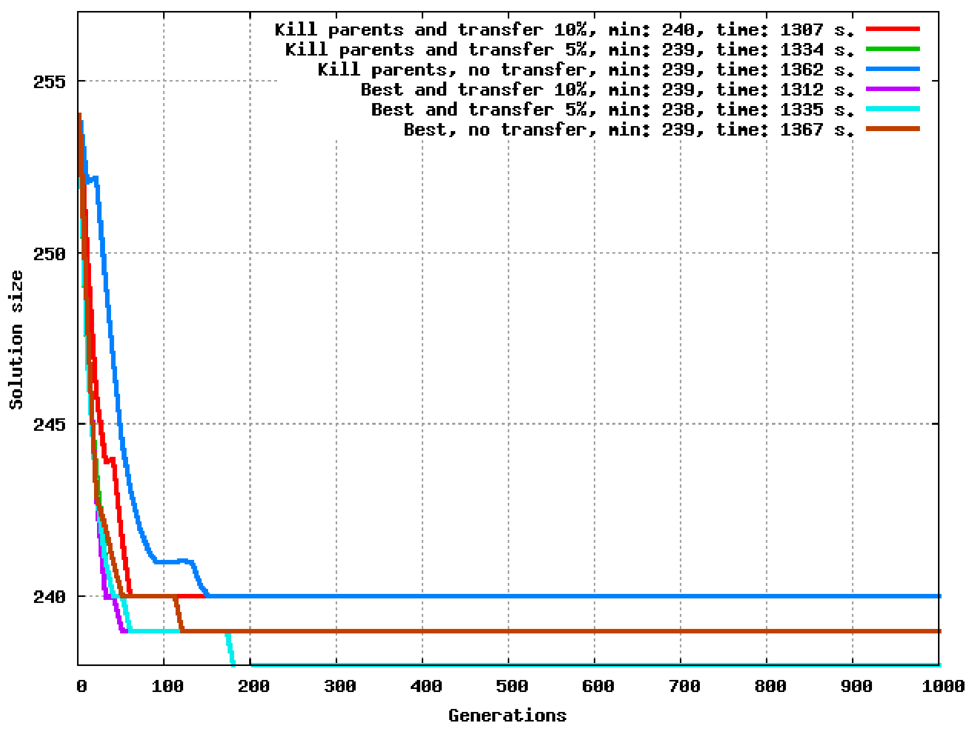


Figure 5.19: Strategies. Graph size: 50, VC size: 10, probability of an edge: 0.7, partition size: 5, selector: Tournament, XOver probability: 0.9, mutation probability: 0.2, population size: 100.

