University of Warsaw
Faculty of Mathematics,
Informatics and Mechanics

THEORY AND PRACTICE OF COMPUTING MAXIMUM MATCHINGS IN GRAPHS

PhD Thesis

Piotr Stańczyk

Supervisor:
prof. Krzysztof Diks

Warsaw, June 2011

Author's declaration

Aware of legal responsibility I hereby declare that I have written this thesis myself and all its contents have been obtained by legal means.

Date   Author's signature

Supervisor's declaration

This thesis is ready to be reviewed.

Date   Supervisor's signature

Abstract

The maximum matching problem is one of the most extensively studied problems in the entire graph theory. The first results were published already in the 19th century. The maximum matching problem is important not only due to its interesting theoretical nature, but also because of a number of practical applications — ranging from scheduling to advanced image processing. In this work we address the problem from two perspectives — theoretical and practical. A number of novel matching algorithms for different subclasses of cubic graphs are introduced. We present a new deterministic perfect matching algorithm for biconnected cubic graphs running in time $O(n \log^2 n)$ and a randomized version running in expected $O(n \log n \log \log^3 n)$ time. Both algorithms are faster than previously known algorithms. We also present two parallel algorithms for solving perfect matching problem in bipartite cubic graphs and bipartite planar cubic graphs. The time complexities of these algorithms are $O(\log^2 n)$ and $O(\log n \log^* n)$ time respectively, while the requirement on the number of processors are $O(n/\log n)$ and $O(n)$. The first parallel algorithm has the same time complexity as the previously known fastest algorithm, but utilizes less processors. The second algorithm, on the other hand, is the fastest algorithm known to date for solving bipartite planar cubic case.

Practical part of this work introduces implementation of a couple of matching algorithms together with performance analysis for different classes of graphs. Among others we present a parallel implementation of algebraic maximum matching algorithm for general graphs ported to a GPU architecture. This algorithm outperforms efficient state of the art matching algorithms for some classes of graphs.

Keywords: maximum matching, perfect matching, cubic graphs, biconnected graphs, bipartite graphs, planar graphs, sequential and parallel algorithms.

ACM Classification: F.2.2, G.2.2.

Streszczenie

Problem najliczniejszego skojarzenia jest jednym z najczęściej badanych problemów grafowych — nie tylko ze względu na ciekawe podłoże teoretyczne, ale również ze względu na wiele praktycznych zastosowań, takich jak planowanie zadań czy zaawansowane metody obróbki obrazu. Pierwsze wyniki zostały opublikowane już w XIX wieku. W niniejszej pracy problem skojarzeń analizujemy w dwóch aspektach — teoretycznym i praktycznym. W części teoretycznej prezentujemy kilka nowych algorytmów wyznaczających najliczniejsze skojarzenie dla różnych podklas grafów kubicznych. Zaczynamy od deterministycznego algorytmu do wyznaczania doskonałego skojarzenia w dwuspójnych grafach kubicznych, działającego w czasie $O(n \log^2 n)$ oraz wersji randomizowanej tego algorytmu działającej w czasie $O(n \log n \log\log^3 n)$. Oba algorytmy są najszybszymi znanymi do tej pory algorytmami sekwencyjnymi. W dalszej części pracy przedstawiamy równoległy algorytm dla problemu doskonałego skojarzenia w dwudzielnych grafach kubicznych działający w czasie $O(\log^2 n)$ z wykorzystaniem $O(n/\log n)$ procesorów oraz algorytm dla planarnych dwudzielnych grafów kubicznych działający w czasie $O(\log n \log^* n)$ z wykorzystaniem $O(n)$ procesorów. Pierwszy z tych algorytmów ma taką samą złożoność czasową jak najszybszy znany do tej pory algorytm, ale wykorzystuje mniejszą liczbę procesorów. Drugi jest z kolei najszybszym znanym algorytmem dla przypadku planarnych dwudzielnych grafów kubicznych.

W części poświęconej zagadnieniom praktycznym prezentujemy implementację kilku algorytmów rozwiązujących problem najliczniejszego skojarzenia i dokonujemy analizy wydajnościowej tych algorytmów w kontekście różnych klas grafów. Jednym z prezentowanych algorytmów jest równoległa implementacja algebraicznego algorytmu do wyznaczania najliczniejszego skojarzenia w dowolnych grafach zrealizowana na karcie graficznej. Dla pewnych klas grafów algorytm ten okazuje się być szybszy od efektywnych implementacji klasycznych algorytmów sekwencyjnych. Wynik ten daje nadzieję na uzyskiwanie coraz szybszych algorytmów dla problemu skojarzeń w grafach poprzez zastosowanie równoległości.

Słowa kluczowe: najliczniejsze skojarzenia, doskonałe skojarzenie, grafy kubiczne, grafy dwuspójne, grafy dwudzielne, grafy planarne, algorytmy sekwencyjne i równoległe.

Klasyfikacja tematyczna ACM: F.2.2, G.2.2.

Contents

Chapter 1

Introduction

Problem definition. For a given graph $G = (V, E)$ with vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and edge set $E = \{e_1, e_2, \ldots, e_m\}$ by a *matching* $M \subseteq E$ we understand any subset of edges of the graph such that no two edges of M have a common endpoint. *Maximum matching* is a matching of the maximum cardinality. *Perfect matching* is a *maximum matching* of size $|V|/2$. Solving the maximum/perfect matching problem is to find a maximum/perfect matching in a given graph G.

The history of the matching problem. The history of matchings goes back to the end of the 19th century when Petersen published a pioneering paper about the matching theory [48]. In this paper he proved that every cubic graph without bridges has a perfect matching. The proof of the Petersen's Theorem by Frink [23] yields an $O(n^2)$ time maximum matching algorithm. The general matching problem, however, remained unknown to be solvable in polynomial time for decades. With time it turned out that it is much easier to solve the bipartite graph case. Characterization of maximum matchings in bipartite graphs was introduced in 1931 independently by König [36] and Egerváry [18]. Their constructive proofs provided polynomial-time algorithms for maximum matching problem in bipartite graphs — so called Hungarian method. Over thirty years later, in 1965, Edmonds [17] provided the first polynomial time algorithm for general graphs. His $O(n^4)$ algorithm, as it turned out later, was far from optimal. In 1976 Gabow [24] showed an optimization of Edmond's algorithm leading to $O(n^3)$ time complexity. Also Even and Kariv [19] provided an improvement with $O(min\{n^{2.5}, m\sqrt{n}\log n\})$ time complexity. The improvement, however, greatly complicated the algorithm. In the meantime (1973) Hopcroft and Karp [33] presented a bipartite graph matching algorithm running in $O(m\sqrt{n})$ time. Seven years later Micali and Vazirani [41] managed to provide an equivalent for general graphs with the same time complexity. Blum [7] and Gabow and Tarjan [25] introduced an alternative matching algorithms with the same time complexity. A number of other matching algorithms have been proposed since that time:

- 1991, Alt, Blum, Mehlhorn and Paul [28] — bipartite matching algorithm running in time $O(n^{3/2}\sqrt{m/\log n})$,

- 1991, Feder and Motwani [21] — bipartite matching algorithm running in time $O(\frac{\sqrt{n}m\log(n^2/m)}{\log n})$,
- 2003, Fremuth-Paeger and Jungnickel [22] — generalization of Feder and Motwani algorithm to general graphs,
- 2004, M. Mucha and P. Sankowski [43] — randomized matching algorithm for general graphs running in $O(n^\omega)$ [1] time,
- 2006, N. J. A. Harvey [29] — simplification of the algorithm by Mucha and Sankowski.

From the theoretical standpoint the fastest maximum matching algorithm for general, dense graphs is Harvey's [29] simplification of M. Mucha and P. Sankowski [43] algorithm. However, as it utilizes fast matrix multiplication, it does not prove to be practical.

Although research in the area of maximum matchings has been lead for over one hundred years it is still a challenge to implement an efficient algorithm in general case. That is why different subclasses of graphs are of a great interest. The most recognized subclass with vast practical applications are bipartite graphs. We have already mentioned some results for this class of graphs. Other interesting classes are planar and cubic graphs. All planar graphs are sparse (according to the Theorem 2.5, the number of edges in a planar graph is not greater than $3n-3$), hence the complexity of $O(m\sqrt{n})$ algorithm for general graphs reduces to $O(n^{3/2})$ in the planar case. M. Mucha and P. Sankowski [42], by applying nested dissection, managed to accommodate their algebraic matching algorithm to planar graphs obtaining an $O(n^{\omega/2})$ time complexity — it is the fastest theoretical matching algorithm for general planar graphs.

Cubic graphs became of greater researchers' interest just recently. All thanks to T. Biedl [5], who showed a linear time reduction of the maximum matching problem in general graphs to the maximum matching problem in 3-regular (cubic) graphs. Her result implies that any $O(f(m))$ maximum matching algorithm for 3-regular graphs yields an $O(f(m)+m)$ maximum matching algorithm for arbitrary graphs. Cubic graphs are the simplest of all graphs for which the matching problem remains as hard as for general case. R. Greenlaw and R. Petreschi in [27] survey algorithms for different classes of cubic graphs giving motivation for further exploration.

