VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

# Graph Partitioning Using Spectral Methods

2006 Pavla Kabelíková

I declare I created this thesis myself. I have cited all literary sources and publications that I have used.

Ostrava, May 8, 2006 .....

I would like to express my thanks to Prof. RNDr. Zdeněk Dostál, DSc. who allowed me the research fellowship at the Montanuniversität Leoben; to Ao. Univ. Prof. Dr. Clemens Brand who helped me during whole five months and gave me rich advices and to O. Univ. Prof. Dr. Wilfried Imrich for his support and kind approach.

## Abstrakt

Pro paralelní programování jsou výpočetně rozsáhlé grafy rozdělovány do subdomén a distribuovány na samostatné procesory. Tato práce prezentuje spektrální metody jako nástroj pro dělení grafů. Lanczosův algoritmus pro výpočet požadovaného vlastního vektoru je důležitou částí spektrálních metod. Pro vyšší výkon jsou imlemenovány modifikace původního Lanczosova algoritmu. Je také popsán víceúrovňový přístup jako možné zefektivění. Nakonec jsou prezentovány výsledky na testovacím souboru a je popsán napsaný software.

**Klíčová slova:** dělení grafu, spektrální metody, Lanczosův algoritmus, řídké matice

## Abstract

For parallel computing, large computational graphs are partitioned into subdomains and distributed over individual processors. This thesis presents spectral partitioning methods as a tool for graph partitioning. A Lanczos algorithm is an important component of the spectral partitioning problem to compute the required eigenvector. For better peformance, some modifications and improvements to the original Lanczos algorithm are implemented. A multilevel approach as a possible improvement is also described. Finally, results on testing set are presented and the written software is described.

**Keywords:** graph partitioning, spectral methods, Lanczos algorithm, sparse matrix

# Contents

# List of Figures

# List of Tables

# 1 Introduction

For parallel computing, large computational graphs are partitioned into subdomains and distributed over individual processors. The cost of communication between processors depends on the number of edges between different subdomains. Load balance constraints require (approximately) equal-sized subdomains.

Finding a partition with minimum number of edges cut is, in general, an NP-complete problem. Suitable heuristics are based on spectral methods. Spectral methods for graph partitioning have been known to be robust but computationally expensive.

We will show the development of hardware equipment enables to compute with implemented spectral methods in reasonable time. We will present the only one limitation is the size of operationg memory that can be partially solve using additional storage on hard disk memory.

The size of the separator produced by spectral methods can be related to the Fiedler value - the second smallest eigenvalue of an adjacency structure. We associate with the given sparse, symmetric matrix and its adjacency graph a matrix called the Laplacian matrix. In its simplest form, one step of spectral bisection sets up the Laplacian matrix of the graph, calculates an eigenvector corresponding to the second-smallest eigenvalue and partitions the graph into two sets $A$, $B$ according to the sign pattern of the eigenvector components. The set of edges joining $A$ and $B$ is an edge separator in the graph $G$.

The use of spectral methods to compute edge separators in graphs was first considered by Donath and Hoffman who first suggested using the eigenvectors of adjacency matrices of graphs to find partitions. Fiedler associated the second smallest eigenvalue of the Laplacian matrix with its connectivity and suggested partitioning by splitting vertices according to their value in the corresponding eigenvector. Thus, we call this eigenvalue the Fiedler value and a corresponding vector the Fiedler vector. Since then spectral methods for computing various graph parameters have been considered by several others. Good summary we can foung e.g. in [14].

Although, various software is available for working with sparse matrix and for computing eigenvalues and eigenvector, there is no suitable implementation for computing them under C++. Thus, a one part of this work includes implementation of the Lanczos algorithm in C++. The software aims at working with graph partitioning and at visualization of the graph partitioning using spectral methods. Used

algotithms work without coordinate information but we need the coordinate information for the visualisation part which is one of the assets of our software.

In a given testing set of graphs the majority of them has a simple geometric shape. For testing some other graphs were given and the quality of spectral partitioning was investigated. In this case the results were not acceptable so another improvement was included. The sparse matrix format [11] is used and also the special graphics format was suggested and the various conversions between these formats were included.

In second section, we bring a background on the spectral properties of the Laplacian matrix and the Fiedler vector and we presented the idea of graph partitioning in general. In section 3, we deal with the spectral partitioning algorithms using the Lanczos method for getting the second eigenvector of graph. A multilevel algorithm is a possibility how to increase performance of whole partitioning. We present it in section 4. In section 5, we are solving the problem of disconnected graphs partitioning. The results on testing sets are presented in section 6.

In appendix, we bring some overview about storage formats using in written software and there is also short software documentation included.

# 2 Background

## 2.1 Laplacian matrix and Fiedler Vectors

Let $G = (V, E)$ be an undirected, unweighted graph without loops or multiple edges from one node to another, on $|V| = n$ vertices. Let $A = A(G)$ be an $n \times n$ adjacency matrix relevant to $G$ with one row and column for each node, $a_{u,v}$ equal to one if $(u, v) \in E$ and zero otherwise. For a simple graph with no loops, the adjacency matrix has on the diagonal. For an undirected graph, the adjacency matrix is symmetric.
The *Laplacian matrix* $L(G)$ of G is an $n \times n$ symmetric matrix with one row and column for each node defined by

$$L_{(ij)}(G) = \begin{cases} d_i & if \quad i = j \\ -1 & if \quad i \neq j \quad and \quad (i,j) \in E \\ 0 & otherwise \end{cases}$$

for $i, j = 1, ..., n$, where $d_i$ is the vertex degree of node $i$. Since the Laplacian matrix is symmetric, all eigenvalues are real. From the Gershgorin circle theorem, all eigenvalues are nonnegative. Thus, $L(G)$ is positive semidefinite.

The matrices $L(G)$ and $A(G)$ are related via

$$L = D - A,$$

$D$ is a diagonal matrix where $d_{ii}$ is the vertex degree of node $i$.

Let the eigenvalues of $L(G)$ be ordered $\lambda_0 = 0 \leq \lambda_1 \leq ... \leq \lambda_{n-1}$. An eigenvector corresponding to $\lambda_0$ is vector of all ones. The multiplicity of $\lambda_0$ is equal to the number of connected components of the graph. The second smallest eigenvalue $\lambda_1$ is greater then zero iff G is connected. Fan R. K. Chung and B. Mohar [4], [8] list basic properties of the Laplacian matrix and present a survey of know results about the spectrum of $L(G)$ with special emphasis on $\lambda_1$ and its relation to numerous graph invariants. Fiedler [5], [6] calls this number *algebraic connectivity*. Fiedler also investigated graph-theoretical properties of the eigenvector corresponding to $\lambda_1$, called second eigenvector. The coordinates of this eigenvector are assigned to the vertices of $G$ in a natural way and can be considered as valuation of the vertices of $G$. Fiedler called this valuation *characteristic valuation of G*. Some of his results follows.

**Theorem 2.1.1** *Let $G$ be a finite connected graph with $n$ vertices $1, ..., n$. Let $y = (y_i)$ be a characteristic valuation of $G$. For any $r \geq 0$, let*

$$M(r) = \{i \in N | y_i + r \geq 0\}.$$

*Then the subgraph $G(r)$ induced by $G$ on $M(r)$ is connected.*

**Remark 2.1.1** *A similar statement holds for $r \leq 0$ and the set $M'(r)$ of all those i's for which $y_i + r \leq 0$.*

From these results simply follows that by dividing a finite connected graph into two parts according to the theorem at least one of the parts is connected. Practical results we describe later.

**Corollary 2.1.1** *Let $G$ be a valuated connected graph with vectices 1,2,...n, let $y = (y_i)$ be a characteristic valuation of $G$.*

1. *If $c$ is a number such that $0 \leq c < max(y_i)$ and $c \neq y_i$ for all $i$ then the set of all those edges (i, k) of $G$ for which $y_i < c < y_k$ forms a cut $C$ of $G$. If $N_1 = \{k \in N | y_k > c\}$ and $N_2 = \{k \in N | y_k < c\}$ then $N = (N_1, N_2)$ is a decomposition of $N$ corresponding to $C$ and the subgraph $G(N_2)$ is connected.*
2. *If $y_i \neq 0$ for all $i \in N$ the set of all alternating edges, i.e. edges $(i, k)$ for which $y_i y_k < 0$, forms a cut $C$ of $G$ such that both subgraphs of $G$ are connected.*

Let us have a look on some other important properties of $L(G)$. These are standard results, presented e.g. [1], [5].

**Theorem 2.1.2** *Suppose $v \in R^n$, $v \neq 0$. Then*

$$\vec{v}^T L(G) \vec{v} = \sum_{(i,j) \in E} (x_i - x_j)^2$$

*Moreover if $L(G) \cdot v = \lambda \cdot v$, the second smallest eigenvalue $\lambda_1$ is given by*

$$\lambda_1 = \min_{\vec{v} \perp (1,1,...,1)} \frac{\vec{v}^T L(G) \vec{v}}{\vec{v}^T \vec{v}}$$

## 2.2 Graph Partitioning

Let a graph $G = (V, E)$ be the computational grid of some large-scale numerical problem. We think of a node $v_i$ in $V$ as representing an independent job to do. An edge $e = (i, j)$ in $E$ means that an amount of data must be transferred from job $i$ to job $j$ to complete all tasks. More generally, we can consider a graph $\widetilde{G} = (V, E, W_V, W_E)$ with the weights $W_V$ as node weights, a nonnegative weight for each node, and $W_E$ as edge weights, a nonnegative weight for each edge. $W_{V_{v_i}}$ means the cost of job $i$, $W_{E_{e(i,j)}}$ means amount of data that must be transferred from job $i$ to job $j$.

Partitioning $G$ means dividing $V$ into the union of $n$ disjoint sets

$$V = V_1 \cup V_2 \cup ... \cup V_n$$

where the nodes (jobs) in $V_i$ are assigned to be done by processor $P_i$. For optimal partitioning, some conditions should hold:

1. The sums of the weights $W_V$ of the nodes in each $V_i$ should be approximately equal. This guarantees a load balance condition across processors.
2. The sum of the weights $W_E$ of edges connecting nodes in different $V_i$ and $V_j$ should be minimized. This minimizes the inter-processor communication.

In following text, we will always use $W_{V_{v_i}} = 1$ and $W_{E_{e(i,j)}} = 1$ for all $i, j$.

On Figure 1, we can see a well-partitioned graph. The green nodes belong to $V_1$ and the blue nodes belong to $V_2$. Both sets are the same size and the sum of edges connecting $V_1$, $V_2$ is minimal, so both conditions are satisfied.