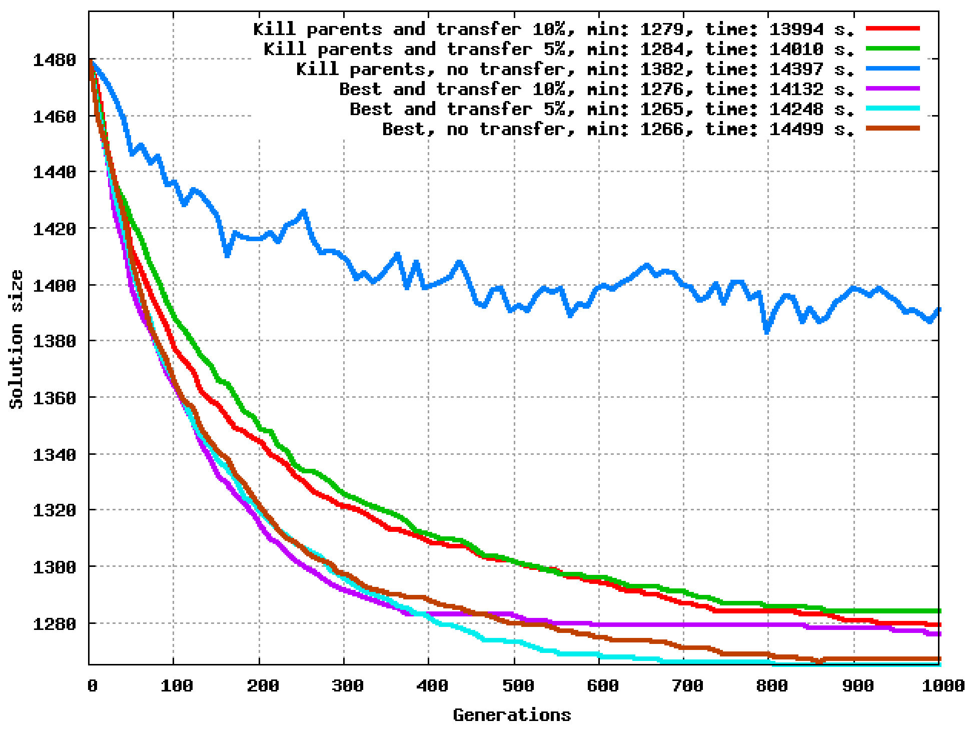


Figure 5.20: Strategies. Graph size: 200, VC size: 30, probability of an edge: 0.3, partition size: 10, selector: Tournament, XOver probability: 0.8, mutation probability: 0.4, population size: 200.

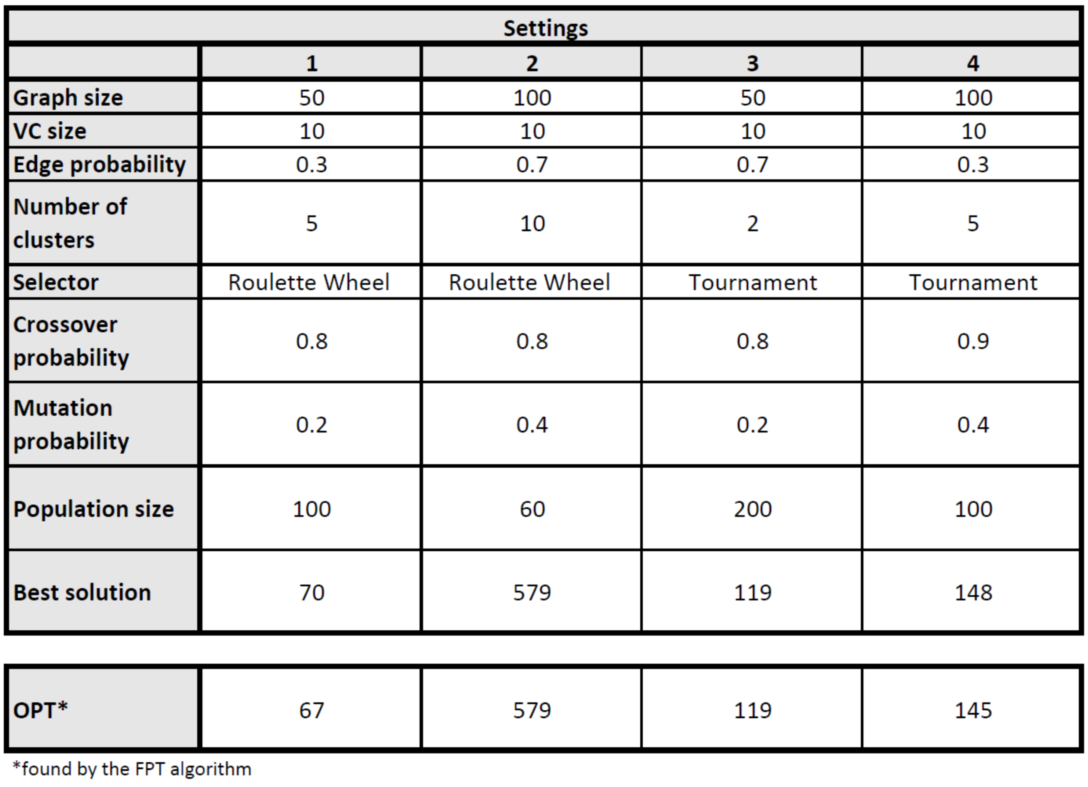
Finally, Fig.5.20 is a good example at which we can witness our predictions from Section 5.3 - all the ”Best only” strategies converge quicker than the ”Kill parents” ones.

5.5.5 Optimum

The most important question is whether the GA is able to find an optimal solution.

We compare different settings of the GA, chosen at random, and try to figure out which setting gives best results.

Let us take 2 settings with the Tournament Wheel Selector and 2 settings with the Roulette Wheel Selector, Fig.5.21.



Settings				
	1	2	3	4
Graph size	50	100	50	100
VC size	10	10	10	10
Edge probability	0.3	0.7	0.7	0.3
Number of clusters	5	10	2	5
Selector	Roulette Wheel	Roulette Wheel	Tournament	Tournament
Crossover probability	0.8	0.8	0.8	0.9
Mutation probability	0.2	0.4	0.2	0.4
Population size	100	60	200	100
Best solution	70	579	119	148

OPT*	67	579	119	145

*found by the FPT algorithm

Figure 5.21: Settings of the GA.