From the presented results it is clear that good understanding of matchings in cubic graphs may lead to faster algorithms in general case. A number of matching algorithms have been proposed so far for different subclasses of cubic graphs. The starting point is already mentioned Petersen's Theorem [48] and Frink's $O(n^2)$ matching algorithm for bridgeless cubic graphs. In 2001 Biedl at al. [6] presented a new algorithm for computing perfect matching in bridgeless cubic graphs which runs in $O(n\log^4 n)$ time. This algorithm is faster than any other algorithm discussed so far. Some more results are known for cubic graphs. Biedl at al. [6] showed that it is possible to find a perfect matching in a planar biconnected cubic graph in linear time,

[1] $O(n^\omega)$ is an optimal matrix multiplication time. It is known that $\omega < 2.376$.

while Schrijver [51] provided a simple linear matching algorithm for bipartite cubic graphs.

Parallel maximum matching algorithms. Research on parallel matching algorithms began around 40 years ago, however the main theorem utilized by parallel algebraic algorithms has been known since 1940. In 1947 W.T. Tutte [61] stated that a graph has a perfect matching if and only if the determinant of a certain skew-symmetric matrix with indeterminates as elements is not identically zero. This was the starting point for randomized parallel algorithms. In 1982 A. Borodin, J. von zur Gathen and J. Hopcroft [9] observed that it is possible, based on the Tutte's Theorem and on the fact that computation of the determinant of numerical matrix is in NC, to verify whether a given graph has a perfect matching. They proposed an *RNC* algorithm running in $O(\log^2 n)$ time.

In 1986 R. Karp, E. Upfal and A. Widgerson [35] published a paper that inspired other researchers to consider a parallel version of the matching problem. This paper was titled "Constructing a perfect matching is in Random NC" and it presented the first parallel algorithm to actually construct a perfect matching. This algorithm runs in $O(\log^3 n)$ time and uses $O(n^{6.5})$ processors.

In 1987 K. Mulmuley, U. V. Vazirani and V. V. Vazirani [44] improved R. Karp, E. Upfal and A. Widgerson's approach by presenting a randomized algorithm based on the Tutte's theorem.

All parallel matching algorithms for general case are in *RNC* class. It is not known whether it is possible to construct a deterministic parallel matching algorithm belonging to *NC*. Even a problem of constructing a single alternating path in a graph (an alternating path approach is commonly used by sequential matching algorithms) is not known to be in *NC*. That is why a lot of effort was spent on analyzing different subclasses of graphs.

In 1988 A. Moitra and R. C. Johnson [4] proposed a matching algorithm for interval graphs that runs in $O(\log^2 n)$ time and utilizes $O(n^6/\log n)$ processors. In 1989 E. Dahlhaus, M. Karpinski and A. Lingas [16] considered planar bipartite graphs. They proposed an $O((n/2-l+\sqrt{n})\log^7 n)$ algorithm utilizing $O(n^{1.5}\log^3 n)$ processors (l is the size of a maximum matching in a graph). More general class — bipartite graphs — was considered by Pranay Chaudhuri [11] in 1994. He proposed an $O(n\log\log^2 n)$ time algorithm that uses $O(n^3/\log\log n)$ processors. What is interesting, this algorithm is designed for a single instruction multiple data computation model (SIMD), so the approach proposed can be realized with nowadays GPUs. M. G. Andrews, M. J. Atallah, D. Z. Chen and D. T. Lee [4] came up in 1995 with reduction of a total work of the matching algorithm by A. J. Moitra and R. C. Johnson [4]. The former algorithm performs $O(n^5\log^2 n)$ total work, while the new algorithm only $O(n\log n)$. However, algorithm's execution time has been sacrificed and increased from $O(\log^2 n)$ to $O(\log^3 n)$.

In 1996 R. Sharan and A. Wigderson [54] presented a new approach to parallel computation of perfect matchings in cubic bipartite graphs. It is somehow similar to the sequential algorithm for computing perfect matchings in regular bipartite graphs by A. Schrijver [51]. The algorithm can be generalized so that it works for any regular bipartite graph, but it becomes very complicated — $O(\log^2 n)$ execution time and $O(n^{5.5})$ processors. In case of cubic graphs the requirement on the number of processors reduces to $O(n \log^* n / \log n)$.

Weighted case of the matching problem has been considered as well. In 2006 M. Fayyazi, D. Kaeli and W. Meleis [20] proposed a parametrized weighted matching algorithm for bipartite graphs running in time $O(n/w)$ utilizing $O(n^{max(2w,4+w)})$ processors ($w >= 1$). For $w = 1$ this leads to a linear time algorithm utilizing $O(n^5)$ processors.

Our results. The main result presented in this work is a new perfect matching algorithm for biconnected cubic graphs running in time $O(n \log^2 n)$. It is faster from it's predecessor by an $O(\log^2 n)$ factor and is much simpler to implement. It is possible to further reduce the time complexity to $O(n \log n \log \log^3 n)$ by utilizing randomized data structures. We also present a modification of our algorithm which allows for utilizing decremental dynamic graph connectivity data structures. There is a chance that it will be possible to implement more efficient decremental dynamic graph connectivity data structure than currently known fully-dynamic versions, which would lead to the faster version of our algorithm.

Some research has been done in the area of parallel matching algorithms. We present two new parallel algorithms for subclasses of cubic graphs:

- $O(\log^2 n)$-time perfect matching algorithm for bipartite cubic graphs utilizing $O(n/\log n)$ processors,

- $O(\log n \log^* n)$-time perfect matching algorithm for planar bipartite cubic graphs using $O(n)$ processors.

The first parallel algorithm has the same time complexity as the previously known fastest algorithm, but utilizes less processors. The second algorithm, on the other hand, is the fastest algorithm known to date for solving bipartite planar cubic case.

This work addresses also practical aspects of the maximum matching problem. We present two matching algorithms:

- parallel implementation of M. Mucha and P. Sankowski [43] maximum matching algorithm for general graphs adopted to a GPU architecture,

- sequential implementation of a perfect matching algorithm via reduction to the Boolean satisfiability problem.

Experimental evaluations prove that both algorithms are useful for some classes of graphs.

This work is based on two papers — [15] presents an $O(n \log^2 n)$-time matching algorithm for biconnected cubic graphs while [56] introduces parallel algebraic matching algorithm on a GPU.

Organization of this work. This thesis are divided into seven chapters, including this one. Chapter 2 presents some basic definitions and theorems which will be referenced latter on. Chapter 3 describes some state of the art matching algorithms which are addressed or improved in the following chapters. Chapter 4 addresses the problem of computing perfect matching in biconnected cubic graphs. First Frink's and Biedl's matching algorithms are presented. Description of our new $O(n \log^2 n)$ algorithm follows. Possible improvements of our algorithm are also suggested. Chapter 5 discusses parallel construction of matching for different classes of graphs. First our new matching algorithms for bipartite cubic and bipartite planar cubic graphs are presented. Parallel algebraic maximum matching algorithm for general class of graphs follows. Chapter 6 addresses practical aspects of computing matchings. Two new algorithms are introduced — a matching algorithm via Boolean satisfiability and an adoption of a parallel algebraic matching algorithm to a GPU architecture. The chapter is concluded with performance evaluation of different matching algorithms. Depending on the subclass of graphs being analyzed different matching algorithms are recommended. The last chapter lists some open problems related to our work. The paper is concluded with an Appendix, which contains source codes of some algorithms presented throughout the publication.

Acknowledgments. I would like to thank prof. Krzysztof Diks, my supervisor and co-author of the results presented in this work. His support during my PhD studies and help while writing this PhD thesis is invaluable. I would also like to thank my wife Marta for tolerating my limited involvement in the daily life when putting this work together.

Chapter 2

Definitions and preliminaries

In the current chapter we introduce definitions, naming conventions and notations. We also present some basic theorems that are crucial to our work.

For any *set* S, we denote by $|S|$ the *size* of S (the number of its elements). If S is a set and $|S| = n$, then $S[0], \dots, S[n-1]$ are elements of S in some, arbitrary order. A *multi-set* is a set in which elements are not necessarily unique. For a given two sets S_1 and S_2, $S_1 \otimes S_2$ is a symmetric difference of those sets — it consists of all elements which are in one of the two sets and not in their intersection.

2.1 Graphs

Let $G = (V, E)$ be a *graph* with *vertex* set $V = \{v_1, v_2, \dots, v_n\}$ and *edge* multi-set $E = \{e_1, e_2, \dots, e_m\}$. Each edge $e = v - w \in E$, $v, w \in V$, connects a pair of vertices — *endpoints* of the edge. We say that e is *incident* to u and v. Any two vertices connected by an edge are *adjacent*. An edge connecting vertex v to itself ($e = v - v$, $v \in V$) is called a *loop*. A *multigraph* is a graph in which any pair of vertices can be connected by more than one edge. All graphs in this paper are undirected *multigraphs* without *loops*. We use n to denote the number of vertices of a graph and m to denote the number of its edges.

For a given graph $G = (V, E)$ and a vertex u, $N_v(G, u)$ denotes the set of all vertices adjacent to u in G ($N_v(G, u) = \{w \in V : \exists u - w \in E\}$), $N_e(G, u)$ denotes the set of all edges incident to u ($N_e(G, u) = \{u - w \in E\}$, $w \in V$). Analogously, for a given edge $e = u - w$, $N_v(G, e)$ denotes the set of all vertices incident to e, namely $\{u, w\}$, $N_e(G, e)$ denotes the set of all edges adjacent to e ($N_e(G, e) = N_e(G, u) \cup N_e(G, w) - e$). We extend the above definitions to the sets of vertices and edges. For any $U \subseteq V \cup E$, $N_v(G, U)$ denotes the set of all vertices adjacent to any vertex or incident to any edge $u \in U$ ($N_v(G, U) = \cup_{u \in U} N_v(G, u)$); $N_e(G, U)$ is the set of all edges incident to any vertex or adjacent to any edge $u \in U$ ($N_e(G, U) = \cup_{u \in U} N_e(G, u)$). When it is clear from the context to which graph we are referring to, instead of writing $N_v(G, U)$ ($N_v(G, u)$) and $N_e(G, U)$ ($N_e(G, u)$) we simply write $N_v(U)$ ($N_v(u)$) and $N_e(U)$ ($N_e(u)$).

