The vertex separation problem in bipartite graphs: A cycle-based algorithm

Bachelorthesis

Author:	Ina Hoffmann
Field of study:	Mathematics
Matriculation number:	302431
Professor:	Prof. Dr. Marco Lübbecke

Contents

1 The problem

1.1 Motivation

Nowadays parallelization is very important in many applications of mathematics and information technology. Especially in the field of operations research it is often not preventable to run complex algorithms with huge matrices as input. If the complexity of an algorithm is to high to run it at a machine, parallelization of the algorithm can make it possible to run it anyway.
Many problems in operations research can be modeled as a mixed integer program.

Definition 1.1 (Mixed integer program)
A mixed integer program be expressed as :
maximize $c^T x$
subject to $Ax \leq b$
and $x \geq 0$
and some entries of x *are known to be integers.*
A *is a matrix,* c *and* b *are vertices and* x *is the vector of variables to be determined.*

In this paper, when speaking about a matrix representing a problem, the matrix A is meant.
The more variables the problem has, the more entries the matrix has. Computing an algorithm that works on a very huge matrix can be inherently problematic, because matrix operations run mostly not in linear time. A straightforward implementation of the matrix multiplication for example has a run time of $\mathcal{O}(n^3)$.
But the main problem is that for a lot of problems not even a polynomial time algorithm is found. To run an algorithm with exponential complexity with a huge matrix as input is almost impossible.
There are two common ways to approach this problem. The first is finding better algorithms to solve the problems. For example the Coppersmith–Winograd algorithm performs a matrix multiplication in $\mathcal{O}(n^{2{,}375})$ instead of $\mathcal{O}(n^3)$. [3] Unfortunately a better algorithm cannot be found for every problem. For many problems a lower boundary for the effectiveness of any algorithm solving the problem is found and proved. Some problems are even proven to be NP-hard which means there is no polynomial time algorithm solving the problem unless P=NP. For example the 0-1 integer linear programming problem is known to be NP-complete, but is very important in many applications. This problem can be described as follows.

Definition 1.2 0-1 integer linear programming problems (0-1ILP)

$$\begin{array}{ll} \textit{maximize} & c^T x \\ \textit{subject to} & Ax = b \\ & x \geq 0 \\ \textit{and} & x \in \{0,1\} \end{array}$$

$A \in \mathbb{N}^{n \times m}$ is a matrix, $b \in \mathbb{N}^n$ is a vector, $c \in \mathbb{N}^m$ is a vector and $x \in \{0,1\}^m$ is the vector we need to find.

This problems are NP-complete, they belong to "Karp's 21 NP-complete problems" [5]. There are many famous problems that can be described as 0-1ILP. One of this is the traveling salesman problem. This is formulated in many papers as follows. "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

Assuming that $P \neq NP$, algorithms finding good solutions have a long run time. Because of this, finding better algorithms alone does not solve the problem of a high complexity. As mentioned above there is a second way to approach this: parallelization.
Especially sparse matrices can be split into sub matrices with just a few rows and/or columns that are omitted. These omitted rows and columns describe the relation between the sub matrices. When a matrix is split into several parts, the problem can be solved for each part individually. Such a partition should not alter the underlying problem that is described by the matrix. Otherwise the solution found to the problem-parts could not be used to find a solution for the complete problem.
Partitioning the matrix has the additional advantage of finding out more about the structure of the underlying problem. It is very useful for solving the problem to know that it naturally consists of a number of separated sub-problems. One way to obtain a split matrix is converting the matrix into a form that can be easily partitioned. One matrix structure which allows easy partitioning is the arrowhead form.

Definition 1.3 (Matrices in arrowhead form)
Let A be a $n \times m$-matrix. A is in arrowhead form if and only if it has the following structure:

$$A = \begin{pmatrix} A_1 & & & & C_1 \\ & A_2 & & & C_2 \\ & & \ddots & & \vdots \\ & & & A_k & C_k \\ R_1 & R_2 & \cdots & R_k & L \end{pmatrix}$$

Thereby each A_j is a matrix and they can differ in size. For each $j \in \{1, \ldots, k\}$

C_j is a matrix with c columns and R_j is a matrix with r rows. L is a $r \times c$-matrix. All other entries of A are zero.

In this paper we call A_j the jth block of A. c is called the column-number and r is called the row-number.

There are several algorithms that use the arrowhead form of a matrix to solve a linear programming problem. For this first each block is considered separately. Then a solution is found by means of the part solutions found before. An example for an algorithm that works this way is the Dantzig–Wolfe decomposition [4]. This paper does not deal with algorithms using the arrowhead form. Its subject matter is how to bring a matrix in arrowhead-form. This will be done by comparing existing ways of doing so, improving them and developing new ways, which hopefully work better.
In this paper additional requirements to the arrowhead form are considered. Possible useful requirements are:

1. The different blocks are as equally sized as possible
2. The column number and the row number are as small as possible
3. The column number and the row number are nearly equally sized

1.2 The graph separator problem

Every matrix can be interpreted as a graph in several ways. One way of representing matrices with graphs will be described in the following. It will be shown that the problem of converting a matrix into arrowhead-form is similar to the problem of finding vertex separators in the representing graph.
In their paper "Partitioning mathematical programs for parallel solution" Michael C. Ferris and Jeffrey D. Horn [6] describe the following way of interpreting a matrix as a bipartite graph:

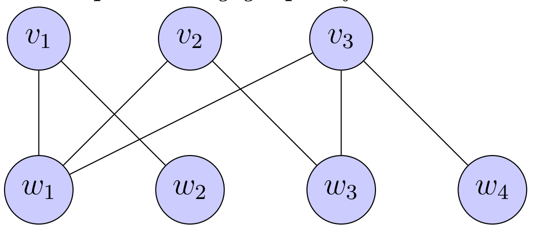
Definition 1.4 (The bipartite graph representing a matrix)
*Let be A a $n \times m$-matrix. The bipartite graph $G = (V_1, V_2, E)$ representing A, contains $|V_1 + V_2| = n + m$ vertices. The n vertices $V_1 = \{v_1, \ldots, v_n\}$ represent the rows of A and the m vertices $V_2 = \{w_1, \ldots, w_m\}$ represent the columns of A.
There is an edge $(v_i, w_j) \in E$ with $v_i \in V_1$ and $w_j \in V_2$ if and only if the entry of A in row i and in column j is not 0.
There is no edge between two vertices that are both in V_1 or both in V_2.*

The graph contains all the relevant information. Each matrix-entry $a_{ij} \neq 0$ is represented by an edge in the graph. However the graph does not represent is the order of the rows and columns. Because of this several possible matrices exist for each graph. However all these possible matrices differ only by permutation. So they all describe the same problem. The so described graph is bipartite because there are per definition no edges within V_1 or within V_2. The definition also describes a unique graph for every matrix-structure except for permutation. This will be illustrated with an example.