We will compare all possible strategies and, to not mix all plots, we will build a separate plot for each setting.

We start with the first setting shown in Fig.5.22.

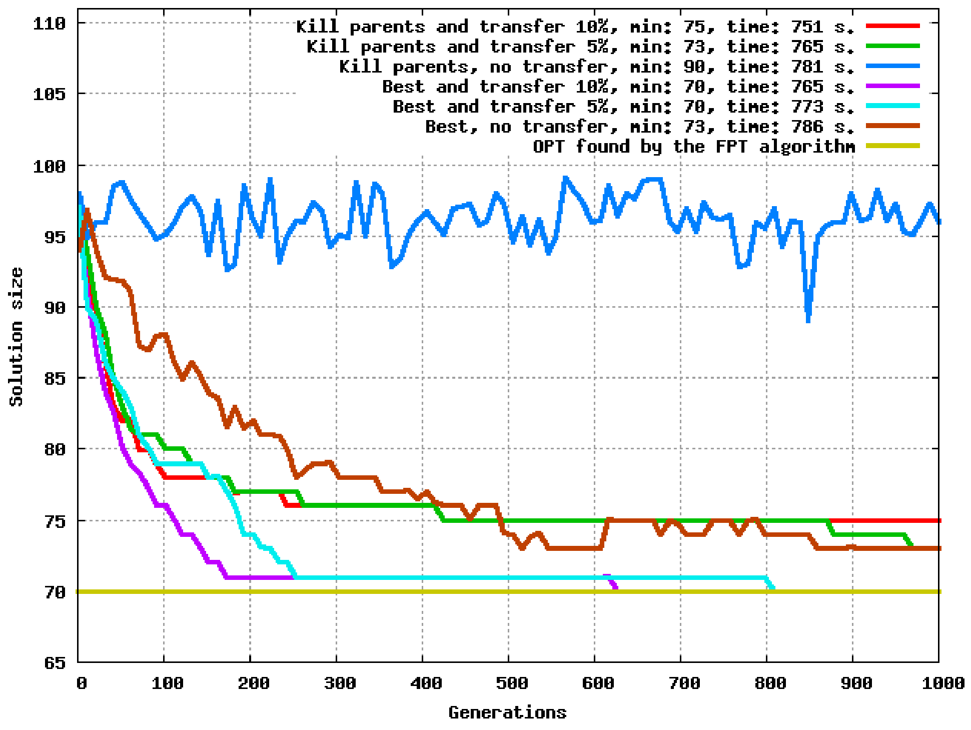


Figure 5.22: First setting.

Not surprisingly, "Kill parents, no transfer" strategy is still almost random. The minimum value it found was 96 which is rather far from the optimum.

The quickest strategy was "Best and transfer 10%" - it managed to find a solution with cost 70, however this result is not optimal.

As we can see from the plot, all the strategies did not succeed - most of them just got stuck at a local optimum.

Let us take a look at the second setting, Fig.5.23.