Parallel computing is a way to decrease computational time for numerical problems related to large computational grids, or graphs. Graphs are partitioned into subdomains and distributed over individual processors. Partial problems can be solved in individual processors, but in global view we need informations from all processors together. The main idea we can describe for the important problem of the *sparse matrix-vector multiplication*.

Consider the computation $y = A \cdot x$, where the sparsity pattern of $A$ corresponds to the adjacency matrix of $G$ as above. To map this problem to parallel computation, we assume node $v_i$ stores $x_i$, $y_i$ and $A_{i,j}$ for all $j$ such that $A_{i,j} \neq 0$. The weight

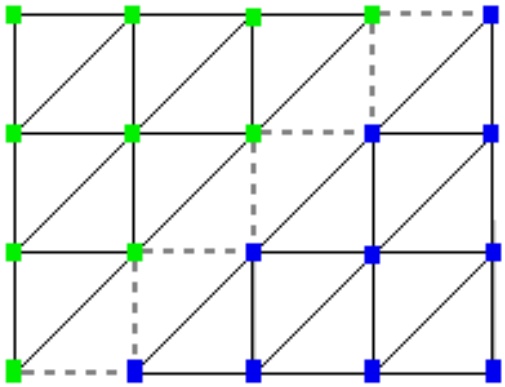


Figure 1: A partitioned graph

of node $i$ is the number of floating point multiplications needed to compute $y_i$, the weight of $e(i, j)$ represents the cost of sending data from node $j$ to node $i$.
The algorithm for computing $y = A \cdot x$ follows:

1. For each $i$, the processor owning node $i$ gets all $x_j$ for that $A_{i,j} \neq 0$. Iff $x_j$ is stored on another processor, get it from there.
2. For each $i$, the processor owning node $i$ computes $y_i = \sum_j A_{i,j} \cdot x_j$.

# 3 Spectral Partitioning

## 3.1 Motivation

The spectral partitioning algorithm is based on the intuition that the second lowest vibrational mode of a vibrating string naturally divides the string in half (Figure 2).

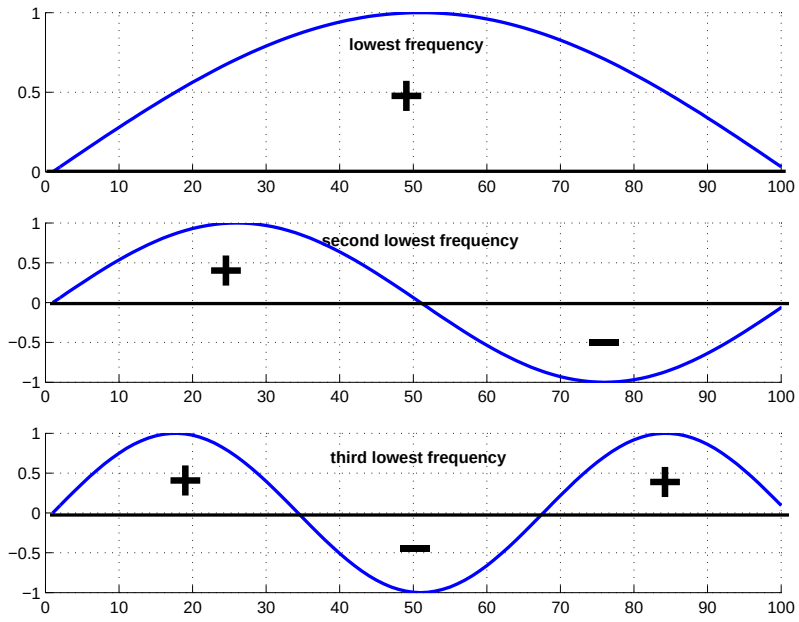



Figure 2: The frequences of vibrating string

Applying that intuition to the eigenvectors of a graph we obtain the partitioning. The lowest eigenvalue of the Laplacian matrix is zero and corresponding eigenvector of all ones provides no information about the graph structure. More interesting is the second eigenvector *(Fiedler vector)*. Dividing its elements according to the median we obtain two partitions (section 3.2.1). We can also use the third, fourth or bigger eigenvector to obtain three, four or more partitions but in fact there is always used only the second eigenvector.

All the algorithms mentioned in the following work with second eigenvector and deal with graph bisection, that means they divide the vector set $V$ into two disjoint sets: $V = V_1 \cup V_2$ that are equally large. To obtain more graph parts they could be applied recursively, until the parts are small and numerous enough. In fact, we try to find *an edge separator* $E'$ - the smallest subset of $E$ such that removing $E'$ from $E$ divides $G$ into two disconnected subgraphs $G_1$ and $G_2$, with nodes $V_1$ and $V_2$ respectively.

## 3.2 A spectral partitioning algorithm

### 3.2.1 Bisection

In this section, we describe how to search an edge separator of a graph $G$. Recall (see 2.2) we can get two parts of $G$ with nearly equals number of vertices and we also require the size of cutting edges to be small.

Let have a connected graph $G = (V, E)$ as in (2.2) and let $\vec{v} = (v_0, v_1, ..., v_n)$ be a second eigenvector - the Fiedler vector of the Laplacian matrix of $G$. In following, we always consider $G$ is connected thus multiplicity of the zero eigenvalue is one and the second eigenvalue is greater than zero. The idea of spectral partitioning is to find a *splitting value* $s$ such that we can partition vertices in G with respect to evaluation of the Fiedler vector. In fact, we create two sets of vertices, denote them $V_1$, $V_2$. The first set of vertices corresponds to $v_i \leq s$ and the second corresponds to $v_i > s$. We call such a partition the *Fiedler cut* (theorem 2.1.1) and we consider median of $v$ as $s$. If there is only one component equal to median then $V_1$, $V_2$ differ in size by at most one. If there are several components equal to median we arbitrarily assign such vertices to $V_1$ or $V_2$ to make these sets differ in size by at most one.

Let $V_1'$ denote the vertices in $V_1$ that are adjacent to some vertex in $V_2$, and similarly let $V_2'$ denote the vertices in $V_2$ that are adjacent to some vertex in $V_1$. Let $E_1$ be the set of edges with both end vertices in $V_1$, and let $E_2$ be the set of edges with both end vertices in $V_2$. Let $E' \in E$ be the set of edges of $G$ with one point in $V_1'$ and the second in $V_2'$. The $E'$ is an edge separator of $G$. Both $V_1'$ and $V_2'$ are vertex separators corresponding to the edge separator $E'$. Thus, to every edge separator, two vertex separators are naturally related. However, we will not work with vertex separators.

Now, we can set up an algorithm for spectral partitioning of graphs:

**Algorithm 3.2.1** (Spectral bisection)
Input: a graph $G = (V, E)$
Output: graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$

1. *compute the Fiedler eigenvector* $v$
2. *search median of* $v$
3. *for each node* $i$ *of* $G$
   - 3.1. *if* $v(i) \leq median$
     *put node* $i$ *in partition* $V_1$
   - 3.2. *else*
     *put node* $i$ *in partition* $V_2$
4. *if* $|V_1| - |V_2| > 1$ *move some vertices with components equal to median from* $V_1$ *to* $V_2$ *to make this difference at most one*
5. *let* $V_1'$ *be the set of vertices in* $V_1$ *adjacent to some vertex in* $V_2$
   *let* $V_2'$ *be the set of vertices in* $V_2$ *adjacent to some vertex in* $V_1$
   *set up the edge separator* $E'$ *- the set of edges of* $G$ *with one point in* $V_1'$ *and the second in* $V_2'$
6. *let* $E_1$ *be the set of edges with both end vertices in* $V_1$
   *let* $E_2$ *be the set of edges with both end vertices in* $V_2$
   *set up the graphs* $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$
7. *end*

### 3.2.2 K-way partitioning

Let us consider a connected graph $G = (V, E)$ as in section 2.2, $n = |V|$. K-way partitioning of $G$ is a division of its vertices into $k$ disjoint subset of size nearly equal to $n/k$. The main idea is to apply the spectral bisection as much as necessary, exactly until we have desired count of parts (say $k$).

There are two posibilities. When $k$ is power of two, we use recursive bisection in its simple form. When $k$ is not power of two, we have to use a modified recursive bisection. Let us consider the first case now.

**Algorithm 3.2.2** (Recursive bisection)
Input: a graph $G = (V, E)$; an integer $k$ (count of desired partitions)
Output: graphs $G_1 = (V_1, E_1)$,..., $G_k = (V_k, E_k)$

1. *apply* Spectral bisection(G) (3.2.1) *to find* $G_1$, $G_2$
2. *if* $(k/2 > 1)$
   - *2.1.* Recursive bisection($G_1$, k/2)
   - *2.2.* Recursive bisection($G_2$, k/2)
3. *return partitions* $G_1$, ... $G_k$
4. *end*

Now, we will look in detail on the situation when $k$ is not a power of two. Recall when we used the Fiedler vector for bisection, we used median as the splitting value. Now, we will use another quantil to get appropriate partitions.

**Algorithm 3.2.3** (Modified spectral bisection)
Input: a graph $G = (V, E)$; quantil (of desired size $|V_1|$)
Output: graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ of prescribed size $|V_1|$, $|V_2| = |V| - |V_1|$

1. *compute the eigenvector* $v$
2. *search quantil (%) of* $v$
3. *for each node* $i$ *of* $G$
   - *3.1. if* $v(i) \leq$ *quantil*
     *put node* $i$ *in partition* $V_1$
   - *3.2. else*
     *put node* $i$ *in partition* $V_2$
4. *5., 6. as in (3.2.1)*
7. *end*

**Algorithm 3.2.4** (Modified recursive bisection)
Input: a graph $G = (V, E)$; an integer $k$ (count of desired partitions)
Output: graphs $G_1 = (V_1, E_1)$,..., $G_k = (V_k, E_k)$ of nearly equal size $|V_i| - |V_j| = \pm 1$

1. *if* $(k > 1)$
   1.1. $k_1 = \lfloor k/2 \rfloor$
   $k_2 = k - k_1$
   *quantil* $= k_1/n$
   1.2. *apply* Modified spectral bisection(G, quantil) (3.2.3) *to find* $G_1$, $G_2$
   1.3. Modified recursive bisection($G_1$, $k_1$)
   Modified recursive bisection($G_2$, $k_2$)
   1.4. *return partitions* $G_1$, ... $G_k$
2. *else end*

### 3.2.3 Lanczos method

In this section, we will describe how to get the second eigenvalue and the second eigenvector of a graph. We do not need a particularly accurate answer because we are only going to use the pattern of the second eigenvector to perform the partitioning. As we will see in section 6.1, a rather loose or a very strict error tolerance on computing the second eigenvalue and the second eigenvector has no essential influence on a resultant partitioning.