Degree of a vertex v (denoted by $d(v)$) is the number of edges adjacent to v. Note that $d(v)$ not necessarily equals $|N_v(v)|$, as graph may have multiple edges.

For any graph $G(V,E)$ and $U \subseteq V$, $G[U]$ denotes the *vertex-induced subgraph* of G, i.e. $G[U] = G'(U,E')$, $E' = \{u - v \in E : u,v \in U\}$. $G - U$ is the graph obtained from G by removing all vertices of U ($G - U = G[V - U]$). For a given graph $G(V,E)$, a graph induced by a vertex set $V' \subseteq V$ and an edge set $E' \subseteq E$ (written as $G[V',E']$) is a graph with vertices from V' and all edges from E' that have both ends in V' ($E'' = \{u - v \in E' : u,v \in V'\}$.

Let $G(V,E)$ be a graph. A *path* $v_1 - v_2 - \cdots - v_l$, $v_k \in V, k \in \{1 \dots l\}$, is a sequence of vertices of the graph, such that each consecutive pair of vertices is connected by an edge ($v_k - v_{k+1} \in E, k \in \{1 \dots l-1\}$). The *length* of a path is the number of edges that it uses. A path is called *simple* if all its vertices are unique. Any path $v_1 - v_2 - \cdots - v_1$ of length at least 2 is called a *cycle*. A cycle is *simple* if all its vertices, excluding the first and last vertex, are unique.

A graph $G = (V,E)$ is *connected* if for any pair of vertices $v,w \in V$ there exists a path $v - \cdots - w$. Every maximal connected subgraph of a graph is called a *connected component*. Two vertices belonging to the same connected component are connected.

For a given graph $G = (V,E)$, a *bridge* (or a *cut-edge*) $e \in E$ is an edge which removal increases the number of connected components of G. Graph is *bridgeless* if it contains no bridges. An *articulation point* (or a *cut-vertex*) $v \in V$ is a vertex of the graph which removal increases the number of connected components of G. A connected graph is *vertex biconnected* (*edge biconnected*) if removal of any vertex (edge) leaves the graph connected. If G is both vertex and an edge biconnected we call it simply *biconnected*.

A graph $G = (V,E)$ is called *regular* if there exists $k > 0$ such that $\forall_{v \in V} d(v) = k$. A *cubic* graph is a regular graph of degree 3. An *almost cubic graph* is a graph with all vertices of degree not greater than 3.

A graph $G = (V,E)$ is *bipartite* if there exists $V_1, V_2 \subseteq V$ such that $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$ and $\forall_{u-v \in E} u \in V_1, v \in V_2$. In other words, it is possible to split the set of vertices of the graph into two subsets V_1 and V_2 in such a way that all edges of the graph have one endpoint in V_1 and the other endpoint in V_2. (V_1, V_2) is called the *bipartition* of G.

A *matching* of a graph $G = (V,E)$ is a set $M \subseteq E$ of edges such that no two distinct edges in M have a common endpoint. *Maximal matching* is a matching that cannot be extended into a bigger matching by adding some edge $e \in E$. *Maximum matching* is a matching of maximum possible cardinality. *Perfect matching* is a matching of size $\frac{|V|}{2}$, i.e. every vertex of the graph has exactly one incident edge in M. An *allowed edge* is an edge of the graph which belongs to some its maximum matching. For a given matching $M \subseteq E$ and edge $e \in E$ we say that e belongs / is *covered* by M if $e \in M$. Analogically, for a given vertex $v \in V$ we say that v belongs / is *covered* by M if $v \in N_v(G,M)$. Any vertex that does not belong to the matching is called *free*.

An *alternating path* with respect to the matching M is a simple path in which every second edge belongs to M. An *augmenting path* is an alternating path with both ends not belonging to M.

By $c_o(G)$ we denote the number of odd (size) connected components in G.

2.1.1 General theorems without proofs

Theorem 2.1 (*Tutte*)**.** *Graph $G = (V, E)$ has a perfect matching if and only if for every $U \subseteq V$ the number of odd connected components in $G[V - U]$ is less than or equal to the cardinality of U ($\forall_{U \subseteq V} c_o(G[V - U]) \leq |U|$).*

Proof. For a proof please refer to [31]. □

Theorem 2.2 (*Hall*)**.** *Let $G = (V, E)$ be a bipartite graph with bipartition (V', V''), $|V'| = |V''|$. G contains a perfect matching if and only if*

$$\forall_{S \subseteq V'} |S| \leq |N_v(G, S)|.$$

Proof. For a proof please refer to [14]. □

Theorem 2.3 (*König*)**.** *Every regular bipartite graph with vertices of degree at least* 1 *has a perfect matching.*

Theorem 2.4 (*Euler*)**.** *Let $G = (V, E)$ be a planar connected graph and P an embedding of G in a plane. Then $n - m + f = 2$, where n is the number of vertices in a graph, m is the number of edges, f is the number of faces.*

Theorem 2.5. *Let $G = (V, E)$ be a planar connected graph with at least* 3 *vertices. Then $|E| \leq 3|V| - 6$.*

2.1.2 Cubic graphs' theorems

In this section we present a number of facts about cubic graphs that will be referenced in the remaining chapters.

Lemma 2.6. *Every cubic graph has an even number of vertices.*

Proof. The number of edges in a cubic graph with n vertices equals $m = \frac{3}{2}n$, hence $2m = 3n$. As m is an integer then n has to be even. □

Lemma 2.7. *Cubic graph without bridges does not have articulation points.*

Proof. Let us assume otherwise that there exists a cubic graph $G = (V, E)$ which does not have bridges, but there is a vertex $v \in V$ which is an articulation point. Removal of v from G leads to a graph $G' = G[E - \{v\}]$ consisting of at least two connected components. As G is cubic and v is incident to up to 3 vertices in G, there exists a connected component V' in G' which is connected to v by exactly one edge e. Removal of e from G splits graph into two components V' and $G - V'$, hence e is a bridge — a contradiction. □

Lemma 2.8. *Every connected cubic graph without bridges is biconnected.*

Proof. It follows from Lemma 2.7 that every bridgeless cubic graph has no articulation points, hence it is biconnected. □

Lemma 2.9. *Every cubic bipartite graph is biconnected.*

Proof. Let us assume otherwise that there is a cubic bipartite graph $G = (V, E)$ which is not biconnected. Let (V_1, V_2) be the bipartition of G. It follows from lemma 2.7 that G has a bridge e. $G[V, E - \{e\}]$ contains a connected component G' connected to the rest of the graph G by e. Let (V_1', V_2') be a bipartition of G'. Assume (without loss of generality) that the bridge e is incident to V_1' in G'. As G is cubic there are $3|V_1'|$ edges leaving vertices of V_1'. Similarly, there are $3|V_2'|$ edges leaving vertices of V_2'. As all but one (namely e) edges connect vertices of V_1' with V_2', $3|V_1'| = 3|V_2'| - 1$ holds — a contradiction. $\square$

2.2 Linear algebra

Symbol $\mathbb{R}$ stands for the set of all real numbers, $\mathbb{Z}$ stands for the set of all integers and $\mathbb{N}$ stands for the set of all natural numbers. For a given prime number p, $\mathbb{Z}_p$ stands for the finite field of cardinality p consisting of elements $0, 1, \ldots, p-1$. All arithmetic operations in $\mathbb{Z}_p$ are performed modulo p. For any ring R, $R^{n \times m}$ stands for the set of all $n \times m$ matrices with elements from R. For a given matrix $M \in R^{n \times m}$ with n columns and m rows by $M[x, y], x \in \{0, 1, \ldots n-1\}, y \in \{0, 1, \ldots, m-1\}$ we denote an element of M located in $x'th$ column and $y'th$ row.

A *zero matrix* is a matrix with all elements equal zero. A *zero matrix* of size $n \times m$ is written as $O_{n \times m}$. I_n denotes a square *identity matrix* of size $n \times n$ which has ones on the main diagonal and zeros elsewhere. These notations will be frequently used in the pseudo-codes of the algebraic algorithms presented throughout this paper.

$M' = M \setminus [x, y]$ is a sub-matrix of M obtained from M by removing $x'th$ column and $y'th$ row. $M[x, *]$ and $M[*, y]$ denote, respectively, x'th column and y'th row of the matrix M.

Chapter 3

State of the art

In this chapter we sketch some state of the art matching algorithms. We start with the description of an augmenting path approach introduced by Edmonds [17]. This is the general approach utilized by the fastest sequential algorithms for constructing maximum matching in both bipartite and general graphs. Latter we present Schrijver's algorithm [51] for constructing perfect matchings in bipartite regular graphs. When applied to bipartite cubic graphs it provides a linear method for matching construction. We conclude this chapter with a brief description of Sharan's parallel matching algorithm [54] for bipartite cubic graphs. Some more state of the art results are presented throughout the remaining chapters, including Frink's [23] and Biedl's [6] algorithms for perfect matching construction in biconnected cubic graphs and Mucha and Sankowski algebraic algorithm [43] for general graphs.