Example 1.1 (A matrix and its representing graph)

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

The representing graph of A is:



Each row is represented by a w_i in the graph and each column is represented by a v_i in the graph. For example there are edges from node v_1 to w_1 and w_2. This represents that the entries $(1, 1)$ and $(2, 1)$ are different from 0.

Graph separators

In this section the graph separation problem will be explained first and then the connection between this problem and the arrowhead-form of a matrix will be revealed.
The graph separator problem is - as the name indicates - the problem of finding good separators in a graph. There are two different types of separators. Sets of edges that separate the graph into k subsets are called edge separators. Sets of vertices that separates the graph into k subsets are called vertex separators. In this paper all separators are vertex separators unless they are explicitly said to be edge separators.

Definition 1.5 (k-Separators)
Let $G = (V, E)$ be a graph. Let $S \subset V$ be a set of vertices. Let B be the graph that is generated by removing all vertices in S from G. S is a k-separator of G if and only if B contains at least k different connected components.

Which separators are "good" separators depends on the context. Mostly a separator is defined "good" if its size is small enough.

Definition 1.6 (Minimal Separators) *A minimal separator is one smallest subset of vertices that is a separator.*

In this paper there will be more requirements to the separators than only being small.
Now the connection between vertex separators and the arrowhead-form will be explained. First the structure of the representing graph of a diagonal-block-matrix is illustrated.

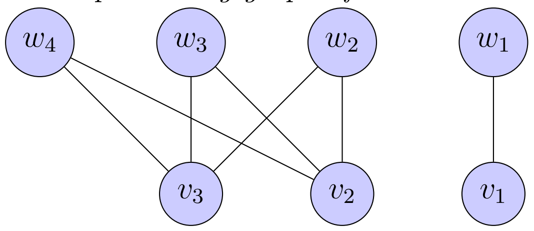
Observation 1.1 (Representing graph of diagonal-block-matrices)
Let A be a matrix with diagonal-block-structure. When A contains at least k blocks, then the representing bipartite graph consists of at least k components.

Proof: Let A be a matrix with diagonal-block-structure. Let $G = (V, W, E)$ be the bipartite representing graph of A. Let A_t be any block of A and $V_t = \{v_1, ..., v_i\} \subseteq V$ the column nodes and $W_t = \{w_1, ..., w_j\} \subseteq W$ the row nodes in the representing graph for the columns and rows of A_t.
Assuming that a node $w_l \in W_t$ is adjacent to a node $v_k \notin V_t$, then $a_{lk} \neq 0$. That means A_t is not a block in A.
Assuming that a node $v_l \in V$ is adjacent to a node $w_k \notin W_t$, then $a_{kl} \neq 0$. That also means that A_t is not a block in A.
Because the graph is bipartite there are no adjacent nodes within V or within W. So there is no adjacent node in $V_t \cup W_t$ to any node in $(V \setminus V_t) \cup (W \setminus W_t)$. That means that $V_t \cup W_t$ is a component. ■

Example 1.2 (A matrix with block structure and its representing graph)

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The representing graph of A is:



A contains two blocks and the graph consists of two connected components.

One strategy for converting a matrix in arrowhead-form is to find columns and rows so that the matrix without these columns and rows would be in diagonal-block-structure. The removed columns and rows are added as the last columns and rows. The so formed matrix is in arrowhead-form. This strategy means several columns and nodes of the matrix are permuted to be the last columns and rows.

Observation 1.2
Let A be a matrix that represents a problem. By permuting rows or columns the matrix still represents exactly the same problem.

Proof: Each row in the matrix stands for a variable. To permute the rows is like renaming them and this does not change the problem. Each column-entry stands for an element in a sum. Addition is commutative so this will not change the problem either. ■

Removing a column is the same as removing a node of the column-set in the representing bipartite graph. Removing a row is the same as removing a node of the row-set of the representing bipartite graph.

In conclusion the problem of transforming a matrix into arrowhead-form can be solved in the following way:

Let Abe a $n \times m$-matrix.

1. Find the representing bipartite graph G of A
2. Find a k-separator in G that
   a) matches all defined requirements
   b) is as small as possible
3. Permute rows and columns until the matrix is in arrowhead-form. The rows and columns represented by the separator will be the last columns and rows in the matrix.

Some possible requirements for the separator are:

- Each subset should contain at least l column-nodes
- Each subset should contain at least l row-nodes
- Each subset should contain similar numbers of column-nodes and row-nodes
- The separator should contain at most l column-nodes
- The separator should contain at most l row-nodes
- The separator should contain similar numbers of column-nodes and row-nodes

1.3 Related problems

Finding k-separators is a known and highly researched problem. The vertex separator number is known to be related to some other quantities.

Edge Separators

The edge separation problem is highly related to the vertex separation problem. As the name indicates an edge separator contains edges instead of vertices.

Definition 1.7 *Let $G = (V, E)$ be a graph. A set of edges S_E is a k-edge-separator if and only if G without the edges in S_E contains at least k disjoint connected components.*

An edge separator S_E can be used to find a vertex separator S_V. For each edge in the edge separator one incident node is put in the node separator.
However this method can not guarantee to find a k-vertex separator.
It can happen that every node in a subset is incident with an edge from S_E. While constructing S_V as described it can happen that each node of this subset is put into S_V. When the whole subset is a subset of S_V, the separator can only segment the other $k-1$ subsets. Then S_V would not be a k-separator but at most a $k-1$ separator.
Nevertheless it is common to use edge separators to find vertex separators, because there are already many good algorithms for solving the edge separation problem. In most cases it is not that important to find a separator that divides the graph into exactly k subsets as long as the number of subsets remains close to k. The method is qualified for this.

More similar problems

Furthermore the vertex separation problem is non constructively related to a couple of other problems. Although the size of the optimal solutions are related this information cannot be used to find an actual separator. Therefor these problems are only non constructively related. Here only the relation between the vertex separation number and the path width will be represented.

Definition 1.8 (Path width of a graph)
Let $G = (V, E)$ be a graph. A graph-decomposition is a sequence $V_1, V_2, ..., V_t$ with $V_j \subseteq V \ \forall j \in \{1, ..., t\}$ for which applies:

1. *For each edge $e_i \in E$ exists a subset V_i so that the begin node and end node of e_i are both in V_i*
2. *For each i, j, k with $i \leq j \leq k$ applies $V_i \cap V_k \subseteq V_j$*

The path width of G is one less than the size of the largest set in a graph-decomposition.