Since $L$ is a sparse matrix in all practical applications, the algorithm most suitable for solving this problem is a Lanczos algorithm. Given a sparse symmetric matrix $L$ $(n \times n)$, Lanczos computes a symmetric tridiagonal matrix $T$ $(j \times j,)$, $j << n$, such that the eigenvalues of $T$ are good approximations to the extreme eigenvalues of $L$ and also eigenvectors of $T$ can be used to get approximate eigenvectors of $L$. The most expensive part of the Lanczos process is typically the sparse matrix-vector multiplication with $L$. Usually the smallest and largest eigenvalues of $L$ are approximated at first so we need to compute $j$ multiplications, but practically $j$ is much smaller than $n$.

In its simplest form, the Lanczos algorithm follows. This is essentially Algorithm 9.2.1. (The Lanczos Algorithm) from [7] according to C++ conventions -

vector indices start with zero instead of one, thus, we must involve some changes in steps order. We also consider $\epsilon$ as tolerance on desired accuracy to computation of eigenvectors. Input vector $v$ is randomly generated and normalized to having unit 2-norm. We assume the existence of a function `L.mult(v)` that returns a sparse matrix-vector multiplication $L \times v$.

**Algorithm 3.2.5** (Lanczos algorithm)
Input: a symmetric matrix $L \in \mathcal{R}^{n \times n}$; a vector $v \in \mathcal{R}^n$; $\epsilon$ as tolerance
Output: a symmetric tridiagonal matrix $T_j \in \mathcal{R}^{j \times j}$

1. *u = L.mult(v);*
2. *j = 0;*
3. *while* $|\beta[j]| > \epsilon$
   - 3.1. $\alpha[j] = v^T u$; $u = u - \alpha[j] \cdot v$; $\beta[j] = \|u\|_2$;
   - 3.2. *for i=0:n*
     $tmp = v[i]$; $v[i] = u[i]/\beta[j]$; $u[i] = -tmp \cdot \beta[j]$;
   - 3.3. *u = u + L.mult(v)*
   - 3.4. *j = j + 1;*
4. *end*

At each step $j$, the algorithm produces a tridiagonal matrix

$$T_j = \begin{bmatrix} \alpha[0] & \beta[0] & 0 & ... & 0 \\ \beta[0] & \alpha[1] & \beta[1] & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & ... & \beta[j-3] & \alpha[j-2] & \beta[j-2] \\ 0 & ... & 0 & \beta[j-2] & \alpha[j-1] \end{bmatrix}.$$

The eigenvalues of $T_j$, also called the Ritz values, are the Rayleigh-Ritz approximations to the eigenvalues of $L$ from the subspace spanned by the vectors $v$ arised in each iteration of the Lanzcos algorithm. Assign $V_j$ as matrix of those $v$.

In this form, the Lanczos algorithm delivers very good approximations to the large eigenvalues instead of converging to the desired second smallest eigenvalue. A practical implementation for numerical computing of the largest eigenvalues is described in [9]. Our implementation is based on this work. Since we are specificelly interesed in the second smallest eigenvalue we need to make several modifications. At first, we will use instead of $L$ a matrix $\widetilde{L} = -L$, comming out from rule the largest eigenvalue of $L$ is equal to the negative of the smallest eigenvalue of $-L$. Second, we use a reorthogonalization to get second smallest eigenvalue instead of the first eigenvalue which is zero. However, we do not need reorthogonalization in its full form in this case. At each step we explicitly orthogonalize the current Lanczos vector $v$ with given vector $e = (1, 1, .., 1)$ (*the first eigenvector*). In this way, we always stay in the subspace orthogonal to the trivial eigenvector $(1, 1, .., 1)$ and the eigenvalue zero is removed from the problem.

Let us assign the largest eigenvalue of $T_j$ as $\tau_j$. For better performance[1], we also involve finding a narrow interval which contains the second eigenvalue $\lambda_1$ and we also involve computing a bound on $|\tau_j - \lambda_1|$. The main idea is to solve an actual tridiagonal system $T_j$ in each step to preconditioning a right boundary of the interval containing $\lambda_1$ (left boundary is zero). In the second iteration of $j$, we compute the middle eigenvalue of $T_2$ which is used to precondition a narrow interval of the eigenvalue computed in next step (second eigenvalue of $T_3$) and so on until the stop condition is break. It comes out from the vibrating model (Figure 2).

To computing a bound on $|\tau_j - \lambda_1|$ we come out from the simple inequality

$$|\tau_j - \lambda_1| \leq \frac{\|Ay - y\tau_j\|}{\|y\|}$$

$y = V_j s$, $s$ is the normalized eigenvector of $T_j$ corresponding to $\tau_j$. After several modifications (see [9]) we get

$$|\tau_j - \lambda_1| \leq \beta_j \sigma_j$$

$\beta_j$ is given from the Lanczos process and $\sigma_j$ is the $j$-th component of the eigenvector of $T$.

Finally, the Lanczos algorithm looks as follows.

**Algorithm 3.2.6** (Modified Lanczos algorithm)
Input: a symmetric matrix $\widetilde{L} \in \mathcal{R}^{n \times n}$; a vector $v \in \mathcal{R}^n$; $\epsilon$ as tolerance
Output: symmetric tridiagonal matrix $T_j \in \mathcal{R}^{j \times j}$, $\lambda_1$ the second largest eigenvalue of $\widetilde{L}$

[1] described in detail in [9]

1. *$u = \widetilde{L}.mult(v)$;*
2. *$j = 0$;*
3. *while $|\beta[j]| > \epsilon$*
   - 3.1. *$\alpha[j] = v^T u$; $u = u - \alpha[j] \cdot v$; $\beta[j] = \|u\|_2$;*
   - 3.2. *for i=0:n*
     $tmp = v[i]$; $v[i] = u[i]/\beta[j]$; $u[i] = -tmp \cdot \beta[j]$;
   - 3.3. *$u = u + L.mult(v)$*
   - 3.4. *make reorthogonalization of $v$ using $e$*
   - 3.5. *$j = j + 1$;*
   - 3.6. *if $(j > 1)$*
     *solve tridiagonal system to get a narrow interval which contains second eigenvalue of $T_j$*
     *compute a bound on locality of second eigenvalue*
4. *solve tridiagonal system $T_j$ with preconditioning from last step to get the second eigenvalue of $L$*
5. *end*

For computing the second eigenvector of $L$, we at first have to search the second eigenvector $w$ of tridiagonal matrix $T_j$ by some appropriate method and second, we have to compute desired second eigenvector of $L$ from stored vectors ${v_i}^2$ (came from each step of the Lanczos algorithm) and given $w$ ($fiedler = \sum_{i=0}^{j} v_i w[i]$). In our program we are using an inverse iteration [7].

### 3.2.4 Complexity

Let have a look on Modified spectral bisection algortihm (3.2.3) again. There are several steps. The most expensive step of whole process is computing the eigenvector.

Since the Lanczos algorithm is an iterative algorithm, the number of Lanczos steps required to compute the second eigenvector will depend on the accuracy desired in

[2]Note the vectors $v_i$ are not sparse vectors so this procedure is the most space required part of whole algorithm.

the eigenvector. We assume the number of iterations of the Lanczos algorithm required to compute a second eigenvector to a small number of digits is bounded by a constant [10]. So each iteration of the Lanczos algorithm costs $O(const)$ flops and in each iteration we have to make one sparse matrix-vector multiplication in complexity $O(n)$.

Next time demanding step is median (quantil) computing. We are using an algorithm that selects $k$th smallest elements from $n$ elements without sorting array (`selip`, [12]). Selecting a random element, in one pass through the array, it moves smaller elements to the left and larger elements to the right. It is important to optimize the inner loop to minimize the number of comparisons. The general idea is to choose randomly a set of $m$ elements, to sort them and to count how many elements of array fall in each of $m + 1$ intervals defined by these elements. Next we choose the interval containing the quantil and make next round with $m+1$ intervals again until the quantil is localized in single array. The complexity of this algorithm is bounded as $O(n \cdot log(n))$.

The third step which divide graph into sets $V_1$, $V_2$ can be done in $O(n)$ time and the others last steps in the same time.

The investigation of complexity of recursive bisection could be found in the work by H. D. Simon and S. Teng, [13].

With increasing performance of computers, there decreases the problem of time complexity and arises the problem of space complexity. Let take an example of graph with one milion vertices, each with about six neighbours. Simple sparse adjacency matrix will takes about 40MB (in case of 4B for each nonzero value). Also we need to count with some additional storage for temporary matrix and vectors.

As we wrote in footnote below the 3.2.6 algorithm, there is problem with store the $v_i$ vectors to compute the second eigenvectors. The $v_i$ vectors are not sparse vectors so they take about 4MB each. In case of one hundred of Lanczos steps, it is 400MB in operating memory.

Fortunately, we can solve this problem with storing this vectors on hard disk memory and take them after for computing the second eigenvector. It causes the deceleration of computation but it enables to work with larger graphs. The exact results are in section 6.

# 4 Multilevel Partitioning

Although, the Lanczos algorithm is very suitable for solving a sparse matrix problem, there is a problem with the first step of partitionig - application of the Lanczos algorithm on whole graph is time expensive process. For improve performance is good idea to use some multilevel algorithm.

The main idea of the multilevel algorithm is to get a coarse approximation to the desired eigenvector and then to use another suitable iteration scheme to get a final eigenvector using coarse approximation from previous step as a reasonable starting vector.

Again, let us consider a connected graph $G = (V, E)$ as in section 2.2. We will replace the problem of partitioning the original graph $G = (V, E)$ by the simpler problem of partitioning a coarse approximation $G' = (V', E')$ to $G$. Given a partitioning of $G'$, we will use it to get a starting guess for a partitioning of $G$, and refine it by an iterative process. In general the multilevel algorithm looks as follows.

**Algorithm 4.0.7** (Multilevel algorithm)
Input: a graph $G = (V, E)$, $N$ as a maximum count of nodes of graph, that can be directly computed via spectral bisection (in a reasonable amount of time)
Output: graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$

1. *if* $|V| < N$
   *apply* Spectral bisection(G) (3.2.1) *to find* $G_1$, $G_2$
2. *else*
   - 2.1. *compute a coarse approximation* $G' = (V', E')$
   - 2.2. ($G'_1$, $G'_2$) = Multilevel algorithm($G'$)
   - 2.3. *expand* $G'_1$, $G'_2$ *to* $G_1$, $G_2$
   - 2.4. *improve partitions* $G_1$, $G_2$
3. *end*

There are several known algorithms to implement this algorithm effectively. We will describe a multilevel implementation by Barnard and Simon [2], which is based on the concept of maximal independent sets. As a suitable iteration scheme they use a Rayleight Quotient Iteration algorithm. The multilevel partitioning consist from three main steps:

- contraction
- interpolation
- refinement

Description of them follows.