3.1 Augmenting path approach

The commonly used approach for constructing maximum matchings in graphs is the one proposed by Edmond's [17]. The concept is based on the following theorem

Theorem 3.1 (Edmonds)**.** *Let $G = (V, E)$ be a graph and M be a matching of G ($M \subseteq E$). M is a maximum matching of G if and only if there is no augmenting path in G related to M.*

Proof. For a proof of this theorem please refer to [17] □

All algorithms based on Edmond's approach follow the general scheme:

- Construct an initial matching M of G.
- Until there is no augmenting path in G related to M:
  - Find an augmenting path P in G related to M.
  - Set $M = M \otimes P$ which increases the size of M by 1.

Searching for an augmenting path P is especially simple in case of bipartite graphs. Let $G = (V, E)$ be a bipartite graph with bipartition (V', V''). As each augmenting path is of odd length, one end of P is in V' while the other in V''. To find P it is sufficient to perform a graph traversal starting from all vertices $v \in V'$ not covered by M and alternately visit edges from $E - M$ (when going from V' to V'') and edges from M (when going from V'' to V'). Once a vertex $v \in V''$ not covered by M is encountered, an augmenting path is found. A naive implementation of Edmond's approach for bipartite graphs requires up to $|V|/2$ graph traversals, hence the total complexity of such an algorithm is $O(|V||E|)$. By applying a BFS traversal it is possible to find a number of disjoint augmenting paths at once, which reduces the total number of phases. This way $O(\sqrt{|V|}|E|)$ complexity can be obtained. To further speed up the algorithm it is possible to apply some fast heuristics for constructing an initial matching. This way search for augmenting paths starts with some non-empty initial matching, so less augmenting paths need to be found.

In case of general graphs the implementation of Edmond's approach is much more complicated. Due to the odd cycles that can be encountered during the graph's traversal it is required to handle such a special cases accordingly. Edmond proposed a solution to the problem which shrinks odd cycles to a single vertices (this process is known as shrinking *blossoms*), so that the graph traversal can continue. Once an augmenting path is found, all blossoms are expanded back to their original form. Apart from the fact that implementation of general case maximum matching algorithm is much more complicated than the bipartite case, it has been proved that the same $O(\sqrt{|V|}|E|)$ time complexity can be obtained.

3.2 Schrijver's algorithm

We already know from König's Theorem 2.3 that every k-regular bipartite graph has a perfect matching. This is equivalent to the fact that edges of such a graph can be colored using k colors so that no two incident edges have the same color. In 1999 A. Schrijver [51] presented an algorithm for k-edge coloring of such graphs in $O(km)$ time. This algorithm provides a linear method for constructing the perfect matching in cubic, bipartite graphs. The core element of the Schrijver's algorithm is a $O(km)$ subroutine for constructing a perfect matching in a regular bipartite graph, which is very elegant and simple to implement:

- Initialize a weight function $w : E \to \mathbb{Z}$ such that $w(e) = 1, e \in E$.
- While G contains a cycle $C = c_0 - c_1 - \cdots - c_{k-1} - c_0$, $w(c_k - c_{(i+1)\%k}) > 0, i \in \{0, 1, \ldots k-1\}$:
  - Split edges of C into two matchings C_1 and C_2.
  - If the sum of weights of the edges from C_1 is smaller than the sum of the edges from C_2 then swap C_1 with C_2.
  - Increase the weight of each edge in C_1 by 1 and decrease the weight of each edge in C_2 by 1.

- Construct a perfect matching by selecting all edges $e \in E$ for which $w(e) > 0$.

We will now prove the correctness and time complexity of this algorithm.

Theorem 3.2. *Schrijver's algorithm terminates after a finite number of steps.*

Proof. Consider the following expression:

$$F(w) = \sum_{e \in E} w(e)^2$$

Notice that $F(w) \leq k^2 m$, as $w(e) \leq k, e \in E$. Once a cycle is found the sum of weights of edges from C_1 is not smaller than the sum of weights of edges from C_2. When the weights are updated, the value of $F(w)$ increases by at least $|C|$:

$$\sum_{e \in C_1}((w(e)+1)^2 - w(e)^2) + \sum_{e \in C_2}((w(e)-1)^2 - w(e)^2) =$$

$$2(\sum_{e \in C_1} w(e) - \sum_{e \in C_2} w(e)) + |C_1| + |C_2| \geq |C|$$

Each update of the edges' weights results in the increase of the value of $F(w)$ and as the $F(w)$ is bounded from the top, the algorithm terminates. $\square$

Theorem 3.3. *Schrijver's algorithm computes a perfect matching of* G.

Proof. It follows from Theorem 3.2 that Schrijver's algorithm terminates after a finite number of steps. Once it does a graph $G' = G[V, E'], E' = \{e \in E : w(e) > 0\}$ contains no cycles (as algorithm would not terminate otherwise), hence G' is a set of trees. We show by induction on $|V|$ that G' is a set of paths of length 1.

The only 2-vertex graph G' without cycles consists of 2 vertices connected by a single edge e, $w(e) = k$. Let G' be an $n+2$-vertex graph without cycles. There exists a vertex v in G' of degree 1 (as G' is a forest) connected by an edge $e = v - w$ to the rest of the graph. As $d(v) = 1$, edge e is of weight k. It follows that w is also of degree 1 in G'. By removing vertices v and w from G' we obtain an n-vertex graph and by the induction hypothesis we get that G'' is a set of paths of length 1. Hence G' is also a set of paths of length 1. $\square$

Theorem 3.4. *It is possible to implement Schrijver's algorithm in* $O(km)$ *time.*

Proof. It follows from Theorem 3.3 that once Schrijver's algorithm terminates the value of $F(w) = \sum_{e \in E} w(e)^2$ equals to $\frac{1}{2}k^2 n = km$. A single update of weights along a cycle C increases the value of $F(w)$ by at least $|C|$, so to implement Schrijver's algorithm in $O(km)$ time it is sufficient to detect cycles in G and update weights of edges in time proportional to the total length of all detected cycles. It is possible to achieve it by applying a DFS traversal. Start the search process from any edge e such that $w(e) > 0, w(e) < k$. Such an edge is incident with another edge which weight is grater than 0 but smaller than k, so it is possible to continue the DFS search until a cycle is encountered. Once it happens update the weights of edges on a cycle, remove all edges of a cycle from a DFS stack and proceed with the DFS search. Once the stack is empty start with another edge e. It is clear that the total execution time of such an algorithm is linear in the total length of all cycles detected. $\square$

3.3 Sharan's algorithm

Sharan's parallel matching algorithm is based on the similar idea as Schrijver's algorithm, but is much more complicated. Two definitions need to be introduced before we describe its structure.

A *pseudo-perfect matching* of a graph G is a subgraph M of G spanning all vertices of G such that every vertex c is of odd degree in M. For a given rooted tree T with t vertices (t $>$ 2), a $\frac{1}{3} - \frac{2}{3}$ *cut-vertex* is a vertex of T which has at least $\frac{t-1}{3}$ descendants and at most $\frac{2t}{3}$ descendants, including itself.

The Sharan's algorithm starts by constructing a pseudo-perfect matching M and then modifies it until a perfect matching is obtained. In case of cubic graphs pseudo-perfect matching construction is simple — all edges of the graph can be selected. As the process of converting of the pseudo-perfect matching to a perfect matching needs to happen in parallel, the algorithm becomes complicated. We present only a short summary of this algorithm to show the level of complication. For more detailed description and the proof of correctness please refer to [51].

The general structure of Sharan's algorithm is the following:

- Construct a pseudo-perfect matching M of G.
- Convert M into a spanning forest of G in such a way that it remains a pseudo-perfect matching of G.
- Convert each tree of M into a vertex-induced subgraph of G in such a way that M remain a pseudo-perfect matching of G.
- Until M is a perfect matching:
  - Find a perfect matching N in the complement of M.
  - Find an augmenting cycle L in $M \cup N$, such that $|M \otimes L| \leq (1 - c)|M|$, for a constant $c > 0$.
  - $M = M \otimes L$.

To convert a pseudo-perfect matching M into a spanning forest of G, Sharan's algorithm selects an arbitrary spanning forest $T = (V', E')$ of G and computes a symmetric difference of M and the set S of all fundamental cycles of G with respect to T.

The next phase of the algorithm converts a pseudo-perfect matching M, which is also a forest in G, into a set of induced graphs of G. This is achieved by the function $Induced(T)$:

- while M is not induced, in parallel for every tree $T(V', E')$ in M which is not induced:
  - Root T arbitrarily at a vertex which is of degree 3 in M.
  - Find a $\frac{1}{3} - \frac{2}{3}$ cut-vertex of T, denote it by v.

  - Search for a cycle C in $G[V']$ which includes v.
  - If C exists then let $M = M \otimes C$, otherwise $M = (M - T) \cup \mathit{Induced}(T - \{v\}) \cup T[S(v)]$.

Once an induced forest M of G, which is also a pseudo-perfect matching, is obtained, the last phase of the algorithm is to remove some edges of M, so that only perfect matching remains. This is done by logarithmic number of iterations of the following steps:

- Compute a perfect matching N in the complement of M.
- Let $H = M \cup N$.
- Compute a spanning forest T of H.
- Compute symmetric difference of M and a set of all fundamental cycles of H.

As Sharan showed in his paper, it is possible to implement this algorithm for bipartite cubic graphs in $O(\log^2 n)$ time using $(n \log^* n / \log n)$ processors.