Lemma 1.3 *For each graph the path width equals the vertex separation number.*

This information could be used to find a lower or upper boundary for the size of a minimal separator. Unfortunately research for this paper did not reveal any way to use this information.
So the proof of this lemma is omitted because it is not used in a any algorithm that will be described. The proof can be found in the paper "The vertex separation number of a graph equals its path-width" by Nancy G. Kinnersley. [10].

1.4 Common results

This section contains some other interesting results about the vertex separation problem.

Theorem 1.4 (NP-hardness)
To find a good approximate 2-vertex separator in a graph is NP-hard.

The proof is outlined in the paper "Finding good approximate vertex and edge partitions is NP-hard" by Jones Bui. [2] However this does not prove that the problem remains NP-hard if only bipartite graphs are considered.

There are some special graphs for which the complexity of the problem is known and has been proved:

Definition 1.9 (Biconvex bipartite graphs)
A bipartite graph $G = (V_1, V_2, E)$ is said to be convex over V_t with $t \in \{1, 2\}$ if an enumeration of $V_t = v_1, ..., v_{|V_t|}$ exists so that for each $w_j \in V_2$ applies: For any two adjacent nodes v_i and v_j of w_j there does not exist a nonadjacent node v_l, such that $i \leq l \leq j$.
G is biconvex if and only if it is convex both over V_1 and over V_2.

Graph structure	**Complexity**
Generic graphs	NP [2]
Induced subgraphs of grid graphs with no holes	P [11]
Bipartite graphs	open
Biconvex bipartite graphs	P [13]
Trees	linear [15]

Accordingly if the bipartite graph is a tree or biconvex there exists an algorithm that solves the problem in polynomial time. Unfortunately most bipartite graphs are neither trees nor biconvex.

2 Related work

There are nearly no papers that deal with vertex k-separators. The algorithms that are described in these papers use an algorithm to find 2-separators repeatedly. In this chapter will be described in detail how this can be done.

2.1 Use 2-partitioning algorithms to solve the k-separation problem

In their paper "Partitioning mathematical programs for parallel solution" Michael C. Ferris and Jeffrey D. Horn present an algorithm that use the Kerningham Lin algorithm to find 2-edge separators for finding k-vertex separators [6]. The algorithm consists of the below-mentioned steps.

1. **Construction of a k-edge separator:** Find k subsets of vertices so that the number of edges between these vertices is minimal. To do this they first suggest to find an initial k-partitioning of the graph either randomly or by the use of more elaborate methods. Then pairwise for each two subsets a 2-edge separator is found and that separator defines a new 2-partitioning for the nodes of the two subsets. These two new subsets replace the former ones. There are $\binom{k}{2}$ subsets to be considered this way. This is done several times to get a partitioning with a edge separator that is as small as possible.

2. **Construction of a k-vertex separator:** An algorithm is used to find a set of vertices S_V, so that for each edge in the edge separator at least one node is in S_V, but $|S_V|$ is as small as possible.
   First for each vertex the number of incident edges that are in the edge separator is counted. Then the vertex with the highest number of incident edges is removed. Then the algorithm is run again on the remaining graph. This will be done until the graph is separated in ideally k subsets. The removed nodes form the separator S_V.

However in this way only balanced partitions can be found. Ferris and Horn solved this problem by adding a high number of dummy nodes that have no incident edges before running the algorithm. This can not influence the quality of the solution since the dummy nodes do not have incident edges. Yet in this way unbalanced partitioning is possible.
Naturally this algorithm can not guarantee to find at least k subsets. By removing the dummy nodes at the end it can happen that a subset only consists of dummy nodes and so is removed altogether. Ferris and Horn described one way to avoid this as to not add too many dummy nodes. If a partitioning with between j and k subsets is desired, the total number of nodes has to be $k\lceil n/j \rceil$.

2.2 Example 2-partitioning algorithms

2.2.1 Spectral method using Laplacian matrices

The problem to find a separator can be reinterpreted as an eigenvalue problem. In this section this will be done with a Laplacian matrix like it is described in the paper "Partitioning mathematical programs for parallel solution" of Ferris and Horn [6]. In section 2.2.3 a second method that uses the pagerank is presented.
Let $G = (V, E)$ be a graph and V_1, V_2 a partition of V. We define x_i for each $i \in \{1, ..., |V_1 \cup V_2|\}$ as follows.

$$x_i = \begin{cases} 1 & \text{if } v_i \in V_1 \\ -1 & \text{if } v_i \in V_2 \end{cases}$$

With this definition the number of edges between V_1 and V_2 is described by

$$|E_{V_1V_2}| = \frac{1}{4} \sum_{(v_i,v_j)\in E} (x_i - x_j)^2.$$

If there is no edge between v_i and v_j then $(x_i - x_j)^2 = 0^2 = 0$ and if there is an edge $(x_i - x_j)$ is 2 or -2 which results in $(x_i - x_j)^2 = 4$.
Now this term is rearranged in the following way:

$$\begin{aligned}\sum_{(v_i,v_j)\in E} (x_i - x_j)^2 &= \sum_{(v_i,v_j)\in E} (x_i^2 + x_j^2) - \sum_{(v_i,v_j)\in E} 2x_ix_j \\ &= \sum_{(v_i,v_j)\in E} 2 - \sum_{v_i\in V} \sum_{v_j\in V} x_iA_{ij}x_j \\ &= 2|E| - \sum_{v_i\in V} x_i \sum_{v_j\in V} A_{ij}x_j \\ &= \sum_{v_i\in V} x_i^2 d_i - x^TAx \\ &= x^TDx - x^TAx \\ &= x^T(D-A)x\end{aligned}$$

Here A is the adjacency matrix of G and D is the diagonal matrix with the number of edges incident to v_i as entry d_i for each $i \in \{1, ..., |V_1 \cup V_2|\}$.
$L = D - A$ is the Laplacian matrix of G. In conclusion $|E_{V_1V_2}| = \frac{1}{4}x^TLx$. To find a minimal edge separator is the same as minimizing $|E_{V_1V_2}|$. When a balanced partition is desired there is the requirement $\sum_{i=1}^{n} x_i = 0$. This is a discrete optimization problem because $x_i = \pm 1$. Discrete optimization problems are very

hard to solve, so the problem is changed to be continuous. Instead of $x_i = \pm 1$ the expression $x^T x = n$ is used. Of course it cannot be expected to obtain an exact solution if the problem is modified. But the basic concept of both problems is the same. The difference is that an entry of the solution vector now is not clearly in V_1 or V_2 but the smaller it is the more likely it is to be in V_1 and the bigger it is the more likely it is to be in V_2.
So there is the following continuous optimization problem to be solved:

$$\begin{aligned} \min \ & \frac{1}{4} x^T L x \\ \text{subject to } & \sum_{i=1}^{n} x_i = 0, \text{ and } x^T x = n \end{aligned}$$