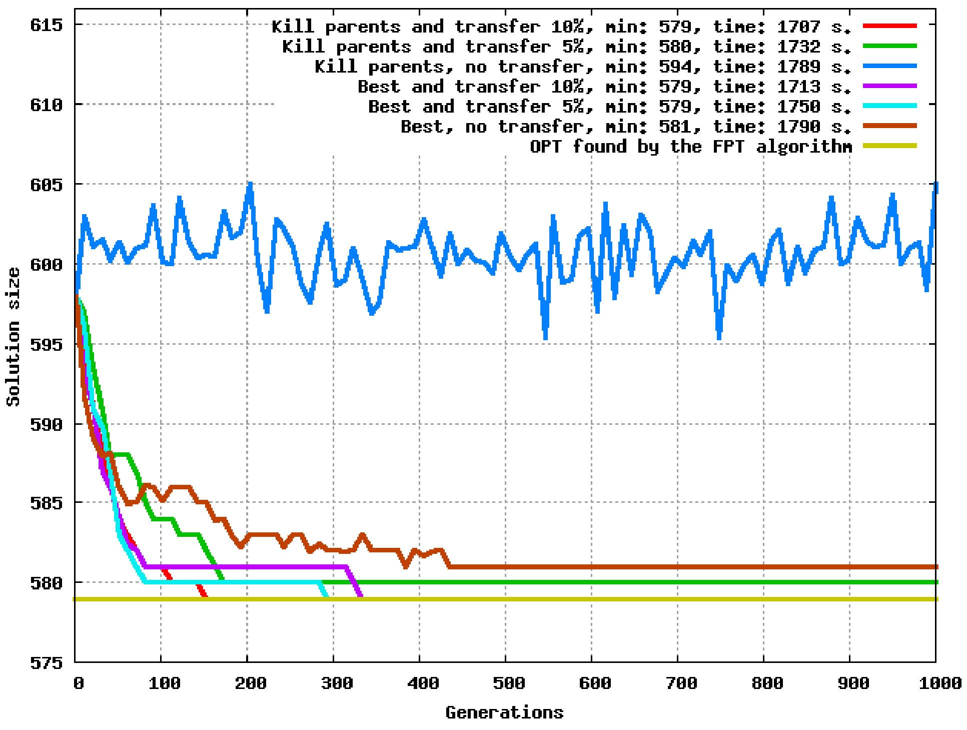


Figure 5.23: Second setting.

Here 3 strategies succeeded at the same time - they found an optimum. The quickest was ”Kill parents and transfer 10%”. The rest of strategies, except ”Kill parents, no transfer”, got stuck at local optimums. However, these local optimums are not far from the optimum in terms of magnitude. Thus we can witness that all these strategies work and can be used in practice.

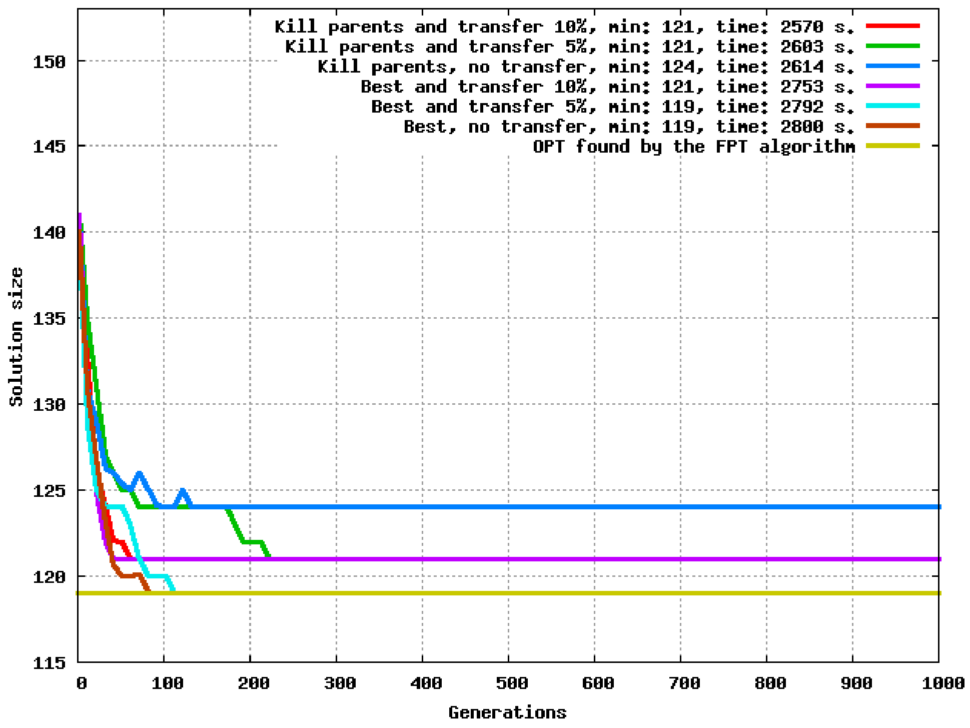


Figure 5.24: Third setting.

In Fig.5.24 the plot for the third strategy is shown. Here, we can see extremelly quick convergence, because the partition has only 2 clusters. However, now only two settings found an optimal solution: the strategies "Best and transfer 5%" and "Best, no transfer". The others got stuck at local optimums and their curves degenerated into a horizontal line.

Surprisingly, "Kill parents, no transfer" also degenerated, but this is simply because we have only two clusters and thus we do not have that many options to traverse the fitness landscape.

Finally, we have the fourth setting, Fig.5.25.