Chapter 4

Matching in biconnected cubic graphs

In this chapter we present our new algorithm for constructing a perfect matching in biconnected cubic graphs. It runs in $O(n \log^2 n)$ time. We begin with a presentation of two historical results which make it easier to follow our approach. The first result is $O(n^2)$ time algorithm by Frink [23] presented in section 4.1. It directly derives from the Frink's proof of the Petersen's theorem. In section 4.2 we present an $O(n \log^4 n)$ matching algorithm of Biedl et al. [6]. This algorithm was the fastest matching algorithm for biconnected cubic graphs known to date. The presentation of our result follows in section 4.3, where we also introduce a randomized version of the algorithm running in expected $O(n \log n \log\log^3 n)$ time. We conclude this chapter in section 4.4 with a significant modification of our algorithm, which makes it possible to utilize decremental dynamic graph connectivity data structure (instead of the fully-dynamic version, utilized by the algorithm from the section 4.3).

4.1 Frink's algorithm

We are interested in constructing an efficient perfect matching algorithm for biconnected cubic graphs. How do we know that such a matching exists in the first place? This fact was proved by Jules Petersen [48] in 1891.

Theorem 4.1 (*Petersen*)**.** *Let $G = (V, E)$ be a bridgeless cubic graph. G has a perfect matching.*

Proof. Consider an arbitrary subset $U \subseteq V$. Let $C = \{C_1, C_2, \ldots, C_l\}$ be the set of all connected components of $G - U$ of odd sizes. Every $C_k \in C$ is connected with U by an odd number of edges, since G is cubic and $|C_k|$ is odd. As G is bridgeless, C_k is connected to U by at least 3 edges. Hence, the number of edges leaving U is at least $3|C|$ and (as G is cubic), $|C| \leq |U|$. It follows that for every $U \subseteq V$ $c_o(G - U) \leq |U|$ and the Tutte's theorem (2.1) concludes the proof. $\square$

The original proof of Petersen's theorem from the 19th century is much more complicated, as the Tutte's Theorem was not known. H. R. Brahana [10] in 1917 followed by A. Errera in 1922 managed to simplify the original proof. Finally in 1926

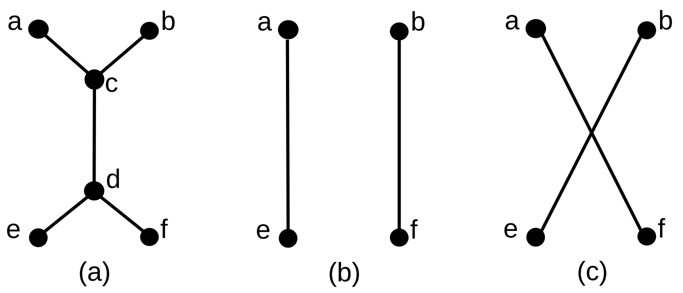

Figure 4.1: Frink's reduction of a biconnected cubic graph. **(a)** Vertices c and d are to be removed from the graph. **(b)** The first type of reduction — vertex a is connected with e, vertex b is connected with f. **(c)** The second type of reduction — vertex a is connected with f, vertex b is connected with e.

Frink [23] came up with a constructive proof that resulted in a matching algorithm running in $O(n^2)$ time. The core of this algorithm is the following Frink's Theorem:

Theorem 4.2 (*Frink*). *Let $G(V, E)$ be a biconnected cubic graph. Consider an edge $p = c - d$ such that $|N_v(c)| = 3$ and $|N_v(d)| = 3$. Let a and b be the neighbors of vertex c in G different from d. Let e and f be the neighbors of d different from c (see Figure 4.1). At least one of the two reductions of graph G consisting of removing vertices c and d and reconnecting vertices a, b, e and f by edges $a - e$ and $b - f$ or $a - f$ and $b - e$ (cases (b) and (c) from the Figure 4.1) leads to a biconnected cubic graph. If $|N_v(c)| \neq 3$ or $|N_v(d)| \neq 3$ similar reduction of a graph is possible (see Figure 4.2).*

Proof. By A, B, E and F we denote connected components of $G[V - \{c, d\}]$ containing, respectively, vertices a, b, e and f. As G is bridgeless (and hence p is not a bridge)

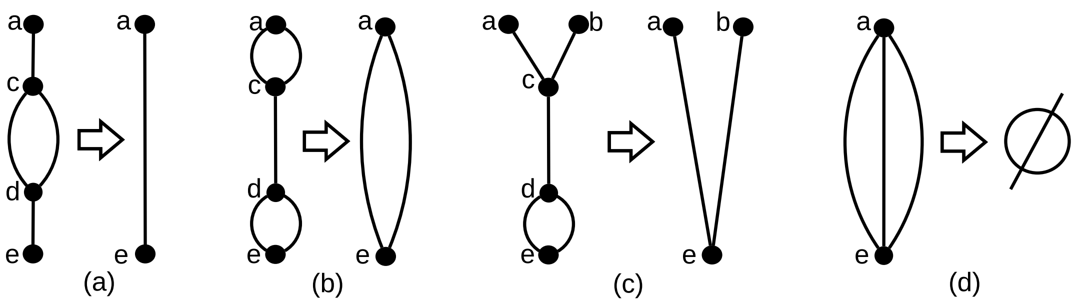

Figure 4.2: Special cases of the Frink's reductions. **(a)** Edge $c - d$ being reduced is double. Reduction removes vertices c and d from the graph and reconnects a and e with a new single edge. **(b)** Edge $c - d$ is incident to 2 double edges $a - c$ and $d - e$. Reduction removes vertices c and d from the graph and adds a new double edge $a - e$. **(c)** Edge $c - d$ is incident to 1 double edge $d - e$. Reduction removes vertices c and d from the graph and introduces two new edges $a - e$ and $b - e$. **(d)** Edge $c - d$ being reduced is a triple edge. Reduction removes vertices c and d.

there must be an edge connecting one of A or B to E or F. Without loss of generality let us assume that A is connected with E (see Figure 4.3). Furthermore edges $b-c$ and $d-f$ are not bridges in G, hence one of the three cases is possible (with respect to isomorphism) — see Figure 4.3:

- component B is connected to F,
- component B is connected to E and component E is connected to F,
- component A is connected to B and component E is connected to F.

In all three cases if $G[V-\{c-d\}]$ is extended with edges $a-f$ and $b-e$ (obtaining graph G'), the added edges lie on some cycle in G' (see Figure 4.4). Also, for any pair of vertices $u, v \in \{a, b, e, f\}$ there exists a cycle in G' containing u and v. To prove that G' is biconnected it is sufficient to show that every edge r of G' lies on some cycle of G'. Let C be a cycle in G containing r (such a cycle exists as G is biconnected). If C does not pass through c nor d then C is also a cycle in G'. If it does pass through c or d then construct a cycle $C' \subset G'$ from C in the following way:

- if $x-c-d-y \in C, x \in \{a,b\}, y \in \{e,f\}$, remove $x-c-d-y$ from C and add any path from x to y in G' not containing r (such path always exists as x and y belong to some cycle in G'),
- if $a-c-b \in C$, remove $a-c-b$ from C and add any path from a to b in G' not containing r,
- if $e-d-f \in C$, remove $e-d-f$ from C and add any path from e to f in G' not containing r.

C' is a set of cycles (as C' was obtained from C by replacing some sub-paths) and it contains r. It follows that every edge of G' lies on some cycle, so the graph does not contain any bridges. From the fact that G' is connected and from Lemma 2.8 we know that G' is biconnected. □

Frink's theorem allows us to construct a relatively simple perfect matching algorithm for biconnected cubic graphs. We start with an input graph G and until there are no vertices left we perform Frink's reduction against an arbitrary edge of the graph. Once the graph contains no vertices we construct an initial empty perfect matching M and start the reversion process of all reductions (starting from the vertices deleted most recently). Each reversion step can lead to one of the four basic cases presented in the Figure 4.5 or to one of the special cases from the Figure 4.6. Reversion process of the special cases, as well as the first three basic cases, is straightforward. They can be performed in $O(1)$ time. The only problematic case is when both edges belong to the perfect matching. In such a situation it is required to find an alternating cycle containing at least one of these edges and update the matching with the cycle. This operation reduces the 4th case to one of the first three.

It is possible to implement Frink's algorithm in $O(n^2)$ time. There are $O(n)$ reduction / reversion steps, each of them requires $O(n)$ operations to be performed.

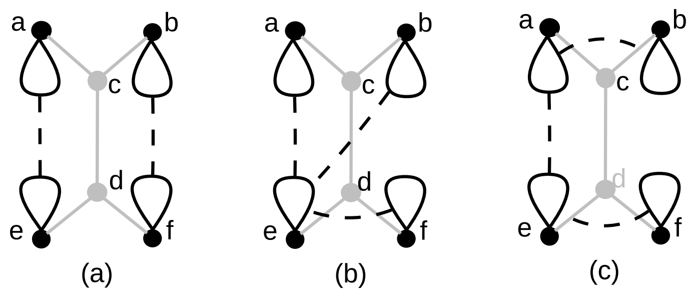

Figure 4.3: All possible connections of biconnected components of $G[V-\{c-d\}]$. **(a)** Component A is connected to component E, component B is connected to component F. **(b)** Component E is connected to components A, B and F. **(c)** Component A is connected to components B and E, component E is connected to component F.