## 4.1 Contraction

At first, we need to contract the original graph $G = (V, E)$ into approximate graph $G' = (V', E')$. This step corresponds to the step 2.1. in algorithm 4.0.7. We choose $V'$ to be a maximal independent set with respect to $G$. See figure 3.

Maximal independent set must hold three conditions:

- $V' \in V$
- no nodes in $V'$ are directly connected by edges in $E$ (independence)
- $V'$ is as large as possible (maximality)

After we have $V'$, we can construct the graph $G'$. The idea is to add vertices not in $V'$ to domains around vertices in $V'$ until all vertices not in $V'$ take part in some domain around each vertex in $V'$. The domain $D_i$ for each node $i \in V'$ just contain its neighbors. To constuct edges in $E'$ we add an adge to $E'$ whenever two domains intersect. The edges in $E'$ now simply connect nodes in $V'$ that are adjacent through another node not in $V'$.

## 4.2 Interpolation

The second step of the Multilevel algorithm is an interpolation and it corresponds to the step 2.3. in algorithm 4.0.7. The idea is to give a Fiedler vector of some contracted graph, to interpolate this vector to the next larger graph and to use this vector to provide a good approximation to the next Fiedler vector.

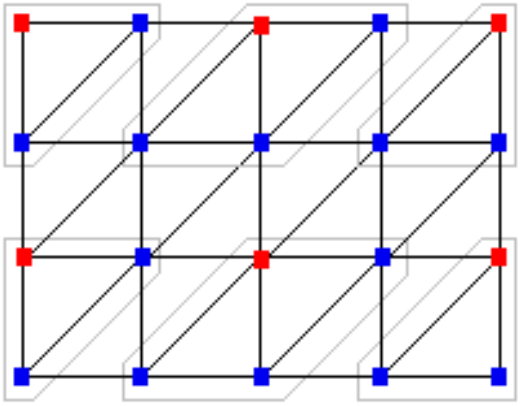


Figure 3: A maximal independent set

Let us consider a Fiedler vector $f' = (f(0), f(1), ...f'(n))$, $n' = |V'|$ of contracted graph $G'$. We will construct an expansion of this vector to the next larger graph $G$ $\widetilde{f} = (\widetilde{f}(0), \widetilde{f}(1), ...\widetilde{f}(n))$, $n = |V|$ that we will use as an approximation for the Fiedler vector of original graph. Interpolation consists of two steps: injection and averaging. The algorithm is briefly described as follows.

**Algorithm 4.2.1** (Interpolation algorithm)
Input: a vector $f' = (f(0), f(1), ...f'(n))$, $n' = |V'|$
Output: vector $\widetilde{f} = (\widetilde{f}(0), \widetilde{f}(1), ...\widetilde{f}(n))$, $n = |V|$

1. *for each node* $i \in V$

   1.1. *if* $i \in V'$
   $\widetilde{f}(i) = f'(i)$
   *... interpolation*

   1.2. *else*
   $\widetilde{f}(i) = \frac{\sum_{j \in U(i)} f'(j)}{|U(i)|}$, *where* $U(i)$ *are all neighbours of node* $i$ *in* $V'$
   *... averaging*

2. *end*

## 4.3 Refinement

The last step of the Multilevel algorithm corresponds to the step 2.4. in algorithm 4.0.7 and it involves refining of an approximate second eigenvector to be more accurate. Here could be used the Lanczsos algorithm. In case we have good initial approximation of a Fiedler vector, it is better to use the Rayleight Quotient Iteration algorithm which takes usually a few steps to convergence.

We present the algorithm from [7], where $\rho \equiv \frac{x^T A x}{\|x^T x\|}$ is the Rayleigh quotient.

**Algorithm 4.3.1** (Rayleight Quotient Iteration algorithm)
Input: a vector $\widetilde{f} = (\widetilde{f}(0), \widetilde{f}(1), ...\widetilde{f}(n)),\ n = |V|$, a Laplacian matrix $L$ of graph $G$
Output: vector $f = (f(0), f(1), ...f(n)),\ n = |V|$

1. $v_0 = \widetilde{f}$
2. $v_0 = \frac{v_0}{\|v_0\|}$
3. *i=0*
4. *repeat*
   - *4.1.* $i = i+1$
   - *4.2.* $\rho_i = v_{i-1}^T L v_{i-1}$
   - *4.3. solve* $v_i = (L - \rho_i I)x \ \ for \ \ x$
   - *4.4.* $v_i = \frac{x}{\|x\|}$

   *until convergence*
5. $f = v_i$
6. *end*

# 5 Disconnected graphs

In previous sections, we considered the connected graph $G = (V, E)$. Remark[3] that only connected graphs have the second smallest eigenvalue greater than zero and the relevant eigenvector is the Fiedler vector. In case we would try to apply the previous algorithms to a disconnected graph, they will not work well because there would arise multiplicity of the zero eigenvalue.

To work with disconnected graphs, we include several changes. The main idea is to find all disonnected components, to use some partitioning algorithm for an appropriate component and to assign remaining components into proper parts.

Let us have a look on figure 4. There is a disconnected graph that consists from three components. Partitioning of this graph in two equal size parts requires to divide only the biggest component appropriately. Still, we require two conditions from section 2.2 to be hold. According to these conditions, we have well-partitioned graph. The green nodes belong to the first part and the blue nodes belong to the second part. Note we had to divide only one component so the second part is still connected.

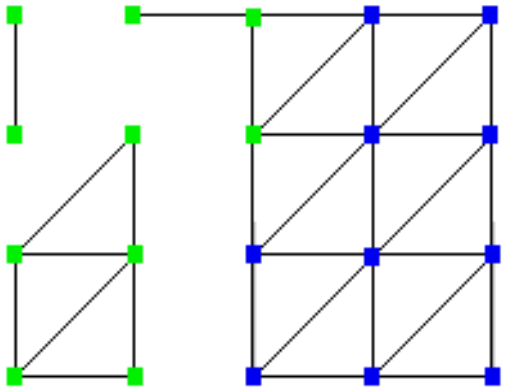


Figure 4: An example of disconnected graph patitioning

It is clear that in each case we need to split only one of all components. We can also consider a tolerance to count of nodes in each partition. Thus, we don't have to make partitioning if count of nodes in smaller part differs from ideal half less then

[3] see section 2.1

tolerance. Let us call this tolerance as *BP tolerance* (Balance of Parts tolerance). The algorithm for disconnected graphs follows.

**Algorithm 5.0.2** (Disconnected graphs partitioning)
Input: a graph $G = (V, E)$, BP tolerance $\Delta$
Output: graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$

1. *find components of graph; let mark them* $C_0 = (V_{C_0}, E_{C_0})$, ..., $C_k = (V_{C_k}, E_{C_k})$
2. *if count of component equal one*
   *apply* Spectral bisection(G) (3.2.1) *to find* $G_1$, $G_2$
3. *else*
   - 3.1. *sum = 0*
   - 3.2. *for i=0..k*
     *sum = sum +* $|V_{C_i}|$
     *if (sum >* $|V|/2$*)*
     *break*
   - 3.3. *quantil = sum* $- |V|/2$
   - 3.4. *if (quantil* $< \Delta$*)*
     *move* $C_0$, ..., $C_{i-1}$ *to* $G_1$ *and* $C_i$, $C_{i+1}$, ..., $C_k$ *to* $G_2$
     *else if ((*$V_{C_i}$ *- quantil)* $< \Delta$*)*
     *move* $C_0$, ..., $C_{i-1}$, $C_i$ *to* $G_1$ *and* $C_{i+1}$, ..., $C_k$ *to* $G_2$
   - 3.5. *else*
     *apply* Modified spectral bisection($C_i$, quantil) (3.2.3) *to find* $C_{i1}$, $C_{i2}$
     *move* $C_0$, ..., $C_{i-1}$, $C_{i1}$ *to* $G_1$ *and* $C_{i2}$, $C_{i+1}$, ..., $C_k$ *to* $G_2$
4. *end*

We consider partitioning in its simplest form, it means we will get two partitions nearly equal size (up to tolerance). Proportional or recursive partitioning can be obtained similarly. It remains to describe the step *1.*. An appropriate algorithm to find connected components is a Breadth First Search algorithm.

## 5.1 Breadth First Search

Let a graph $G = (V, E)$ be an undirected, unweighted graph without loops or multiple edges from one node to another. Let choose some node $v_r$ from $V$ and let call it the root. The Breadth first search algorithm in its general form produces a subgraph $T$ of $G$ on the same set of nodes, where $T$ is a tree with root $v_r$. In addition, it associates a level with each node $v_i$ from $V$, which is the number of edges on the path from $v_r$ to $v_i$ in $T$.

For our case, it suffices that the Breadth first search algorithm is able to search all nodes that are available from $v_r$. For getting all connected components, we have to keep an information about visiting nodes and to aply the Breadth first search algorithm until all nodes are visited.

**Algorithm 5.1.1** (Breadth First Search)
Input: a graph $G = (V, E)$
Output: list of connected components $C_0 = (V_{C_0}, E_{C_0})$, ..., $C_k = (V_{C_k}, E_{C_k})$

*1. $k = 0$*

*2. until all vertices visited*

*2.1. choose an unvisited vertex $v_r$*

*2.2. add $v_r$ to a new component $C_k$*
*$k = k + 1$*
*add $v_r$ to the stack*
*mark $v_r$ as visited*

*2.3. until stack is empty*

*2.3.1. get w from stack*

*2.3.2. for all unvisited neighbours of w $n_i$*
*add $n_i$ to stack*
*add $n_i$ to actual component*
*mark $n_i$ as visited*

*2.4. assign all edges between vertices in $V_{C_i}$ to $E_{C_i}$*

*3. return list of components $C_0 = (V_{C_0}, E_{C_0})$, ..., $C_k = (V_{C_k}, E_{C_k})$*

*4. end*

This implementation of the Breadth first search algorithm requires a data structure called a Stack, or a FIFO (First In First Out) list. There are two available operations: adding and getting. The adding operation adds an object to the top of stack and the getting operation removes the top object from the stack and returns it to use.