A problem of this form can be solved by using eigenvalues. Let $\omega_1, ..., \omega_n$ be an orthonormal basis of the eigenvectors of L with the eigenvalues $\lambda_1, ..., \lambda_n$ and sorted as $|\lambda_1| \leq |\lambda_2| \leq ... \leq |\lambda_n|$. Because $\omega_1, ...\omega_n$ is an orthonormal basis each x can be formulated as $x = \sum_{i=1}^{n} a_i \omega_i$ for some $a_i \in \mathbb{Q}$ with $\sum_i^n a_i^2 = n$. It is a common result that for the Laplacian matrix of each graph $\lambda_1 = 0$ and the associated eigenvector is $(1/\sqrt{n})e$.
With $\lambda_1 = 0$ and substitution of x for $|E_{V_1 V_2}|$ applies:

$$\begin{aligned} |E_{V_1 V_2}| &= \frac{1}{4} x^T L x \\ &= \frac{1}{4} (\sum_{i=1}^{n} a_i \omega_i)^T L (\sum_{i=1}^{n} a_i \omega_i) \\ &= \frac{1}{4} (\sum_{i=2}^{n} a_i^2 \lambda_i) \end{aligned}$$

Furthermore because $|\lambda_2|$ is greater than or is equal to $|\lambda_i|$ with $i \in \{2, ..., n\}$ it for $|E_{V_1 V_2}|$ applies:

$$|E_{V_1 V_2}| \geq \frac{1}{4} (a_2^2 + a_3^2 + ... + a_n^2) \lambda_2 \geq \frac{n \lambda_2}{4}$$

With this formulation it can be seen that $x^* = \sqrt{n} \omega_2$ is a lower bound for $|E_{V_1 V_2}|$. This solution also meets the requirement $\sum_{i=1}^{n} x_i = 0$, since

$$\sum_{i=1}^{n} x_i^* = e^T x^* = (\sqrt{n} \omega_1)^T (\sqrt{n} \omega_2) = \omega_1^T \omega_2 = 0$$

In conclusion x^* is a solution to the continuous minimization problem. If the entries of x^* would be ± 1 we would have found an optimal partition. Because this is not the case one additional step has to be made.
Like described above the higher a entry of the solution is the more likely it is in subset V_1 to get a good partition.
To get a at least good partition we simply allocate the $n/2$ vetrices with the greatest entry in x^* to V_1 and the rest to V_2.

Summarized a 2-edge separator can be found the following way:

1. Construct the Laplacian matrix $L = D - A$ of the graph.
2. Determine the second biggest eigenvalue λ_2 and its eigenvector ω_2.
3. Set $x^* = \sqrt{n}\omega_2$.
4. Allocate the $n/2$ vertices with the highest entry in x^* to one subset and the rest to the other subset.

The edges between this subsets are the edge separator.

2.2.2 Kerningham Lin

As early as 1969 B. W. Kerningham and S. Lin developed a very effective algorithm to find a balanced partition minimizing the edges between these subsets [7]. In fact this algorithm is very popular to this date because it is not only effective but also very simple.
The algorithm is based on a partition with subsets V_1 and V_2 and improves this partition gradually. The main idea of the algorithm is to find a value for each pair (v_i, v_j) with $v_i \in V_1$ and $v_j \in V_2$ that describes the gain of exchanging v_1 and v_2. For this purpose for each node v_i an external cost E_{v_i} and an internal cost I_{v_i} is defined.

Definition 2.1 (External cost)
For each node $v_i \in V_1$ the external cost E_{v_i} is defined as the number of nodes $v_j \in V_2$ adjacent to v_i. For each node $v_j \in V_2$ the external cost E_{v_j} is defined as the number of nodes $v_i \in V_1$ adjacent to v_j.

Definition 2.2 (Internal cost)
For each node $v_i \in V_1$ the internal cost I_{v_i} is defined as the number of nodes $v_j \in V_1$ adjacent to v_i. For each node $v_j \in V_2$ the internal cost I_{v_j} is defined as the number of nodes $v_i \in V_2$ adjacent to v_j.

Definition 2.3 (Difference between external and internal cost)
For each node v_i we define $D_{v_i} = E_{v_i} - I_{v_i}$

So the external cost is the number of adjacent nodes in the other subset and the internal cost is the number of adjacent nodes in the same subset. With this definition the gain of a swap of two nodes can be described. First two helping constants are defined then the gain of a swap is stated and proven.

Definition 2.4 (Number of Edges E)
E_{AB} is defined as the edge-set containing all edges between the vertex sets A and B. So $|E_{AB}|$ is the number of edges between A and B. This is also named the cost of the partition.

Definition 2.5 (Edge indicator function)
For each two vertices the edge indicator function is defined as
$$e_{vw} = \begin{cases} 1, & (v,w) \in E \\ 0, & (v,w) \notin E \end{cases}$$

Definition 2.6 (Gain of swapping)
The gain of swapping the nodes $v_i \in V_1$ and $v_j \in V_2$ is defined as the old cost minus the new cost.
Let $V_1' = V_1 \setminus \{v_i\} \cup \{v_j\}$ and $V_2' = V_2 \setminus \{v_j\} \cup \{v_i\}$ be the subsets after the swap.
The gain is defined by $g(v_i, v_j) = |E_{V_1V_2}| - |E_{V_1'V_2'}|$

Lemma 2.1 *Let $G = (V, E)$ be a graph. Let $V_1 \subset V$ and $V_2 \subset V$ be a partition of G. Let $v_i \in V_1$ and $v_j \in V_2$. The gain swapping v_i and v_j is $g(v_i, v_j) = d_{v_i} + d_{v_j} - 2e_{v_iv_j}$.*

Proof: First it is argued how the number of edges can be described in relation to the internal and external cost. The edges that are neither incident to v_i nor to v_j can not be changed with a swap, so they can be safely ignored. Let z be the number of edges between V_1 and V_2 that are neither incident to v_i nor to v_j. The number of edges between V_1 and V_2 that are incident to v_i is defined as E_{v_i}. The same applies for v_j. If v_i and v_j are adjacent to each other then $E_{v_i} + E_{v_j}$ would count their mutual edge twice. So the number of edges in the old partition can be described as $|E_{V_1V_2}| = z + E_{v_i} + E_{v_j} - e_{v_iv_j}$.
Now is the question what changes in this formula if v_i and v_j are exchanged. Of course instead of the external costs, the internal costs have to be used. In $I_{v_i} + I_{v_j}$ the edge (v_i, v_j) is not counted even if it exists. So the complete new number of edges between the changed sets is $|E_{V_1'V_2'}| = z + I_{v_i} + I_{v_j} + e_{v_iv_j}$.