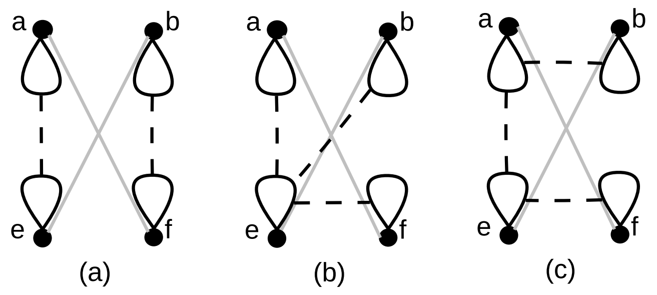

Figure 4.4: Possible connections of connected components A, B, C and D after removal of edge $c-d$ and addition of edges $a-f$ and $b-e$.

Reduction step performs an arbitrary reduction (case (b) or (c) from the Figure 4.1) and verifies in $O(n)$ time whether a graph remains biconnected (linear DFS-based algorithm for computing biconnected components of a graph can be found in [32]). If the graph is not biconnected, the other reduction needs to be selected.

The first three cases of the reversion process are simple $O(1)$-time procedures. Only the last case requires $O(n)$ time to find an alternating cycle in a graph. The algorithm for constructing such a cycle can be found in [6, 57].

The implementation of the Frink's algorithm — *frink_matching* function, is presented as Algorithm 1. It utilizes four functions — *reductions*, *simple_reversion*, *bridgeless* and *alternating_cycle*. The first two functions are presented, respectively, as Algorithms 2 and 3. *Bridgeless* function takes as a parameter a graph and reports whether it contains a bridge. *Alternating_cycle* function takes three parameters — a graph $G = (V, E)$, perfect matching $M \subset E$ and an edge $e \in M$. It returns an alternating cycle $C \subset E$ related to the matching M such that $e \in C$. All of the above functions but *alternating_cycle* will be used by the matching algorithms presented in the next sections. Implementation of the *bridgeless* function will vary from algorithm to algorithm, leading to different overall performances.

From the performance analysis of Frink's algorithm it turns out that there are

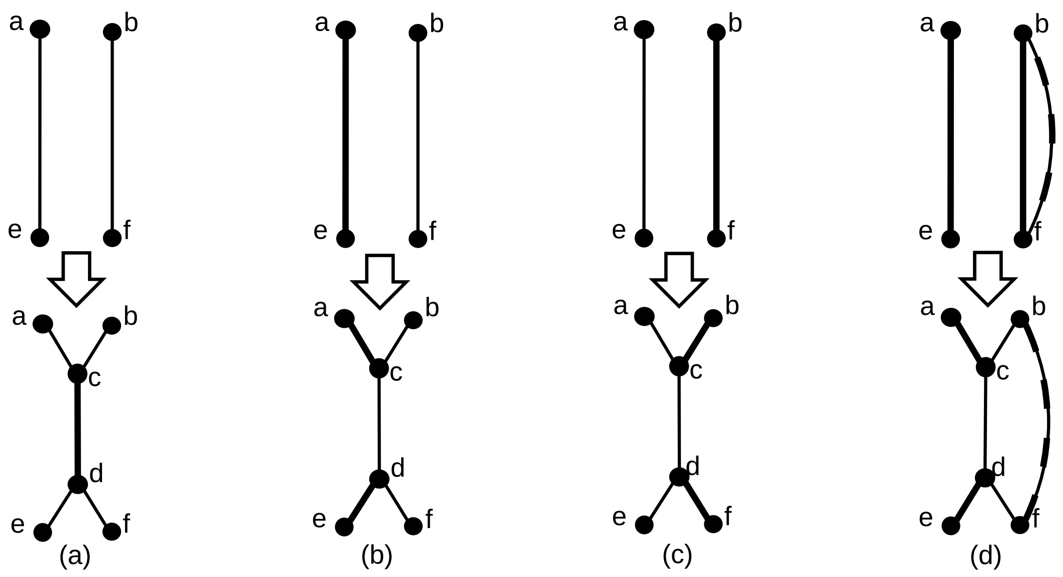

Figure 4.5: Reversion process of the basic Frink's reduction. **(a)** None of the edges being reverted are matched — add edge $c-d$ to the matching. **(b)** Edge $a-e$ is in the matching — remove $a-e$ from the matching and add $a-c$ and $d-e$ to the matching. **(c)** Edge $b-f$ is in the matching — remove $b-f$ from the matching and add $b-c$ and $d-f$ to the matching. **(d)** Both edges $a-e$ and $b-f$ are in the matching — find an alternating cycle C containing at least one of these two edges ($b-\cdots-f$ in the example) and update the matching with C. This way case (d) is reduced to one of the cases (a), (b) or (c).

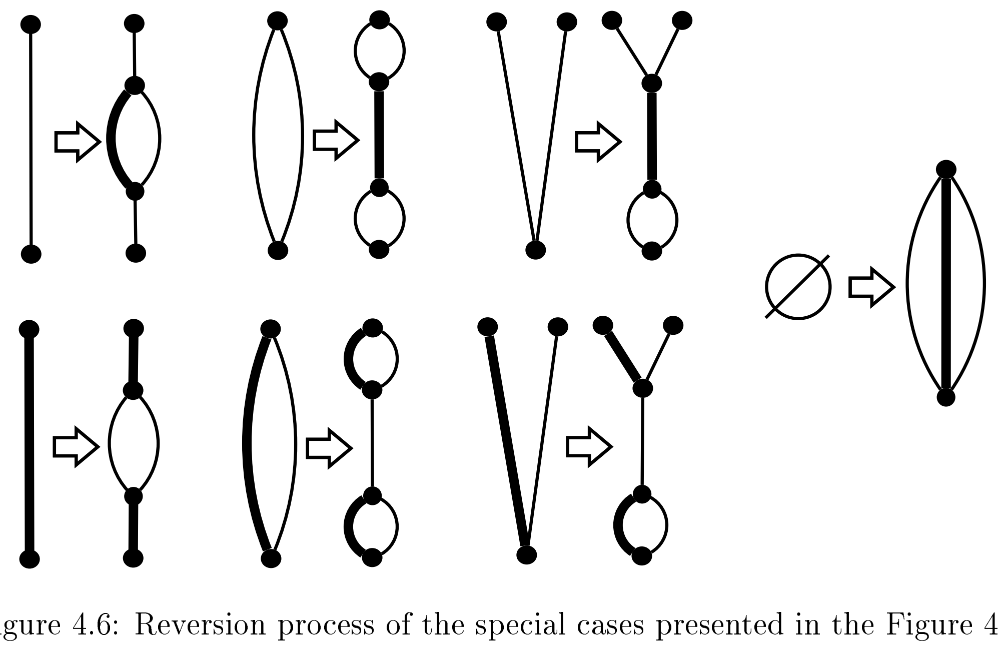

Figure 4.6: Reversion process of the special cases presented in the Figure 4.2.

two reasons for this algorithm not to run in linear time. The first reason is that each reduction step performs linear test of graph's biconnectivity. The second reason is the 4th case of the reversion process. It turns out, however, that it is possible to get rid of the first limitation by applying recent results on dynamic graph data-structures proposed by J. Helm, K. de Lichtenberg and M. Thorup [30]. The authors introduced, among other dynamic algorithms, a data structure that for a given graph G allows for insertion / deletion of vertices and edges. It also answers the question whether G is biconnected. Each operation against the data structure takes $O(\log^4 n)$ time. The only remaining obstacle to reduce the Frink's algorithm complexity to $o(n^2)$ is the 4th case of the reversion stage.

Algorithm 1 *frink_matching*(G)

Require: Biconnected cubic graph $G = (V, E)$.
Ensure: Perfect matching of G.

```
1: if $|V| = 0$ then
2:     return $\emptyset$
3: else
4:     $v - w = E[0]$
5:     $R = reductions(G, v - w)$
6:     if $bridgeless(G[V - \{v, w\}, E \cup R[0]])$ then
7:         $r = R[0]$
8:     else
9:         $r = R[1]$
10:    end if
11:    $M \leftarrow frink\_matching(G[V - \{v, w\}, E \cup r])$
12:    if $|r \cap M| = 2$ then
13:        $C \leftarrow alternating\_cycle(G, M, r[0])$
14:        $M \leftarrow M \oplus C$
15:    end if
16:    $M \leftarrow (M - r) \cup simple\_reversion(G, v, w, r, M)$
17:    return $M$
18: end if
```

4.2 Biedl's algorithm

T. Biedl et al. [6] proposed an algorithm for constructing perfect matching in biconnected cubic graphs in $O(n \log^4 n)$ time. This algorithm is a clever modification of the Frink's algorithm which makes use of the following Lemma:

Lemma 4.3. *For an arbitrary edge e of a biconnected cubic graph G there exists a perfect matching in G not containing e.*

Proof. As G is a biconnected cubic graph, according to Theorem 4.1 it has a perfect

Algorithm 2 *reductions*(G, e)

Require: Biconnected cubic graph $G = (V, E)$.
Require: Edge $e = v - w \in E$.
Ensure: List of possible reductions against the edge e. Each reduction is represented as a set of edges to be added to the graph.

```
1:  if $v - w$ is a triple edge in $G$ then
2:      return $[\emptyset]$
3:  else
4:      $V' \leftarrow N_v(G, v) - w$
5:      $W' \leftarrow N_v(G, w) - v$
6:      if $|V'| = 1$ and $|W'| = 1$ then
7:          return $[\{V'[0] - W'[0]\}]$
8:      else if $|V'| = 2$ and $|W'| = 1$ then
9:          return $[\{V'[0] - W'[0], V'[1] - W'[0]\}]$
10:     else if $|V'| = 1$ and $|W'| = 2$ then
11:         return $[\{V'[0] - W'[0], V'[0], W'[1]\}]$
12:     else
13:         return $[\{V'[0] - W'[0], V'[1] - W'[1]\}, \{V'[0] - W'[1], V'[1] - W'[0]\}]$
14:     end if
15: end if
```

matching M. If M contains e then we can construct an alternating cycle C containing e and update matching M with C, so that M does not contain e. □

By utilizing Lemma 4.3 it is possible to get rid of the 4th reversion case from the Frink's algorithm. Biedl's algorithm does not perform reduction against arbitrary edge of a graph. Instead, it first selects an arbitrary edge e and tries to construct a perfect matching not containing e (from 4.3 it is known that such a matching exists). We will refer to edge e as *excluded* from the matching being constructed. Each Frink's reduction step is performed against an edge f incident to e in $O(\log^4 n)$ time by utilizing the dynamic graph biconnectivity data structure [30].