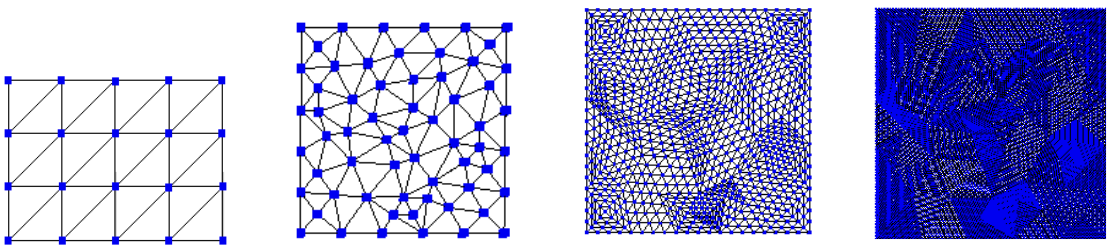
# 6 Test examples and results

In this section, we show some practical resuls of written software. Tests were made on CPU AMD Athlon 64 3000+, RAM 1024MB, Win XP Professional, Microsoft Visual C++ 7.0 compiler. Our testing set consists from graphs that represent two or three dimensional computational grid of some numerical problem. We have worked with four two-dimensional graphs (`mesh, square1, square2, square3`) and with four three-dimensional graphs (`cube1, cube2, cube3, cube4`). There is also one graph that represent a disconnected graph (`athlete`). Later, we introduce several bigger graphs for testing maximal performance.

There are several parametres to observe. We can choose a *number of partition*, a *Lanczos tolerance* (understand tolerance on precision in the Lanzcos algorithm) or a *maximum of Lanczos steps* (understand how many iterations of the Lanczos algorithm we allow) - see section 3.2.3. For the maximum of Lanczos steps, we will use the *maxit* abbreviation. In case of disconnected graphs, there is possible to choose the *BP tolerance* - see section 5. We will interest especially in computational time and count of divided edges in sense of the edge separator $E'$ from section 3.2.1. According to section 3.2.1, we denote $V$ as a vertex set and $E$ as an edge set. $|V|$ denotes count of vertices, $|E|$ denotes count of edges.

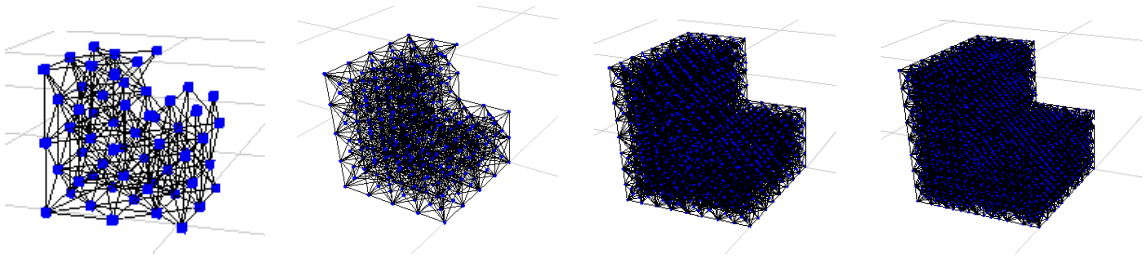
## 6.1 Connedted graphs

On figure 5, we can see two-dimensional graphs and on figure 6, we can see three-dimensional graphs that we have used for testing purposes.



(a) `mesh` (b) `square1` (c) `square2` (d) `square3`

Figure 5: Two-dimensional graphs



(a) `cube1` (b) `cube2` (c) `cube3` (d) `cube4`

Figure 6: Three-dimensional graphs

At first, we have tested time demands. Table 1 shows the resuls with using *number of partition = 2* and *Lanczos tolerance = 0.001*.

Table 2 shows the results on large three-dimensional graphs. Again, we have used the *number of partition = 2* and *Lanczos tolerance = 0.001*. All this graphs represent cubic grid with maximal degree of vertex equal six. As we have wrote in section 3.2.4, the main limitation is the space memory. For better idea, see column 7 in the table. The first number means the count of bytes allocated at the beginning in virtual memory and the second number means the maximum of bytes used during computation. Since `large_cube3` we have to use swapp (on hard disk memory) so the size decreased. We can also see that the only one limitation is the size of operating memory (or the size of addressable space). On figure 7, graphs are joined for better illustration[4].

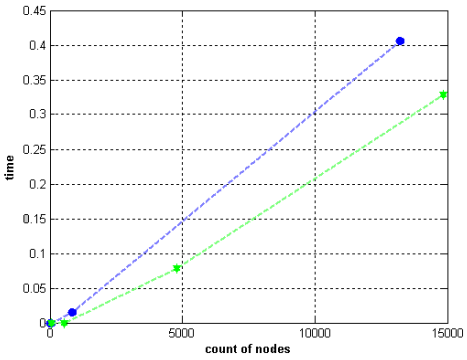
| NAME | dim | $\|V\|$ | $\|E\|$ | time[s] | Lanczos steps | $\|E'\|$ | $\|E'\|$ [% of $\|E\|$] |
|---|---|---|---|---|---|---|---|
| `mesh` | 2D | 20 | 43 | 0 | 12 | 7 | 16,28% |
| `square1` | 2D | 62 | 163 | 0 | 20 | 17 | 10,43% |
| `cube1` | 3D | 63 | 257 | 0 | 17 | 43 | 16,73% |
| `cube2` | 3D | 522 | 2522 | 0 | 31 | 201 | 7,97% |
| `square2` | 2D | 857 | 2488 | 0,015 | 56 | 63 | 2,53% |
| `cube3` | 3D | 4784 | 24836 | 0,078 | 70 | 896 | 3,61% |
| `square3` | 2D | 13217 | 39328 | 0,406 | 148 | 262 | 0,67% |
| `cube4` | 3D | 14836 | 79261 | 0,328 | 105 | 1953 | 2,46% |

Table 1: Software performance in dependence on count of nodes

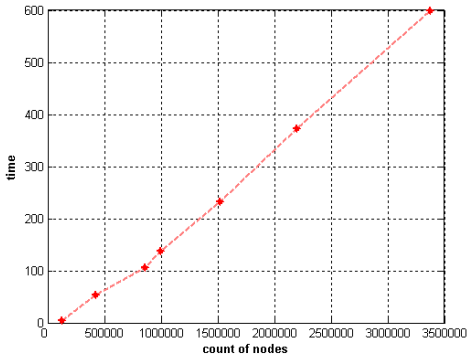
[4]We have also made measurement of participation of the Lanczos algorithm computation and the quantil computation on whole time requirements. The results in arithmetic mean: 95,41% takes Lanczos and 0,08% takes quantil (without table).

| NAME | $\|V\|$ | $\|E\|$ | time[s] | L. steps | $\|E'\|$ | VM size .. max size |
|---|---|---|---|---|---|---|
| `large_cube1` | 125000 | 367500 | 4 | 120 | 4669 | 18 MB .. 146 MB |
| `large_cube2` | 421875 | 1248750 | 54 | 160 | 9565 | 59 MB .. 670 MB |
| `large_cube3` | 857375 | 2545050 | 106 | 199 | 18317 | 88 MB .. 260 MB |
| `large_cube4` | 1000000 | 2970000 | 137 | 197 | 19133 | 101 MB .. 315 MB |
| `large_cube5` | 1520875 | 4522950 | 233 | 201 | 23483 | 149 MB .. 463 MB |
| `large_cube6` | 2197000 | 6540300 | 372 | 217 | 31399 | 214 MB .. 670 MB |
| `large_cube7` | 3375000 | 10057500 | 599 | 242 | 41405 | 325 MB .. 1048 MB |

Table 2: Software performance in dependence on count of nodes - large graphs




(a) `squares (blue), cubes (green)`




(b) `large_cubes`

Figure 7: Dependence of time on count of nodes

Now, we show the influence of tolerance in the Lanzcos algorithm on final quality of partitioning. For presentation, we have chosen the `square2` graph. Results are in table 3 and on figure 8. Tolerance $10^{-4}$ is a sufficient choise to get an optimal edge cut $|E'| = 63$. In case of two parts, it is evident that the increasing of precision of tolerance has only small influence on count of divided edges as we assumed in section 3.2.3. Different situation arises in case of four parts. There is a turn in points [$10^{-3}$, 129] and [$10^{-2}$, 126]. Explanation of this situation is easy. In k-way partitioning there should hold the conditions of optimality in each level separately but in global view there could exists some better partitioning which is not optimal in lower levels but which is optimal in global view. Specifically, the optimal four-cut $|E'| = 126$ corresponds to suboptimal two-cut $|E'| = 67$. See figure 9, 10.

| | TWO PARTS | | FOUR PARTS | |
|---|---|---|---|---|
| tolerance | Lanczos steps | $\|E'\|$ | Lanczos steps | $\|E'\|$ |
| $10^{-6}$ | 103 | 63 | 252 | 133 |
| $10^{-5}$ | 92 | 63 | 229 | 133 |
| $10^{-4}$ | 77 | 63 | 188 | 134 |
| $10^{-3}$ | 56 | 64 | 155 | 129 |
| $10^{-2}$ | 31 | 67 | 89 | 126 |
| $5 \cdot 10^{-2}$ | 18 | 87 | 56 | 182 |
| $10^{-1}$ | 14 | 103 | 40 | 216 |

Table 3: Dependence of count of divided edges on the Lanczos tolerance in `square2`

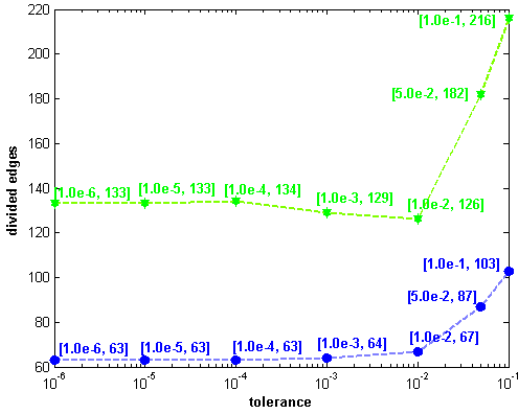
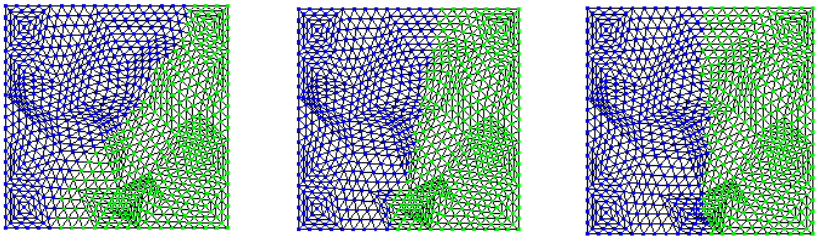
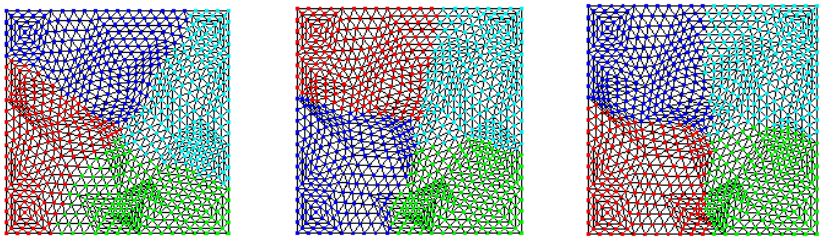