The gain of the exchange is the old cost minus the new cost.

$$\begin{aligned}
g(v_i, v_j) &= |E_{v_i v_j}| - |E_{V_1' V_2'}| \\
&= z + E_{v_i} + E_{v_j} - e_{v_i v_j} - (z + I_{v_i} + I_{v_j} + e_{v_i v_j}) \\
&= E_{v_i} - I_{v_i} + E_{v_j} - I_{v_j} - e_{v_i v_j} - e_{v_i v_j} \\
&= D_{v_i} + D_{v_j} - 2e_{v_i v_j}
\end{aligned}$$

■

Each step of the algorithm consists of the following steps: Let C be a set of vertices. This set contains the nodes that already have been considered by the algorithm.

1. Determine $g(v_i, v_j)$ for each pair of nodes with $v_i \in (V_1 \backslash C)$ and $v_j \in (V_2 \backslash C)$.
2. Store the v_i and v_j for which $g(v_i, v_j)$ is maximal as a_s and b_s. Each cycle the variable s is increased by one. a_s and b_s are added to C.
3. For each $v_i \in V \setminus \{a_s, b_s\}$ change D_{v_i} as if a_s and b_s would have been exchanged.

This steps are repeated as long as there are still nodes that have not been examined. After the completion of this algorithm the accumulated gain of exchanging $a_1, a_2, .., a_k$ with $b_1, b_2, ..., b_k$ is $g(a_1, b_1) + g(a_2, b_2) + ... + g(a_k, b_k)$. To determine how many nodes should be exchanged to maximize the gain the k which maximizes $\sum_i^k g(a_i, b_i)$ is determined. Obviously the sum is 0 if k is the number of nodes in each subset. Since this would mean to exchange the nodes of V_1 with the nodes of V_2 which would essentially be the same as renaming them.
If this maximized gain is bigger than 0 $a_1, ..., a_k$ are exchanged with $b_1, ..., b_k$.
Then the algorithm starts again until the gain is 0 or lower.

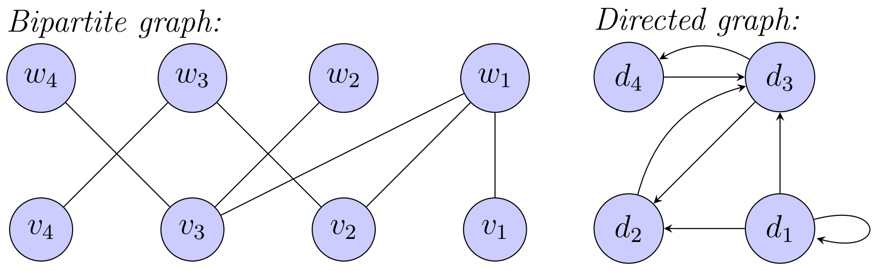
2.2.3 Separating using the pagerank

The pagerank is a measurement for the quality of nodes in a network. It was developed for the google search to rate search results. In their paper "Local Graph Partitioning using pagerank Vectors" Reid Andersen, Fan Chung and Kevin Lang [1] developed an algorithm that uses the pagerank for partitioning a graph. The way of using the pagerank to find partitions they developed is described in this section. Though the method to find the pagerank presented here is not the one Andersen and company developed, but a spectral method. This method is often described and used, for example in "Authoritative Sources in a Hyperlinked Environment" of Jon M. Kleinberg [8].

The pagerank is defined only for directed graphs. Of course every undirected graph can be interpreted as a directed graph with the requirement $\forall(v, w) \in E : (w, v) \in E$. Due to the fact that in this paper the main subject are bipartite graphs, there is a second, better way to interpret the graph as a directed graph.

Observation 2.2 (Interpreting a bipartite graph as a directed graph)
Each bipartite graph $G = (V_1, V_2, E)$ with $|V_1| = |V_2|$ can be interpreted as directed graph in the following manner:
Let $v_1, ..., v_n$ be an enumeration of the nodes in V_1 and $w_1, ..., w_n$ an enumeration of the nodes in V_2. $G_2 = (V, E_d)$ with $|V| = |V_1| = |V_2|$ and for each $a_i, a_j \in V$ applies: $(a_i, a_j) \in E_d$ if and only if $(v_i, w_j) \in E$. Notice that this graph could contain loops if an edge (v_i, w_j) with $j = i$ exists in the bipartite graph.

Example 2.1 (Interpreting a bipartite graph as a directed graph)



Proof: The adjacent matrix will help to show this lemma. The difference between adjacent matrices of directed and undirected graphs is that adjacent matrices of undirected graphs are symmetric. That means in fact not n^2, but only $n^2/2$ entries are needed to characterize an undirected graph. If the graph is even bipartite there are no edges within the subsets, so half of the $n^2/2$ entries are known to be 0. That means that if an undirected graph is bipartite only $n^2/4 = (n/2)^2$ matrix entries are needed to fully characterize the graph.
Now a matrix $A \in \mathbb{Q}^{\frac{n}{2} \times \frac{n}{2}}$ will be created with exactly the needed information.
Let $v_1, ..., v_{n/2}$ be an enumeration of the nodes in V_1 and $w_1, ..., w_{n/2}$ an enumeration of the nodes of V_2.

$$a_{ij} = \begin{cases} 1 & (v_i, w_j) \in E \\ 0 & (v_i, w_j) \notin E \end{cases}$$

In this matrix every edge between a node in V_1 and a node in V_2 is represented by an entry that does not equal zero. So with the matrix and the information that the graph is bipartite the complete graph is characterized. The so defined matrix is the adjacency matrix of the directed graph described above. ■

This definition covers only bipartite graphs with $|V_1| = |V_2|$. However a bipartite graph with $|V_1| \neq |V_2|$ can be easily transformed into a bipartite graph with

$|V_1| = |V_2|$. One simply has to add dummy nodes without any incident edges.

As said before the pagerank of a node is supposed to describe the quality of this node. The quality in this definition is the higher the more high rated pages link to the page.