After reduction takes place (edge e has been deleted) an edge e' — a newly added edge incident to the endpoint of e — becomes excluded from the matching being constructed.

It follows from Theorem 4.1 and 4.3 that reduced graph has a perfect matching not containing e'. Frink's reductions are applied over and over again until graph is empty. When reduction steps are being reverted, e' never belongs to the perfect matching, hence the 4th case of the Frink's reversions does not occur.

The above modification of the Frink's algorithm leads to the $O(n \log^4 n)$-time algorithm. Implementation of the Biedl's algorithm is presented as Algorithm 4. It computes a perfect matching for a given bridgeless cubic graph G not containing specified edge e. *Biedl_matching* reuses *reductions*, *bridgeless* and *simple_reversion* functions. *Bridgeless* function, however, is implemented differently then in case of Frink's algorithm — it utilizes dynamic graph biconnectivity data structure [30].

Algorithm 3 $simple\_reversion(G, v, w, R, M)$

Require: Cubic biconnected graph $G = (V, E)$.
Require: Pair of vertices v and w being added to G by the reversion.
Require: Set of edges R being removed from G by the reversion.
Require: Perfect matching M being extended by introduction of vertices v and w.
Ensure: Set of edges to be added to the matching M such that M remains a perfect matching.

```
1: $V' \leftarrow N_v(G, v) - w$
2: $W' \leftarrow N_v(G, w) - v$
3: if $R \cap M = \emptyset$ then
4:     return $\{v - w\}$
5: else if $|R| = 1$ then
6:     return $\{v - V'[0], w - W'[0]\}$
7: else
8:     if $R[0] \in M$ then
9:         $u' - w' = R[0]$
10:    else
11:        $u' - w' = R[1]$
12:    end if
13:    if $u' \in W'$ then
14:        $(u', w') = (w', u')$
15:    end if
16:    return $\{u - u', w - w'\}$
17: end if
```

4.3 The new algorithm

It turns out that it is possible to perform an efficient graph biconnectivity testing for each reduction performed by the Frink's algorithm without utilizing complex dynamic biconnectivity data structures. By taking into account some biconnected cubic graphs' properties and by applying the fully-dynamic connectivity data structure [30] and the Sleator/Tarjan's dynamic trees [55], it is possible to solve the matching problem faster than in $O(n \log^4 n)$ time.

The fully-dynamic graph connectivity data structure supports the following operations:

- edge insertion,
- edge deletion,
- answering a question whether two given vertices of the graph are connected by a path.

Each of the above operations takes $O(\log^2 n)$ amortized time. M. Thorup [59] introduced another dynamic data structure for solving the same problem which supports

Algorithm 4 *biedl_matching*(G, e)

```
Require: Biconnected cubic graph $G = (V, E)$.
Require: Excluded edge $e = v - w \in E$.
Ensure: Perfect matching of $G$ not containing $e$.
 1: if $|V| = 0$ then
 2:     return $\emptyset$
 3: else
 4:     $v' - w' = N_e(G, e)[0]$
 5:     $R = reductions(G, v' - w')$
 6:     if $bridgeless(G[V - \{v', w'\}, E \cup R[0]])$ then
 7:         $r = R[0]$
 8:     else
 9:         $r = R[1]$
10:     end if
11:     if $N_v(G, r[0]) \cap \{v - w\} = \emptyset$ then
12:         $e' \leftarrow r[1]$
13:     else
14:         $e' \leftarrow r[0]$
15:     end if
16:     $M \leftarrow biedl\_matching(G[V - \{v', w'\}, E \cup r], e')$
17:     $M \leftarrow (M' - r) \cup simple\_reversion(G, v, w, r, M)$
18:     return $M$
19: end if
```

the same set of operations in $O(\log n \log\log^3 n)$ expected time per one operation. Selecting one of these two data structures leads to a perfect matching algorithm for biconnected cubic graphs running in $O(n \log^2 n)$ time or $O(n \log n \log\log^3 n)$ expected time.

The second data structure used are dynamic trees of Sleator and Tarjan [55]. It maintains a dynamic forest by supporting edge deletions and insertions in $O(\log n)$ time per operation. It also supports computation of the nearest common ancestor of any two vertices of a rooted tree in $O(\log n)$ time.

Both dynamic connectivity data structures from [30] and [59] maintain internally a spanning forest T of graph G. A newly added edge $e = x - y$ to G is added to T if x and y were not connected in G prior to insertion of e. No other edge is removed or added to T. In case of removal of a spanning forest edge $e = x - y \in T$, the algorithm tries to find a replacement edge reconnecting components of x and y in the spanning forest. Any operation on dynamic connectivity data structure results in $O(1)$ edges inserted/removed from T. Our new matching algorithm needs to know the lowest common ancestor of some pairs of vertices in an arbitrarily rooted spanning forest of G. To answer these questions we maintain a rooted copy of the spanning forest T in the form of Tarjan's dynamic trees.

Our algorithm works in the following way:

- Initialize the dynamic connectivity data structure $\mathcal{D}$ by adding all edges of an input biconnected cubic graph G.
- Initialize the dynamic tree data structure $\mathcal{T}$ by adding all spanning tree edges of $\mathcal{D}$ to $\mathcal{T}$.
- Perform Biedl's algorithm but instead of using dynamic biconnectivity data structure for verifying which of the two reductions maintains biconnectivity, use the new approach described below.

We already know that at least one of the two possible reductions performed in each step of the Frink's algorithm leads to a biconnected graph. Searching for the correct reduction might be hard sometimes. However, if we find out that one reduction does not preserve biconnectivity, then the Frink's theorem implies that the other reduction is the one of our interest.

Let $G = (V, E)$ be a biconnected cubic graph. We want to perform a reduction against an edge $c - d$ (see the Figure 4.1). We first need to remove vertices c and d and edges $a - c$, $b - c$, $c - d$, $d - e$ and $d - f$ obtaining graph G'.

The data structure $\mathcal{D}$ needs to be updated accordingly. Two different scenarios are now possible — graph G' remains connected or not (this can be verified by querying $\mathcal{D}$ if a is connected with b, e and f).

4.3.1 Case 1: G' is not connected

Since graph G is biconnected, each of the removed edges ($a - c$, $b - c$, $c - d$, $d - e$ and $d - f$) lies on some cycle in G. As G has a cycle containing $c - d$, there has to be at least one of the following paths in G':

- between a and e,
- between a and f,
- between b and e,

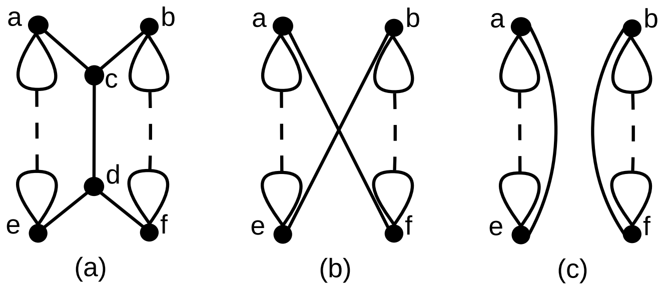

Figure 4.7: **(a)** The structure of G prior to edge deletions (every edge to be removed lies on some cycle). **(b)** Reduction leading to a biconnected graph. **(c)** Reduction leading to not connected graph.

- between b and f.

Assume (without loss of generality) that G' contains a path P connecting vertices a and e (see Figure 4.7). A cycle in G consisting of path P and edges $a-c$, $c-d$ and $d-e$ does not contain edges $b-c$ and $d-f$, hence there has to be another cycle (or cycles) in G containing those edges. Since G' is not connected, there has to be a path between b and f in G'. In order to maintain biconnectivity of the graph it is required to connect vertex a with f and vertex b with e (the second type of connection leads to not connected graph and it follows from the Frink's Theorem that the first connection maintains biconnectivity).

4.3.2 Case 2: G' is connected

As G' is connected, it is possible to use Tarjan's trees to compute the lowest common ancestor of any pair of vertices of G' in a rooted spanning tree T being maintained. Consider the subtree T' of the spanning tree T consisting of all edges spanned by paths connecting vertices a, b, e and f in T. We need to select one of the two possible connections of vertices a, b, e and f using new edges u and v such that every edge of T' lies on some cycle in $T' \cup \{u, v\}$. As we prove later, such a connection guarantees biconnectivity of the entire reduced graph.