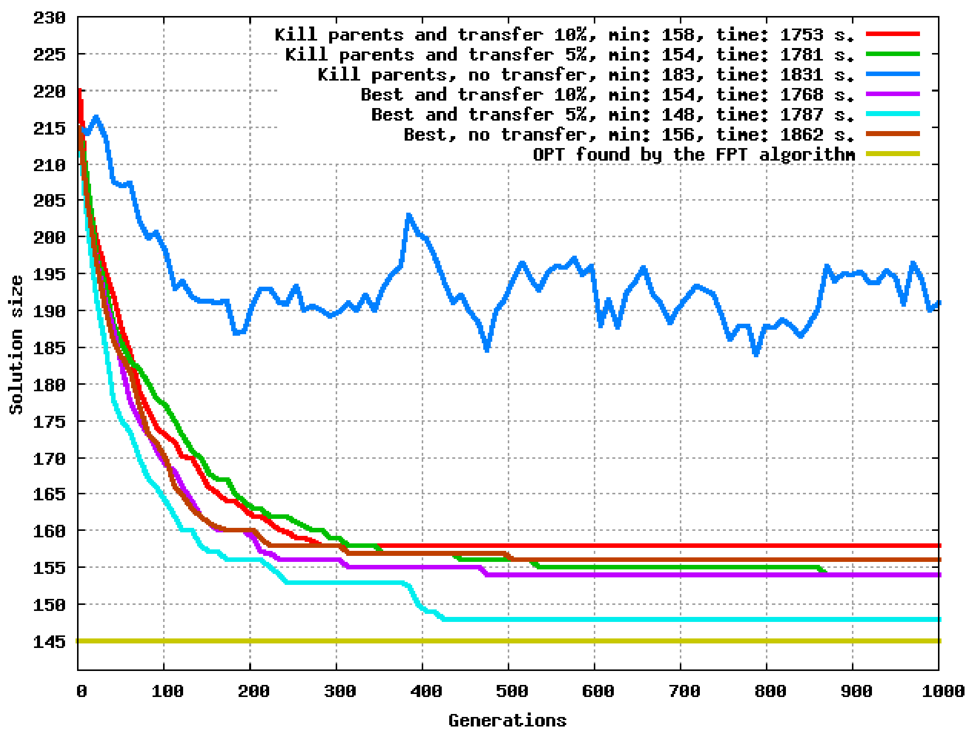


Figure 5.25: Fourth setting.

Here we can see that all the strategies (again, except for "Kill parents, no transfer") almost identical, and the difference is only in the local optimums they are captured at in the end. However, neither of the strategies found an optimum; the best strategy is "Best and transfer 5%".

5.5.6 Cumulative time complexity

Let us show the dependency of the GA running time on the number of generations for the following settings:

- Graph size: 100.
- Probability of an edge: 0.3.
- Size of the vertex cover: 10.
- Number of clusters: 5.
- Selector: Roulette Wheel Selector.
- Probability of crossover: 0.9.
- Probability of mutation: 0.2.
- Population size: 100.
- Strategy: Kill parents and transfer 10

The dependency is shown in Fig.5.26 and as we can see it is linear, as was predicted in Section 3.4.