Figure 8: Dependence of count of divided edges on the Lanczos tolerance in `square2`



(a) tolerance=$10^{-4}$, $|E'| = 63$ (b) tolerance=$10^{-3}$, $|E'| = 64$ (c) tolerance=$10^{-2}$, $|E'| = 67$

Figure 9: `square2` in two parts



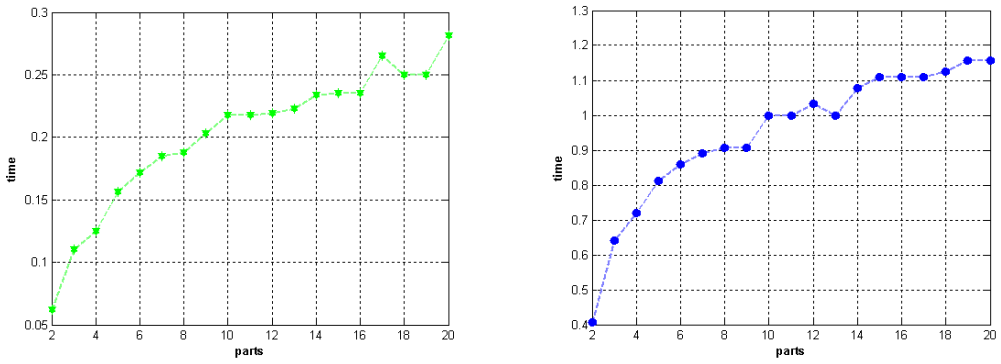
(a) tolerance=$10^{-4}$, $|E'| = 134$ (b) tolerance=$10^{-3}$, $|E'| = 129$ (c) tolerance=$10^{-2}$, $|E'| = 126$

Figure 10: `square2` in four parts

At the end of this subsection, let us have a look on a dependence of time on number of desired partitions. It is expectable that time grows with number of partitions but the progress in time is reduced because in higher levels smaller matrices need to be computed. Let us have a look in table 4 and on figure 11. Note the turns in points 8, 16, etc. ($2^k$, $k = 1, 2, ...$ generally).

| | cube3 | | square3 | |
|---|---|---|---|---|
| parts | time[s] | $\|E'\|$ | time[s] | $\|E'\|$ |
| 2 | 0,062 | 896 | 0,406 | 262 |
| 3 | 0,11 | 1253 | 0,641 | 373 |
| 4 | 0,125 | 1834 | 0,719 | 505 |
| 5 | 0,156 | 2199 | 0,812 | 655 |
| 6 | 0,172 | 2927 | 0,86 | 748 |
| 7 | 0,185 | 2890 | 0,891 | 861 |
| 8 | 0,188 | 3212 | 0,907 | 1052 |
| 9 | 0,203 | 3472 | 0,907 | 1023 |
| 10 | 0,218 | 3951 | 1 | 1241 |
| 11 | 0,218 | 3957 | 1 | 1272 |
| 12 | 0,219 | 4182 | 1,032 | 1372 |
| 13 | 0,223 | 4441 | 1 | 1364 |
| 14 | 0,234 | 4720 | 1,078 | 1398 |
| 15 | 0,235 | 4896 | 1,109 | 1543 |
| 16 | 0,235 | 5184 | 1,109 | 1586 |
| 17 | 0,265 | 5673 | 1,11 | 1626 |
| 18 | 0,25 | 5327 | 1,125 | 1709 |
| 19 | 0,25 | 5606 | 1,156 | 1783 |
| 20 | 0,281 | 5717 | 1,157 | 1822 |

Table 4: Depencence of time on number of partitions

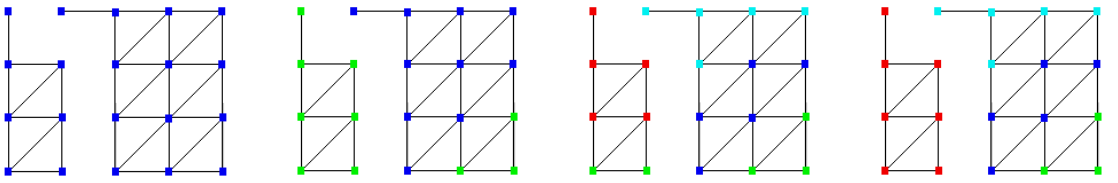



(a) cube3 (b) square3

Figure 11: Depencence of time on number of partitions

## 6.2 Disconnected graphs

The behaviour of disconnected graphs is a little different. Simple demonstration is on figure 12. The subfigure (a) shows us whole mesh. On subfigure (b), there are results of first partitioning. On subfigure (c), mesh is partitioned into three equal parts. One of them is disconnected (green nodes). Sometimes, it could be better to reduce the condition on equality of parts to get "more suitable" results. So on subfigure (d), we have included some tolerance[5] (*BP tolerance*) on count of nodes in each partition and we have got all resultant parts connected.



(a) whole mesh (b) two parts (c) four parts (d) four parts with tolerance

Figure 12: Disconnected mesh

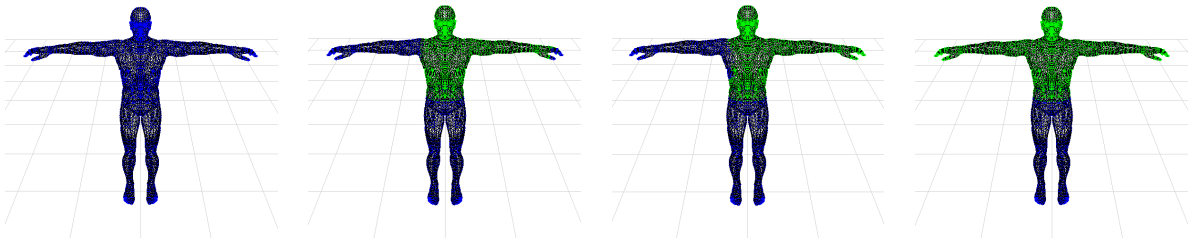
To testing, we have chosen graph of surface of human body, see figure 13 (a). It consists from six disconnected parts. Right leg (1), left leg (2), swimming trunks (3), body with hands and head (4), right eye (5) and left eye (6). We have observed primarily the influence of *BP tolerance* on resultant quality of partitioning. Results are in table 5 and a view of graphical output is on figures 13, 14.

The detail view of this figures is in appendix C.

[5]We have chosen tolerance = 10%. It means that counts of nodes in each part can differ from ideal third in 10% of all nodes (see section 5). In this case 10% are two nodes. It is too hight for standard computations but in this case it is acceptable.

| parts | BP tolerance | Lanczos tolerance | total Lanczos steps | $\lvert E' \rvert$ | time[s] |
|---|---|---|---|---|---|
| 2 | 0 | $2 \cdot 10^{-3}$ | 122 | 127 | 0,109 |
| 2 | 0 | $10^{-3}$ | 138 | 110 | 0,14 |
| 2 | 15% | $2 \cdot 10^{-3}$ | 0 | 0 | 0 |
| 4 | 0 | $10^{-3}$ | 354 | 297 | 0,25 |
| 4 | 5% | $10^{-3}$ | 274 | 249 | 0,234 |
| 4 | 15% | $10^{-3}$ | 138 | 162 | 0,141 |
| 4 | 15% | $10^{-5}$ | 244 | 248 | 0,234 |

Table 5: Dependence of BP tolerance on resultant quality of partitioning



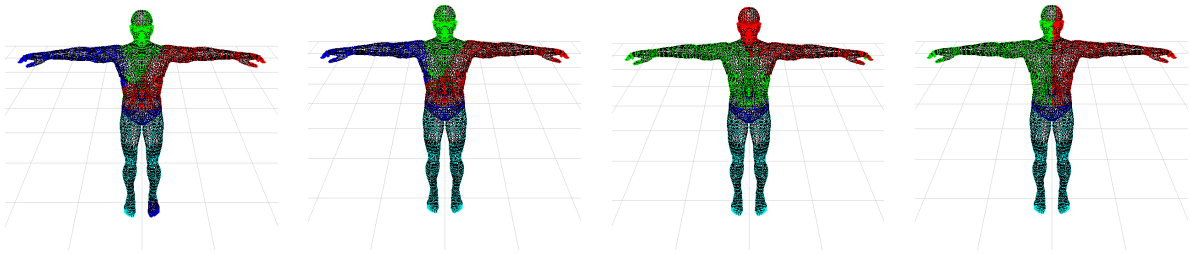
(a) whole graph

(b) BP_tolerance=0%, Lanczos_tol.=$2 \cdot 10^{-3}$

(c) BP_tolerance=0%, Lanczos_tol.=$10^{-3}$

(d) BP_tolerance=15%, Lanczos_tol.=$2 \cdot 10^{-3}$

Figure 13: `athlete` - the whole graph and two parts



(a) BP_tolerance=0%, Lanczos_tol.=$10^{-3}$

(b) BP_tolerance=5%, Lanczos_tol.=$10^{-3}$

(c) BP_tolerance=15%, Lanczos_tol.=$10^{-3}$

(d) BP_tolerance=0%, Lanczos_tol.=$10^{-5}$

Figure 14: `athlete` - four parts

# 7 Conclusion

Spectral methods for graph partitioning have been known to be robust but computationally expensive. We have presented appropriate algorithms to compute partitioning in their simple forms and we have also introduced several modifications to compute partitioning effectively.

We have also investigated various strategies how to achieve an optimal cut in a reasonable time. The set of test matrices was used to test the influence of several parameters to resultant quality of partitioning. There were also used some large graphs to find the limitation of written software.

We have found out the only one limitation is the size of operating memory that can be partially solve by appropriate datas storing on hard disk memory. There is also the possibility to use another store structure in the program to decrease the space demandingness.

On the other hand, we have also brought a new case for visualisation of graph partitioning.