Definition 2.7 (Link)
In a directed graph an edge between two nodes n_1 and n_2 can be interpreted as link from n_1 to n_2.

A second requirement is, that the pagerank for a node is lower if the node possesses many outgoing links. In conclusion the pagerank of each node can be defined as a fixed point problem as follows:

Definition 2.8 (Pagerank)
Let $G = (V, E)$ be a directed graph. For every $v_i \in V$ let N_{v_i} be the number of outgoing links and I_{v_i} the set that contains the start point of each incoming link. A vector r that contains a pagerank for each node in G is a vector for which applies:

$$r_i = \sum_{v_j \in I_{v_i}} r_j / N_j$$

This fixed point problem can be reinterpreted as an eigenvalue problem.

Definition 2.9 (Matrix Q)
$$q_{ij} = \begin{cases} \frac{1}{N_j} & if\ a\ link\ from\ v_j\ to\ v_i\ exists \\ 0, & else \end{cases}$$

With this definition $\sum_{v_j \in I_{v_i}} r_j/N_j$ is the ith entry of Qr. Hence the problem to find a pagerank vector could be reformulated as follows:
A vector r is to be determined that solves $Qr = r$. This is an eigenvalue problem. Note that r is the eigenvector to the eigenvalue 1.
There are two problems that should be considered before the implementation.
The first problem are 0-rows and 0-columns. If Q has 0-rows or 0-columns it may do not have an eigenvalue 1. This problem can be solved by filling each 0-row-entry and each 0-column-entry with $1/|V|$. This will not change the pagerank vector except for scaling, because these new links are evenly distributed.
The second problem is Q could be reducible. In this case there could be more than one eigenvector to the eigenvalue 1. This fortunately is no problem in this context, because it does not matter if more than one possible partitioning is found.

How can the pagerank be used to partition the graph? Pages with a high pagerank are highly networked within the graph and nodes with a high pagerank are likely to be connected to multiple nodes. In this way a good 2-partitioning can be found by taking the k nodes with the highest pagerank in one subset and the rest in the other subset. k has to be determined as the value for which the subsets are optimal in terms of the requirements. Because each node represents two nodes of the bipartite graph, this way of partitioning can only find partitions with the represented nodes v_i and w_i in the same subset. If other partitions are required, the trivial directed graph instead of the here described can be used to avoid this. This method of partitioning finds one very good subset and simply puts the rest in the other subset. Unfortunately this does not help with the k-partitioning problem. When determining the pagerank of the subsets the result will be very similar to the origin pagerank because the network-structure was not changed and former highly connected nodes remain so. Therefore the continued partitioning of these subsets will generate bad results.
In conclusion this is a very promising way to solve the 2-partitioning problem, but it can not be used for the k-partitioning problem.

2.2.4 A genetic algorithm

Another intriguing idea to solve the problem is to use a genetic algorithm as the one developed by Hasan Prikull and Erik Rolland and published 1994 in paper named "New heuristic solution procedures for the uniform graph partitioning problem: Extensions and evaluation". [12]
A genetic algorithm is an algorithm that uses the concepts of mutation and survival of the fittest to solve a problem. First an initial generation P_0 is defined. This is a set of sub optimal solutions. In the context of partitioning a graph it would contain several partitions. How many individuals the initial generation has is very important for the quality of the solution the algorithm will find. The more individuals the initial population contains the better results will be found, but the more complex the algorithm is as well.
Then this initial population is evaluated. That means it must be defined which individuals have the best chance to survive, or in the context of graph partitioning: Which partitions are better than others. In the following this property is called "fitness" of an individual. The evaluation function describes the number of edges between the subsets of a partitioning. The higher this value, the worse the partitioning. As discussed in the first chapter, we have additional requirements for the subsets. For each partitioning that does not meet the requirements the evaluation function is set very high no matter how many edges are between the subsets.
Now a new generation is created. Therefore individuals with a high fitness are more likely to create offspring. If two individuals are chosen to produce the off-

spring an evaluation operator is used to create two individuals that have some features of both parents. In this context that means that in the new partitionings some nodes are in the same subset as they were in the father-partition and some nodes are in the same subset as they were in the mother-partitioning.
Additional mutation is important for each genetic algorithm. So every new partitioning is slightly and randomly altered. Now the evaluation function of this new population P_1 is determined.
In this way new generations are created as many times as predefined.

Prikull and Rolland advise in their paper to use a so called "queen bee"-evaluation. That means not both parents are chosen more or less random, but there is only one "mother" for each population. Before creating a new population the best partitioning is determined and this will be the new "queen bee". This is the mother of the whole population. Additionally they advise to make a cross over only on one point. Crossing over is the procedure described above for each child to show some qualities of both parents. First one child is a copy of the father and one child is a copy of the mother. Then random features are chosen to resemble this feature of the other parent. In this context that means a single random node is put in the subset in which the other parent had this node.
An interesting point to this algorithm is, that it can be easily transformed into an algorithm to solve the k-partitioning problem. The only difference is there are k states for each node instead of two.

2.3 Using the bipartite structure of the graph

The only algorithm that uses the bipartite structure of the graph is the pagerank algorithm and it provides only the advantage to have a smaller directed graph. Unfortunately there is no known way to benefit from the bipartite structure for partitioning. No researched algorithm could be improved with this fact. Of course a bipartite graph has always a trivial partition. If one subset of the bipartitition is declared as separator, there will be as many subsets as the number of nodes in the other subset. Perhaps an algorithm could be defined which begins with this trivial partition and adds nodes as long as the graph is comprised of more than k nodes. But this algorithm would only find separators that contain only row-nodes or only column-nodes. This solution does not meet the requirement that a minimum of row nodes and a minimum of column nodes should be included in the separator.
Every other algorithm find unavoidable subsets and separators that contain column nodes as well as row nodes. Of course this is exactly what was required, but it also makes it nearly impossible to use the bipartite structure.
The only difference between a bipartite graph and a non-bipartite graph is the

bipartite graph has some nodes that are known to be nonadjacent to certain other nodes. Depending on the used data structure this might result in greater memory efficiency.
Also the graph is known to be not complete. A complete graph cannot be partitioned in any useful manner. Unfortunately a complete bipartite graph is still possible, but cannot be partitioned apart from the trivial partitioning either.
Each above considered algorithm runs faster if less edges are in the graph. But this only means that the algorithms will improve with less non zero entries the matrix has. The matrix being interpreted as a bipartite graph has nothing to do with it.
In conclusion it is unlikely that the problem is any different when the graph is bipartite.