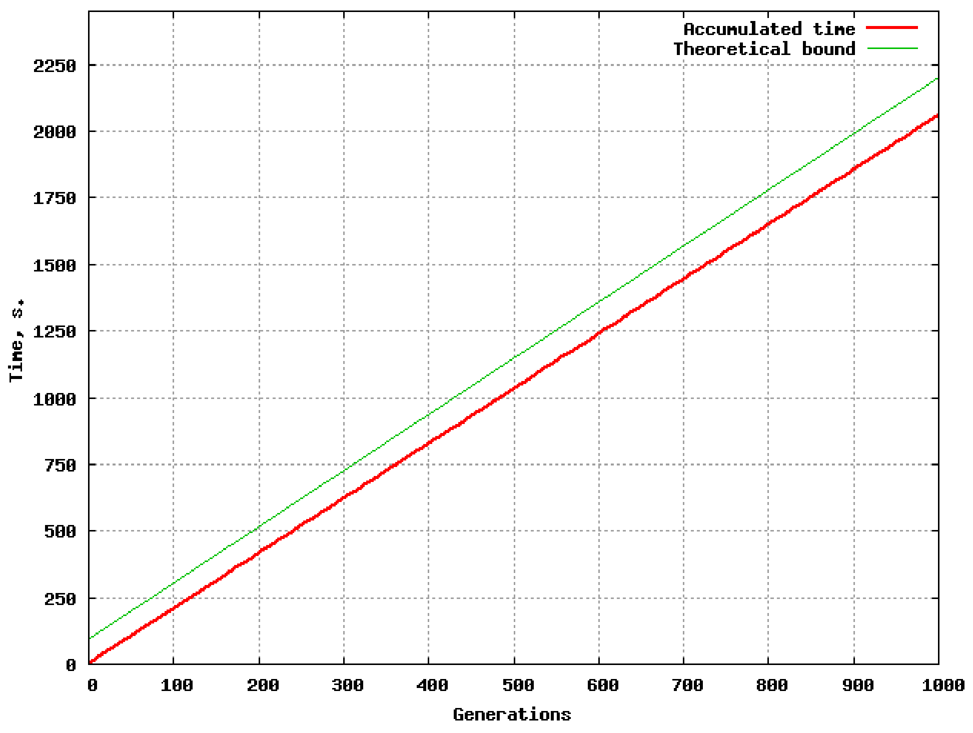


Figure 5.26: The time complexity of the GA.

5.5.7 General observations

As we can see from the plots, there are no breaks on them and the functions are continuous. This is because the algorithm behaves in the exact manner as we expected - it uses crossover and mutation correctly and therefore evolves populations using previously accumulated information. Thus, it almost never deteriorates populations.

The only exception is the ”Kill parents, no transfer” strategy that almost blindly traverses the fitness landscape. However, even here it relies on the previously revealed information since it still uses crossover and mutation that do not allow the algorithm to ”jump” into some completely random area of the fitness landscape.

As we expected, generally speaking, the bigger portion of best individuals we transfer to a new population, the faster the fitness function converges. On the other hand, the fewer best individuals we transfer, the more time we give to the algorithm for traversal of the fitness landscape. It can be vividly seen from the plots owing to the fact that in such cases the fitness function curve is flatter.

Also, as we can notice, the Roulette Wheel Selector in the current version of the program builds a wheel every time it is being executed. However, the time complexity of the selector can be improved by introducing the ”init” method that could build such a wheel at the initialization step, and then could just use the same wheel for the whole generation. That would significantly decrease the Roulette Wheel Selector time complexity for the whole generation: $\Theta((2m + \log m) \cdot m)$ to just $\Theta(2m + m \cdot \log m)$.

Out of the experiments mentioned in Section 5.5.4 and Section 5.5.5 we can observe that we do not have any clear favorite among the strategies and the settings we used. However, the "Best only" strategies give slightly better results in general.

The most important observation is that even though the GA does not always find an optimal solution, it approaches optimum quite close, as we can see in Section 5.5.5. Thus, usage of the GA for the Balanced Graph Partitioning problem solving is definitely reasonable and may be advised when other, more precise algorithms, are unavailable.

Epilogue and Future Work

Nowadays, the vast majority of modern computers have multi-core architecture - an architecture that exploits so-called *multi-core processors*.

A multi-core processor has two or more separate processing units, called *cores*, and each such a core is able to execute program instructions independently.

Multi-core processors are able to increase overall speed of computer programs if we use parallel computing techniques. One such technique, *task parallelism*, parallelizes computer code across several cores in multi-core environments.

The FPT algorithm implemented in this thesis can be efficiently parallelized - we only need to distribute restricted growth strings between cores and each core must keep its local optimum. The global optimum can be found simply by taking the best result among all the cores.

Regarding the GA, it also can be parallelized - we can select individuals for sexual reproduction, perform crossover and mutation in parallel while having the same population for all cores.

Thus, the running time of both algorithms can be reduced in a multi-core environment.

An interesting future work would be to evolve settings of the GA - in this case an individual would be a setting of the GA algorithm and its fitness would be, say, the fitness of a best individual found by the algorithm. However, in this case, it would take a lot of computational time.

Out of the results of the experiments performed for this thesis we can notice that our predictions about the GA's behavior were correct. Thereby, we can conclude that its behavior correlates with the theory and having this fact in mind we can definitely exploit this algorithm as a heuristic approach to solve the Balanced Partitioning problem, when other, more precise and quicker algorithms, are unavailable for us. The biggest advantage is that every generation contains only correct individuals meaning that every individual represents a correct solution of the problem. Thus, we can stop execution of the algorithm at any time taking the best individual as an approximate solution. However, as the theory says, the more time the GA has, the better result we may achieve, and this leads us to the conclusion that if we want to get better result we must sacrifice time - nothing comes for free. However, this safety is a big advantage that can be used in applications that do not require an optimal solution.