# References

[1] W. N. Anderson and T. D. Morley, *Eigenvalues of the Laplacian of a graph*, Linear and Multilinear Algebra, 18:141-145, 1985

[2] S. T. Barnard and H. D. Simon, *A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems*, Concurrency: Practice and Experience, 6(2):101-117, 1992

[3] J. Demmel, *http://www.cs.berkeley.edu/∼demmel/cs267/*, University of California at Berkeley, 1996

[4] F. R. K. Chung, *Spectral Graph Theory*, Regional Conference Series in Mathematics, 92:0160-7642, 1994

[5] M. Fiedler, *Algebraic connectivity of graphs*, Czech. Math. J., 23(98):298-305, 1973

[6] M. Fiedler, *A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory*, Czech. Math. J., 25(100):619-633, 1975

[7] G. H. Golub and C. F. Loan, *Matrix Computation*, John Hopkins University Press, Baltimore, MD, Second edition, 1989

[8] B. Mohar, *The Laplacian spectrum of graphs*, Preprint Series Dept. Math. University E. K. Ljubljana 26, 261, 1988

[9] B. N. Parlett, H. Simon and L. M. Stringer, *On Estimating the Largest Eigenvalue With the Lanczos Algorithm*, Math. Comp., 38(157):153-165, 1982

[10] A. Pothen, H. D. Simon and K. Liou, *Partitioning Sparse Matrices with Eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11(3):430-452, 1990

[11] R. Pozo, K. Remington and A. Lumsdaine, *SparseLib++ Sparse Matrix Class Library, http://math.nist.gov/sparselib++/*, 1996

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipiens in C*, Cambridge University Press, Second edition, 1992

[13] H. D. Simon and S.-H. Teng, *How good is Recursive Bisection*, SIAM J. Sci. Comput., 18(5):1436-1445, 1997

[14] D. A. Spielman and S.-H. Teng, *Spectral Partitioning Works: Planar graphs and finite element meshes*, Technical Report UCB//CSD-96-898, University of Berkeley, 1996

# A Supported file formats

We can use two formats. As we use the SparseLib++ package by [11] for mathematical computations, we use the compress column format from there. Second, we can use the adjacency list format - in this structure data are stored in program. Input and output are possible in both of them and also conversions between them are implemented. With both formats is an information about coordinates connected. The information is stored into the coordinations file.

To illustrate both formats we will use graph on figure 15.

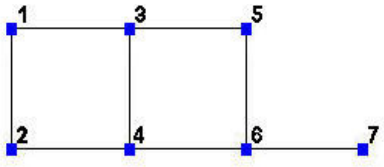



Figure 15: An example for file formats presentation

Adjacency matrix of this graph sees as follows:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

## A.1 Adjacency list format

Adjacency list file is a text file. The first number represents count of vertices and the second number represents count of edges[6]. Each next line describes a node and its adjacent edges. Indexing starts from *one* (to compatibility with e.g. Metis).

[6]In fact, if you don't know the count of edges nothing happends because we need it only for information purposes

Adjacency list is terminated by the end of the line.

Adjacency list file

```
7 16
2 3
1 4
1 4 5
2 3 6
3 6
4 5 7
6
```

## A.2 Compressed column format

The compressed column format consists from three vectors files and one dimensions file. They are all stored as text files. Indexing starts from *zero*. In compressed column format each column is understood as a sparse vector. Pointers to the first element in each column are stored in column pointer file (row vector), nonzero values and their associated row indices are stored in the nonzero values file and row index file (column vectors). An aditional element is appended to the column pointer file specifying the number of nonzero elements. See table 6.

| column pointer | 0 | 2 | 4 | 7 | 10 | 12 | 15 | 16 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| row indices | 1 | 2 | 0 | 3 | 0 | 3 | 4 | 1 | 2 | 5 | 2 | 5 | 3 | 4 | 6 | 5 |
| nonzero values | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 6: The compressed column format

The dimensions file contains three numbers in one row. First number is a count of rows, second is a count of columns and third is a count of nonzero values. See table 7.

| dimension | 7 | 7 | 16 |
|---|---|---|---|

Table 7: The compressed column format - dimensions file

## A.3 Coordinates

For graphical purposes, we need coordinates more. Coordinates are stored in separated text file, three coordinates for each node on line. There are possible both positive and negative values.

Coordinate file

```
0 1 0
0 0 0
1 1 0
1 0 0
2 1 0
2 0 0
3 0 0
```

# B User documentation

Program was developed under Microsoft Visual C++ with MFC and OpenGL support. The source codes and the testing set of graphs are enclosed on CD. To launch the program you need MFC libraries in your computer. Program starts with opening `graph_partitioning.exe` file.

Program consists from two windows. First, there is a main window for work with graphs and second, there is a console window in which we can see the results of program. We can see the main window on figure 16.

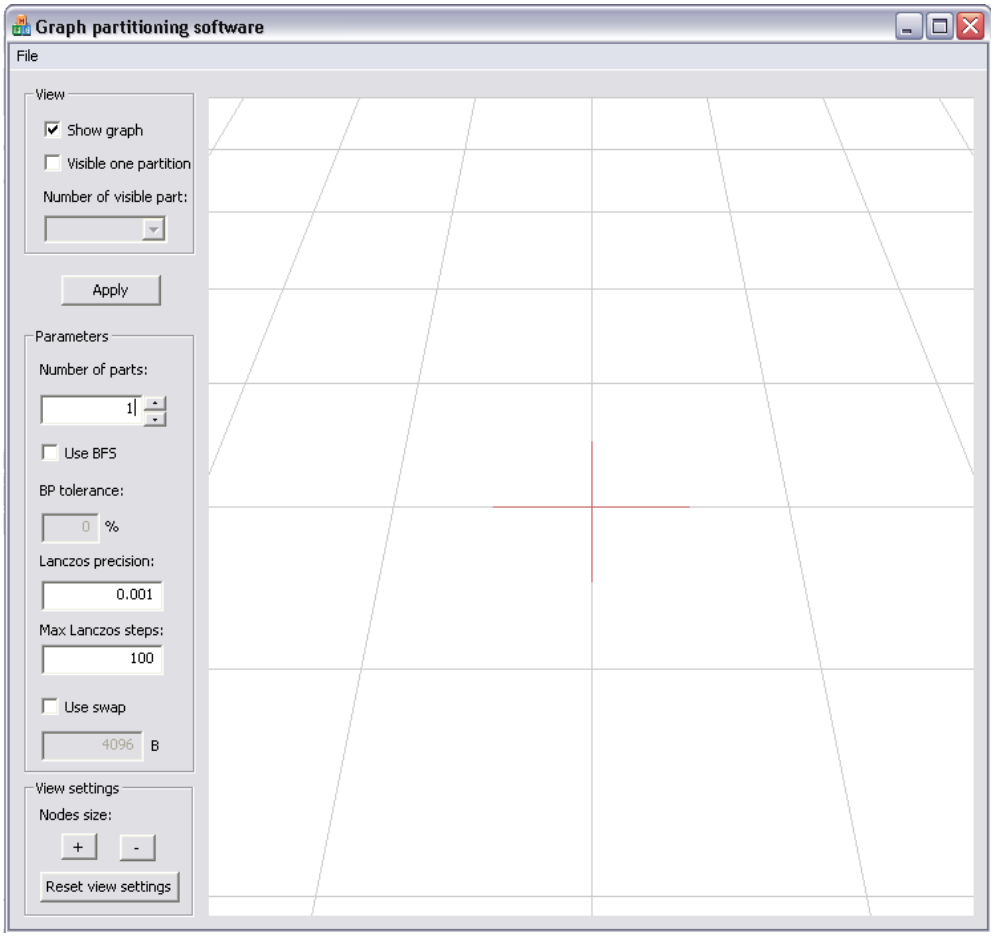



Figure 16: The main window

## B.1 The main menu

At first, let us have a look on the menu *File*. It provides functions *Open adjacency list format files* and *Open compressed column format files* for reading graph, function *Export partitioned graph* for export graph and the *Exit* function.

The *Open adjacency list format files* dialog is on figure 17. There are boxes for typing an Adjacency list file and a Coordinates file[7]. You can also choose them from directory. Default directory is the starting directory of the program. Adjacency list format is described in A.1.

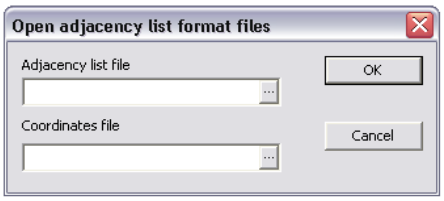



Figure 17: The Open adjacency list format files dialog

The *Open compressed column format files* dialog is on figure 18. Similarly, there are boxes for typing a Dimensions file, a Column pointer file, a Row index file, a Values file and a Coordinates file. You can also choose them from directory. Default directory is the starting directory of the program. The compressed column format is described in A.2.

On figure 19, there is the *Export divided graph* dialog. There are several possibilities how to export graph. First choice is the Adjacency list format. In this case there arises one file for each part with appendix `*_i.txt`, $i$ is index of partition. Indexing of vertices stays the same as in whole graph with blank rows in places of vertices from another parts. Second choice is the Reindexed adjacency list format. It differs from previous in system of indexing. It starts from one for each part and there are no blank rows. Last choice is the Compressed column format. In this case there arise compressed column format files for each part. They have appendixs `*_i_dim.txt` for the dimensions file, `*_i_colptr.txt` for the column pointer file, `*_i_rowptr.txt` for the row index file and `*_i_nzerovals.txt` for the nonzero values file, $i$ is index of partition. Indexing of vertices starts from zero for each part.

[7]In both formats (adjacency list format and compressed column format), you don't need to read the coordinates files. In this case you can work with graphs without visualisation. It is good choise in case of large graphs.

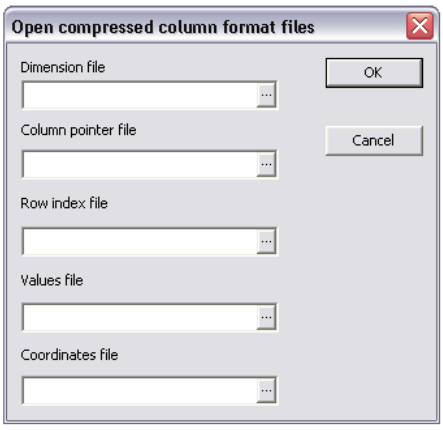



Figure 18: The Open compressed column format files dialog

If you choose to export coordinates, there arises one file for each part with appendix `*_i_coords.txt` in each choise of format.

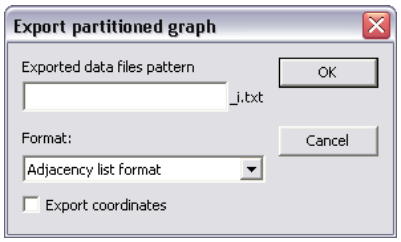



Figure 19: The Export divided graph dialog

## B.2 Software functions

Now, let us have a look on software functions in left box of the main window on figure 16.