2.4 Ways to improve the algorithm

Each of these algorithms can be improved on general graphs by finding cliques before implementing it.

Definition 2.10 (Clique)
A clique of a graph G is a subset of vertices, where every two different vertices are adjacent to each other.
In this section with clique only cliques with at least 3 nodes are meant.

There is no partition with a vertex separator that has parts of a clique in one subset and parts of the same clique in another subset. Otherwise the subsets would not be separated because nodes from the first subset would be adjacent to nodes from the second s. That means a clique is either completely in one subset, it is completely in the separator or it is in the separator besides for 1 node.
So before implementing the cliques are defined as one supernode instead of several nodes. The supernode is adjacent to all nodes nodes of the clique were adjacent to but that were not in the clique themselves.
Unfortunately a bipartite graph has no cliques with at least 3 nodes. Also an attempt to define a similar concept for bipartite graphs and use it did not bring results. Of course a bipartite clique can be defined as follows.

Definition 2.11 (Bipartite clique)
A bipartite clique of a bipartite graph $G = (V_1, V_2, E)$ is a subset of vertices, where every two different vertices $v_1 \in V_1$ and $v_2 \in V_2$ are adjacent to each other. Let C_1 be the set of all vertices in the clique with $v_1 \in V_1$ and C_2 be the set of all vertices with $v_2 \in V_2$.

With this definition there is no partition where in one subset are nodes of C_1 and in another partition are nodes of C_2. Nevertheless there could be a partitioning

with either C_1 or C_2 completely contained in the separator and the other one split up in many subsets.
So this improvement cannot be used for bipartite graphs.

3 Find separators cycle based

In this section a new algorithm to find vertex separators is developed. The algorithm is mainly based on two thoughts.

1. Trees are easily separated.
2. Cycles are easily separated.

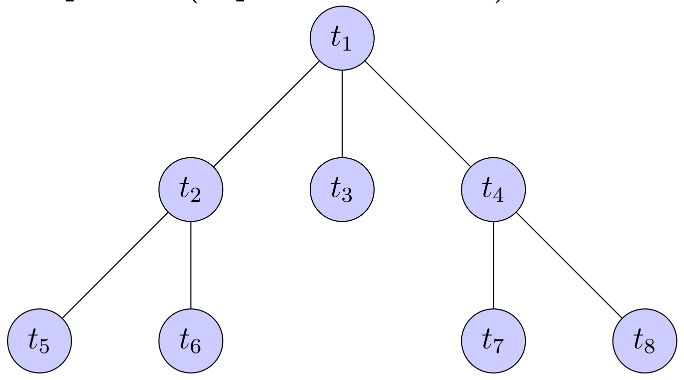
3.1 Separating a tree

In this section a method to separate a tree is developed. With tree in this section always a ordered tree with a fixed root is meant. So every node besides one has a defined father.
In a tree almost every node is a separator. Except the leaves each node separates the tree into d subsets where d is the degree of the node. When a k-separator is searched and there is no node with degree k or higher, the following formula can be used.

Theorem 3.1 *Let T be a tree. Let $S = \{s_0, ..., s_n\}$ be a set of nodes. Let t be the number of nodes in S whose fathers are in S, too. Let $d(v)$ be the degree of the node v. S is a k-separator of T if and only if*

$$\sum_{i=0}^{|S|} d(s_i) - |S| + 1 - t = k$$

Example 3.1 (Separators of trees)



t_1 separates the tree in $d(t_1) = 3$ subsets.
$\{t_2, t_4\}$ separates the tree into $d(t_2) + d(t_4) - 2 + 1 = 3 + 3 - 2 + 1 = 5$ subsets.
$\{t_2, t_1\}$ separates the tree into $d(t_2) + d(t_1) - 2 + 1 - 1 = 3 + 3 - 2 + 1 - 1 = 4$ subsets.

Proof: A single node n_0 in a tree separates exactly $d(n_0)$ subsets. When a node n_i is added to a non empty separator there are two possibilities:

1. The node is not adjacent to a node in the separator. In this case the separator partitions the tree into $d(n_i) - 1$ more subsets.
2. The father of the node is in the separator. In this case one already counted subset is removed, because when the father was added to the separator the subset of each child was counted. So the new node brings one new subset less than if its father had not been in the partition and the separator partitions the tree into $d(n_i) - 2$ more partitions than before.
3. The node is a father of a node in the separator. The same as number 2 but the other way around.

In conclusion this gives exactly the above formula. ■

With this formula it can be determined if a subset is a separator. However it does not help to find a partition that meets the requirements that are defined in chapter 1. While the separator is build node by node it can be ensured that the separator contains a predefined number of row- or columnodes. But there is no way to affect the size of the subsets. This problem can be solved in the following way.
Before searching for a separator at least k subsets that meet the requirements are found the following way:

1. Each leaf is declared as a subset.

2. As long as the subsets do not meet the requirements the father of the last added node is added to the subset. In this way in each step subsets are merged together.

Now a k-separator is searched for in the rest of the graph. But a requirement to this separator is that it segments at least k of the found subsets.

3.2 Separating a cycle

A k-separator of a cycle is each set of nodes that contains $k - 1$ pairwise not adjacent nodes.

3.3 Cycle based separating of a graph

When in a graph each cycle is eliminated, it is a tree. In this algorithm the graph is gradually converted to a tree. One node is defined as father. This can be done randomly but the result can be improved if a highly connected node is chosen. Then each adjacent node is defined as child of this father. For each adjacent node of this children do one of the following:

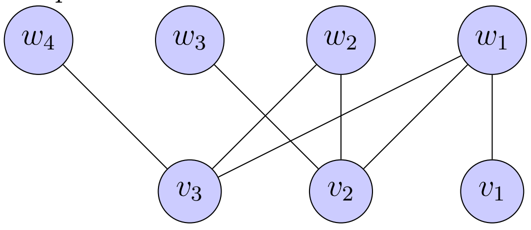
1. If the adjacent node is the father do nothing.

2. If the adjacent node is one node that is not already in the tree define this node as a child

3. If the adjacent node is one node that is already in the tree but is not the father: create a supernode with the cycle that contains the node and the adjacent node. This cycle can be found if the shared forefather is found by going up the tree one by one after going up with the lower node until it has the same deep. When the supernode is created each child of a node contained in the supernode that is not contained in the supernode itself is a child of the supernode. The father of the found forefather is the father of the supernode.