Bibliography

[1] METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows. Theory, Algorithms and Applications.* Prentice-Hall Inc., Upper Saddle River, New Jersey, USA, 1993. ISBN 0-13-617549-X.

[3] K. Andreev and H. Racke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39:929, 2006. URL `https://doi.org/10.1007/s00224-006-1350-7`.

[4] D.A. Bader, H. Mayerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering.* Contemporary Mathematics, 201 Charles Street, Providence, Rhode Island 02904-2294, USA, 2013.

[5] H. Bateni and K. Aydin. Balanced Partitioning and Hierarchical Clustering at Scale. *Google AI Blog*, 2018. URL `https://ai.googleblog.com/2018/03/balanced-partitioning-and-hierarchical.html`.

[6] B. Bollobas. *Graph Theory. An Introductory Course.* Springer-Verlag New York Inc., 175 Fifth Avenue, New York, USA, 1979. ISBN 0-387-90399-2.

[7] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In L. Kliemann and P. Sanders, editors, *Algorithm Engineering. Selected Results and Surveys*, volume 9220, pages 117–158. Springer, 2016. doi: 10.1007/978-3-319-49487-6_4.

[8] M. Cygan, F.V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms.* Springer, Switzerland, 2016. ISBN 978-3-319-21274-6.

[9] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms.* McGraw-Hill Higher Education, 2006. ISBN 978-0073523408.

[10] R.G. Downey and M.R. Fellows. Parameterized computational feasibility. In P. Clote and J.B. Remmel, editors, *Feasible Mathematics II*, pages 219–244. Birkhäuser Boston, Boston, MA, 1995.

[11] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:2:291–307, 1970.

[12] J. Kim, I. Hwang, Y-H. Kim, and B. Moon. Genetic Approaches for Graph Partitioning: A Survey. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 473–480, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0557-0. doi: 10.1145/2001576.2001642.

[13] Kim, H. and Kim, Y. Recent Progress on Graph Partitioning Problems Using Evolutionary Computation. 2018. arXiv 1805.01623.

[14] D.E. Knuth. *The Art of Computer Programming.* Addison-Wesley, Boston, 2011. ISBN 978-0-201-03804-8.

[15] J. Matousek and B. Gartner. *Understanding and Using Linear Programming.* Springer-Verlag Berlin Heidelberg, 2007. ISBN 978-3-540-30697-9.

[16] M. Mitchell. *An Introduction to Genetic Algorithms.* A Bradford Book The MIT Press, 1998. ISBN 0-262-13316-4.

[17] A. Ocheretyanyy. Repository: Bachelor Thesis - Balanced Partitioning of Graphs with Small Vertex Cover. `https://github.com/Alexander-Ocheretyany/Bachelor-Thesis`.

[18] A.S Pope, D.R. Tauritz, and A.D. Kent. Evolving Multi-level Graph Partitioning Algorithms. *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2016.

[19] F. Rahimian, R.H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. JA-BE-JA: A distributed algorithm for balanced graph Partitioning. *International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, pages 51–60, 09 2013. doi: 10.1109/SASO.2013.13.

[20] C.P. Ravikumar. *Parallel Methods for VLSI Layout Design.* Greenwood Publishing Group Inc., Westport, CT, USA, 1995. ISBN 0893918288.

[21] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the Meting on Algorithm Engineering & Expermiments*, ALENEX '12, pages 16–29, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[22] K. Schloegel, G. Karypis, and V. Kumar. *Graph Partitioning for High Performance Scientific Simulations.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-55860-871-0.

[23] René van Bevern, A.E. Feldmann, M. Sorge, and O. Suchy. On the Parameterized Complexity of Computing Balanced Partitions in Graphs. *Theory of Computing Systems*, 57(1):1–35, 2015.

[24] Vijay V. Vazirani. *Approximation Algorithms.* Springer-Verlag, 2001. ISBN 3-540-65367-8.

[25] X. Yu and M. Gen. *Introduction to Evolutionary Algorithms.* Decision Engineering. Springer-Verlag, London, 2010. ISBN 978-1-84996-128-8.