First, there is the *View* functions group. The first check box enable to show or hide graphical visualisation of graphs. It is important if you can work with large graphs - the graphical output require quite a bit of operating memory. The second check box enable to view only one part from partitioned graph. You can choose it in list box below.

Second, there is the *Parameters* group. First parameter is the *Number of parts*. Here you can choose a count of desired parts as an integer value. Second there is

the *Use BFS* check box for enable/disable a Breath first search. In fact, it is always used if you are working with a disconnected graph. It provides an information about count of disconnected components. If it is enabled you can choose a *BP tolerance* (Balance of Parts tolerance). For explaining see section 5. The edit boxes for setting a *Lanczos precision* (understand tolerance on precision in the Lanzcos algorithm) and a *Max Lanczos steps* (understand how many iterations of the Lanczos algorithm we allow) follow. For explaining see section 3.2.3. The last parameter from this group is the *Use swap* check box. It should be enabled during work with large graphs. It enable to store some information from operating memory to hard disk memory. You can also choose the size of swap buffer. The button *Apply* above starts partitioning with chosen parameters.

Last group is the *Viw settings* functions group. First item *Nodes size* enable to change size of nodes in visible graph. Second item is the *Reset view settigs*, that reset all view settings to original (it means nodes size, move and rotation of graph).

## B.3 The console window

The console window provides us the informations about partitioning. The typical view is on figure 20. There are results of partitioning of the `square2` graph into two parts[8].

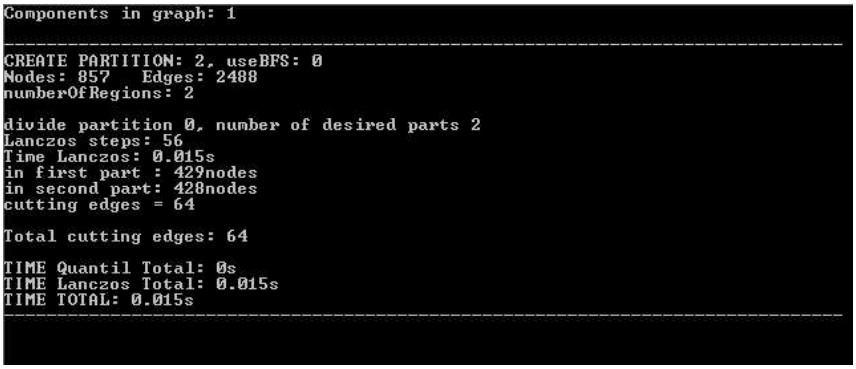



Figure 20: The console window

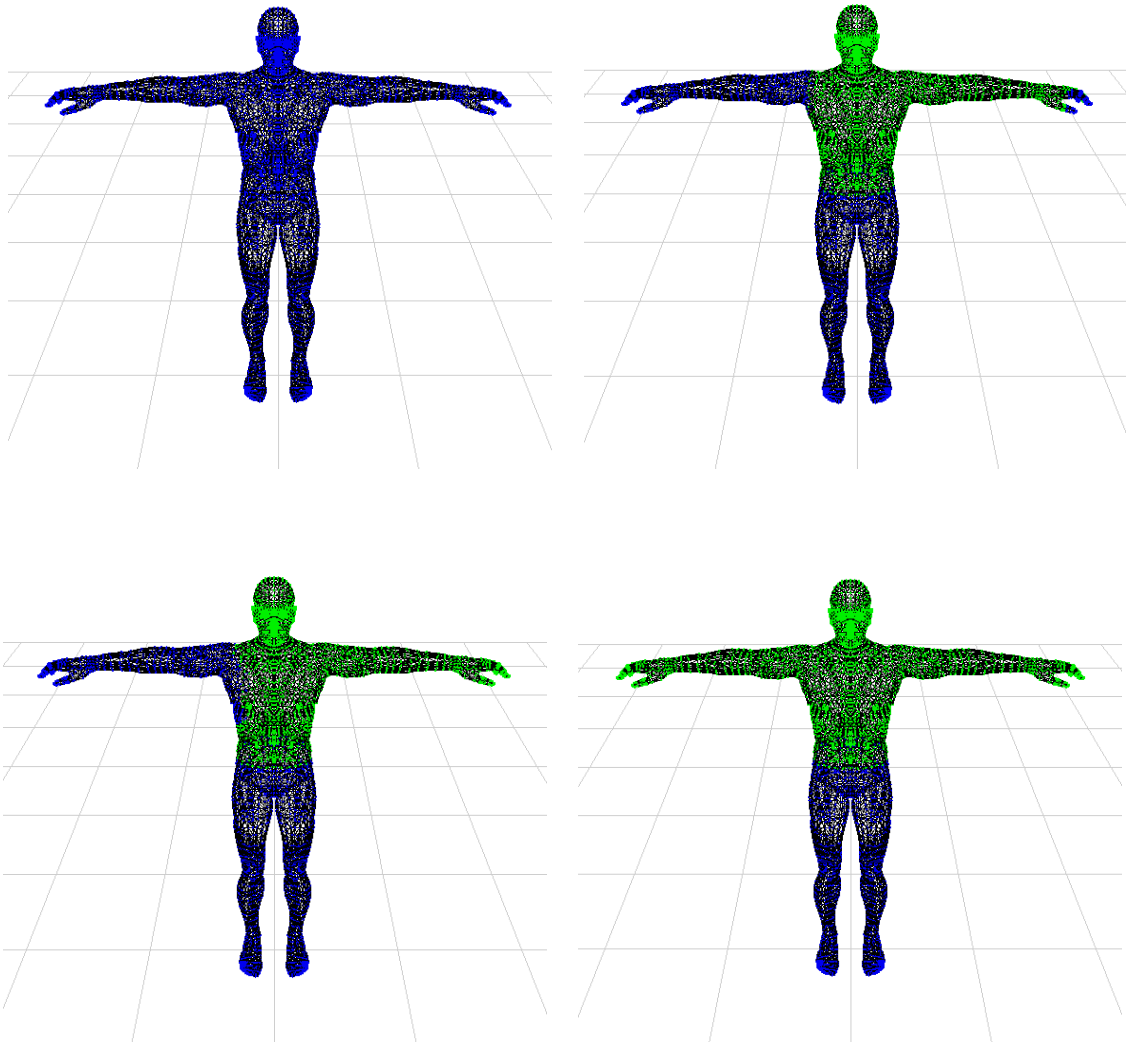
[8] see figures 5(c), 9(b)

## B.4 Mouse control

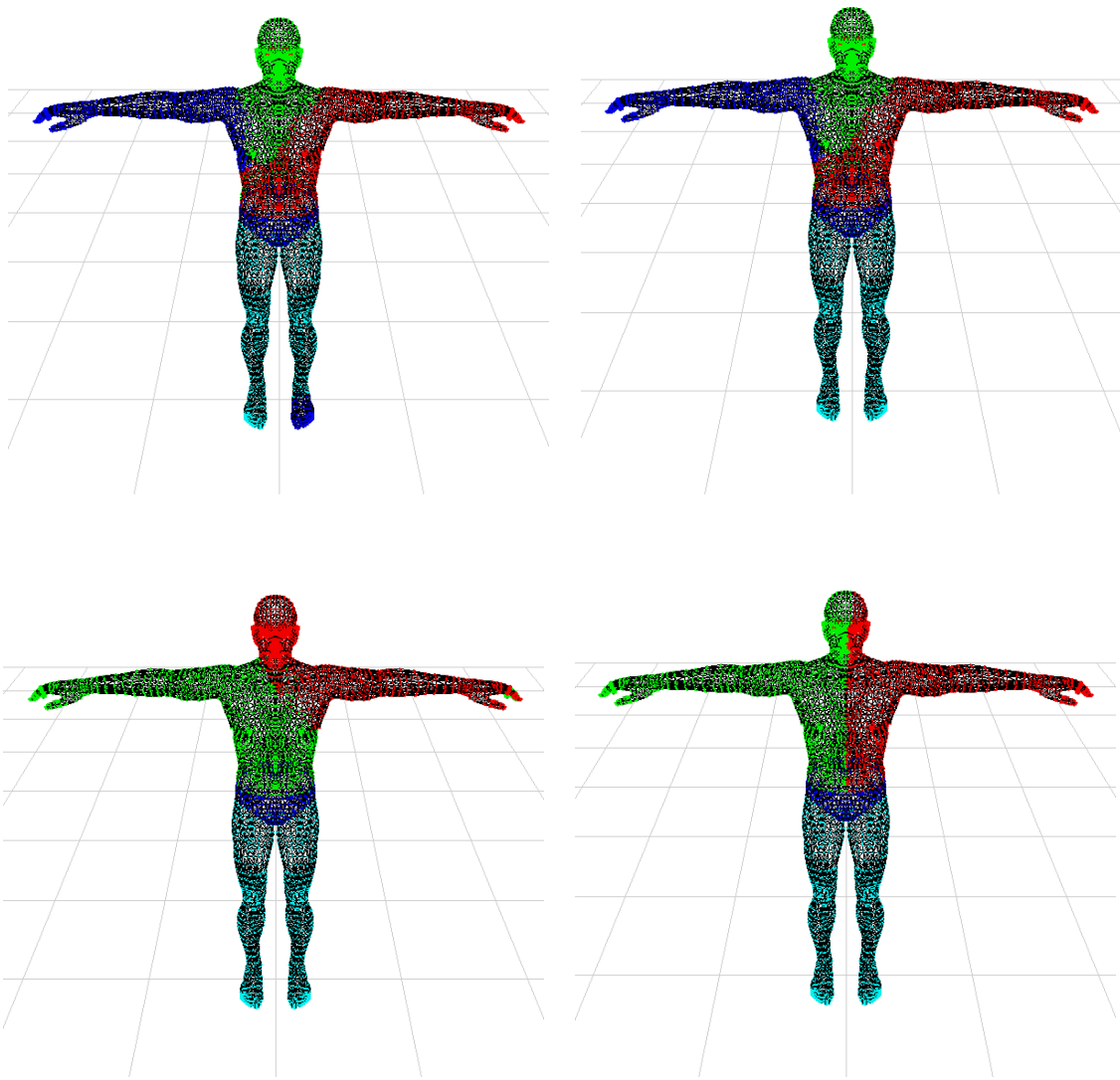
It remains to descibe the possibilities of visualisation. The view is controled by a mouse in graphical window.

- Translation (left button + move): translate the graph with mouse move
- Zoom (wheel button + move): zoom in or out in dependence on mouse move
- Rotation (right button + move): rotate the graph around axes x, y, z.

# C Figures



The detail of the figure 13



The detail of the figure 14