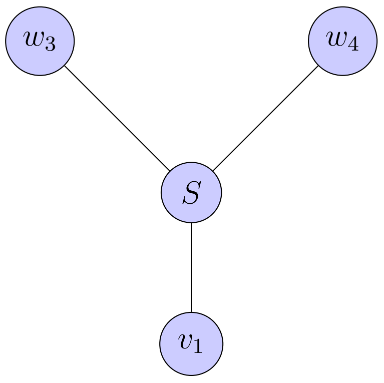
This will be done until there are no nodes left that are not in the tree.

Example 3.2 Cycle stacking
Graph G:

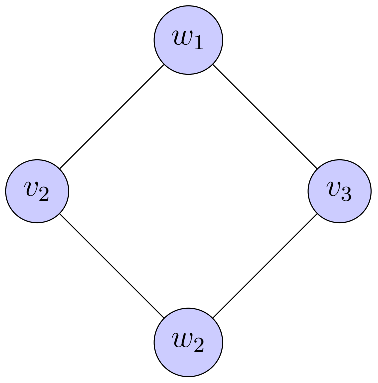


1. *step: v_1 is declared as father.*
2. *step: The only adjacent node of v_1 is w_1. w_1 is declared as a child of v_1.*
3. *step: The adjacent nodes of w_1 are v_1, v_2 and v_3. v_1 is the father of w_1, the other two are declared as children of w_1.*
4. *step:*
   a) *The adjacent nodes of v_2 besides its father are w_2 and w_3. They are declared as children of v_2.*
   b) *The adjacent nodes of v_3 besides its father are w_4 and w_2. w_4 is declared as child of v_3. w_2 is already in the tree, so a supernode will be created. The shared forefather of w_2 and v_3 is w_1. Therefore the new supernode consists of $w_2, w_1, v_3, andv_2$.*

The tree is:



The separator S is:



Now this tree must be separated. For each supernode in the generated separator the cycle in the supernode must be separated gradually until we arrive at a separator that contains no supernodes.

3.4 Implementation

The algorithm to stack cycles described above is implemented in C++. Therefore the following data structure is developed.

Treenode
+ node: Node* + father: Treenode* + children: list <Treenode*> + deep: Integer

+ Treenode(Node* node)
+ Treenode(Node* node, Treenode* father)
+ createSupernode(Treenode* newTreenode)
+ removeChild(Treenode* t)
+ newChild(Treenode* t)
+ hasChild(Treenode* t): boolean
+ hasGrandchild(Treenode* t): boolean

Node
+ columnNodes: Integer + rowNodes: Integer + index: Integer + containedNodes: list <Node*> + name: String + Node(int c, int r, String n) + Node(int c, int r, int i, String n) + Node(const Node& other) + addNode(Node* n) + contains(Node* n): boolean

Additional there is a map with the nodes as keys and a list with all adjacent nodes as value. With the method createSupernode(Treenode* t) a treenode determines the shared forefather of itself and the commit treenode. Then a supernode is created as described above. Note that in this implementation a supernode is a node that contains at least one node.
A problem of this implemenetation is to contain every adjacency information four times. Both the child has a link to its father and the father has a link to its children. Plus the adjacency lists of two adjacent nodes both contain this information. This structure allows fast running through the tree in both directions. Though in the algorithm every time an adjacency relationship is changed it has to be changed in all four places.

The algorithm to stack cycles was implemented exactly as described above. Unfortunately this implementation revealed the that every researched matrix contains too many very small cycles that are connected with each other, to be partitioned in this way. Because of this the resulting structure is a tree that contains a very big supernode and very few other nodes. Thereby the supernodes are interleaved so that nearly every supernode contains just two or three nodes plus another supernode.

If this structure would be separated as described above the whole supernode would be declared as separator because in no level there are enough nodes to declare a k-separator that does not contain a supernode. Because the supernode contains more nodes than the rest of the graph together, this were a much to big separator to be useful. There is no possible way to separate this structure properly.

4 Conclusions

The k-vertex separator problem for bipartite graphs is little explored. Ferris and Horn developed an algorithm which solves this problem using 2-edge separators. In this paper another algorithm was developed. The approach of this algorithm was to unit nodes, then separate the graph and finally dissolve the units again. If all cycles of a graph were united to supernodes the remaining graph is a tree and because of this it easily separated. The implementation showed, that most graphs contains to many interleaved cycles to be separated this way. However there could be other methods to unit nodes so that the remaining graph is easier separated. In any case the necessity exists to investigate this topic further.

References

[1] Andersen R., Chung F., Lang K. "Local Graph Partitioning using PageRank Vectors"

[2] Bui, T.N. and C. Jones (1992). Finding good approximate vertex and edge partitions is NP-hard, Information Processing Letters 42 (1992): 153-15

[3] Coppersmith, Don; Winograd, Shmuel (1990). "Matrix multiplication via arithmetic progressions". Journal of Symbolic Computation 9 (3): 251, doi:10.1016/S0747-7171(08)80013-2.

[4] Dantzig G. B., Wolfe P.(1960). "Decomposition Principle for Linear Programs". Operations Research 8: 101–111.

[5] Richard M. Karp (1972). "Reducibility Among Combinatorial Problems". In R. E. Miller and J. W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum. pp. 85–103.

[6] Ferris M. C., Horn J. D. (1994). "Partitioning mathematical programs for parallel solutions". Mathematical Programming 80 (1998): 35-61.

[7] Kernighan B. W., Lin S. (1969). "An Efficient Heuristic Procedure for Partitioning Graphs". The Bell System Technical Journal (February 1970): 291-307.

[8] Kleinberg J. M. (1997). "Authoritative Sources in a Hyperlinked Environment". 1997, Dept. of Computer Science, Cornell University, Ithaca.

[9] T. Kloks, D. Kratsch, H. Mfiller. "Dominoes".

[10] Kinnersley G. Nancy (1992). "The vertex separation number of a graph equals its path-width". Information Processing Letters 42 (1992): 345-350.

[11] Papadimitriou C.H., Sideri M. (1996). "The Bisection Width of Grid Graphs". Math. Systems Theory 29, (1996): 97-110

[12] Pirkul H., Rolland E. (1994), "New heuristic solution procedures for the uniform graph partitioning problem: Extensions and evaluation". Computers Ops RPS. Vol. 21, No. 8: 895-907

[13] Sheng-Lung Peng, Yi-Chuan Yang. "On the Treewidth and Pathwidth of Biconvex Bipartite Graphs".

[14] Rosenberg A., Heath L. S. (2001). "Graph separators, with applications". Kluwer Academic/Plenum Publishers. New York.

[15] Skodinis, K. (2002). "Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time". Journal of Algorithms 47 (2003) 40–59.