If we connect vertex a with e and vertex b with f, edges spanned by paths connecting a with e and b with f are covered by cycles of $T' \cup \{u, v\}$. The only edges in question are located between the lowest common ancestor of a and e ($LCA(a, e)$) and the lowest common ancestor of b and f ($LCA(b, f)$). There are three possible cases depending on the relative position of $LCA(a, e)$ and $LCA(b, f)$ in T':

- $LCA(a, e) = g$, $LCA(b, f) = h$, $LCA(g, h) = i$, $i \neq g, h$ (see Figure 4.8(a),(b)) — after connecting a with f and b with e ($LCA(a, f) = LCA(b, e) = i$), every edge of T' lies on some cycle in $T' \cup \{u, v\}$.

- $LCA(a, e) = i$, $LCA(b, f) = i$ (see Figure 4.8(c),(d)) — after connecting a with e and b with f, every edge of T' lies on some cycle in $T' \cup \{u, v\}$.

- $LCA(a, e) = g$, $LCA(b, f) = i$, $LCA(g, i) = i$ (see Figure 4.8(e),(f); $LCA(g, i) = g$ is the symmetric case) — it is required to add such edges that generate cycles containing all edges between vertices g and i in T'. Edge w connecting vertex g with its parent in T' has to be a part of the cycles as well. If a cycle obtained by connecting b and f contains w (which can be tested by checking if $LCA(b, g) = g$ or $LCA(f, g) = g$) then connection of a with e and b with f is the correct reduction. Otherwise, we have a situation presented in the Figure 4.8(e) — edges between g and h are not included in any cycle. However, by selecting the second reduction — connection of a with f and b with e — all edges are included in both cycles.

The only remaining thing to prove is the fact that if every edge of T' is included in some cycle of $T' \cup \{u, v\}$, the graph resulting from the reduction is biconnected. In order to prove this fact it is sufficient to show that every edge of the reduced graph

G' lies on some cycle (according to the Theorem 2.8 a bridgeless cubic graph is also biconnected).

It has been already shown that it is possible to perform reduction in such a way that every edge e of $T' \cup \{u, v\}$ lies on some cycle. The rest of the edges remain. Let us consider an arbitrary edge of graph G' which is not part of the tree T'. As G is biconnected, there is a cycle C in G containing e. Assume (without loss of generality) that when performing the reduction we have connected a with e and b with f. A few cases have to be considered:

- Cycle C does not contain any of the removed edges — C is also a cycle in the reduced graph, so e belongs to a cycle in the reduced graph.
- Cycle C contains three removed edges $a - c$, $c - d$ and $d - e$ ($b - d$, $c - d$ and $d - f$) — replacing those edges in C with $a - e$ ($b - f$) leads to a cycle in the reduced graph containing e.

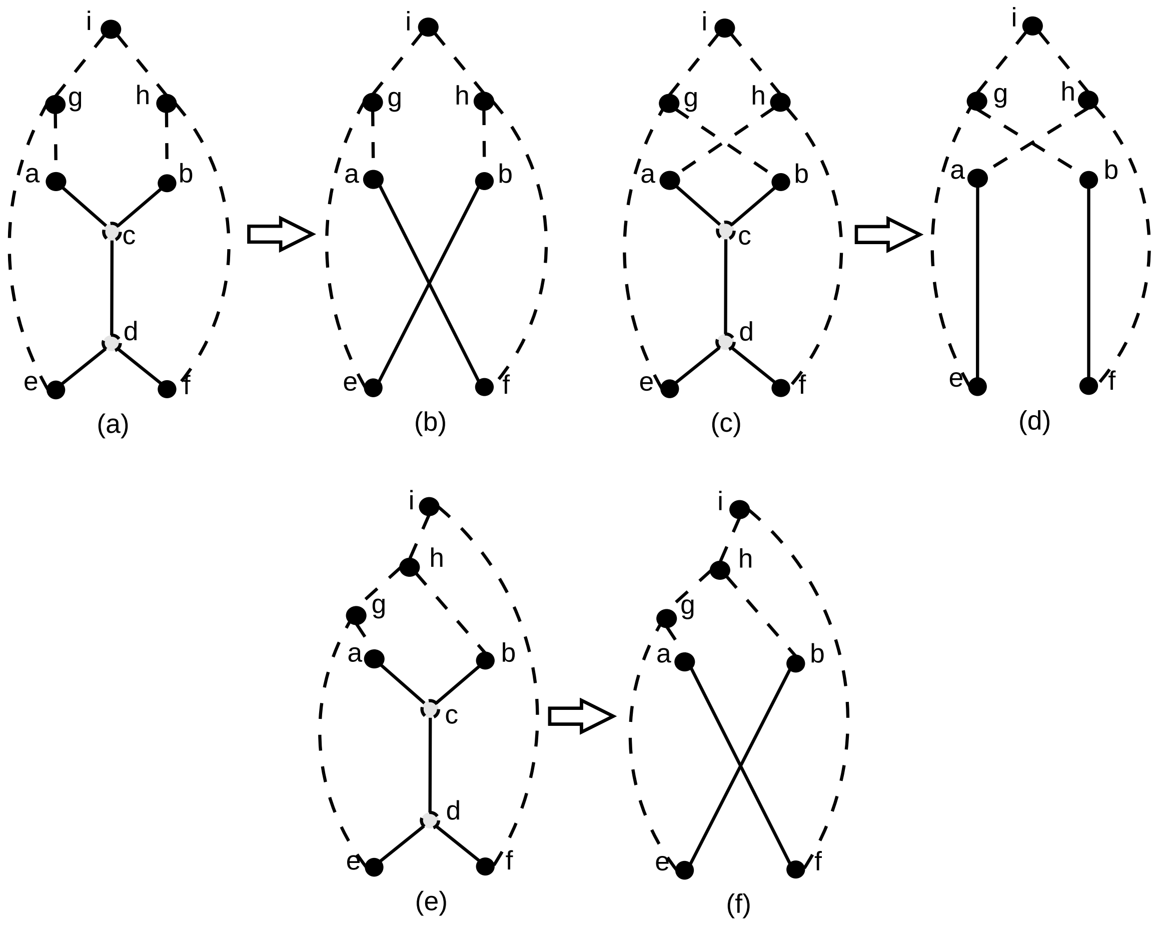

Figure 4.8: **(a)** Example graph G with $LCA(a, e) = g$, $LCA(b, f) = h$ and $LCA(g, h) = i$. **(b)** Graph G' obtained from (a) after reduction. **(c)** Example graph G with $LCA(a, e) = i = LCA(b, f)$. **(d)** Graph G' obtained from (c) after reduction. **(e)** Example graph G with $LCA(a, e) = g$, $LCA(b, f) = i$. **(f)** Graph G' obtained from (e) after reduction.

- C contains two removed edges $a-c$ and $b-c$ ($d-e$ and $d-f$) — T' is a tree, so there exists a path D connecting a and b in T' (e and f). $C \otimes (D \cup \{a-c, b-c\})$ is a collection of cycles containing e.
- C contains three removed edges $a-c$, $c-d$ and $d-f$ ($b-c$, $c-d$ and $d-e$) — T' is a tree, so there exists a path D connecting a and f in T' (b and e). $C \otimes D$ is a collection of cycles containing e.
- C contains four removed edges $a-c$, $b-c$, $e-d$ and $d-f$ — by replacing those edges with $a-e$ and $b-f$ we obtain a cycle (possibly two cycles) containing e.

This completes the proof that the reduced graph remains biconnected.

The proposed algorithm requires $O(n)$ operations of both dynamic connectivity and dynamic tree data structures, so its total execution time is $O(n \log^2 n)$ in case of applying dynamic connectivity algorithm from [30] and $O(n \log n \log\log^3 n)$ expected time in case of the data structure from [59].

4.4 Utilizing decremental data structure

In this section we sketch a modification of our algorithm from the previous section which makes it possible to replace dynamic connectivity data structure with its decremental equivalent. If the Frink's reductions didn't insert new edges to the graph we could have directly applied a decremental dynamic connectivity data structure. In case of many problems decremental data structures are faster than fully dynamic versions. M. Thorup proposed a decremental data structure for graph connectivity problem [58]. It starts with n-vertex, m-edge graph and maintains a spanning forest of the graph allowing to perform m edge deletion operations in $O(min\{n^2, m\, n \log n\} + \sqrt{n\, m}\, \log^{2.5} n)$ expected time. In case of graphs with $\Omega(\frac{n^2}{\log n})$ edges the expected execution time of a single operation is $O(\log n)$. For graphs with $\Omega(n^2)$ edges the complexity reduces to $O(1)$. Thorup's data structure does not help in our case, as cubic graphs are very sparse. There is a chance, however, that one designs an efficient decremental algorithm for biconnected cubic graphs in the future.

In order to utilize a decremental dynamic connectivity data structure, apart from the original graph $G = (V, E)$ we maintain a modified representation of G denoted by G'. Each vertex v of G is represented in G' as two vertices v_1 and v_2 connected by an edge $v_1 - v_2$. Each edge $v - w$ of G is represented as four edges $v_1 - w_1$, $v_1 - w_2$, $v_2 - w_1$, $v_2 - w_2$. The Figure 4.9(a) shows the representation of two vertices connected by a single edge. Connectivity of graph G can be verified by querying the decremental dynamic connectivity data structure maintained for G'. As it is not possible to add edges to G' once Frink's reductions of graph G are performed, new edges of G are stored in G' utilizing representations of already removed edges of G. The figure 4.10 presents changes in the graph G' introduced by a single Frink's reduction in graph G. Five edges are removed and two edges are added in G. Every added edge is represented in G' as a subset of already removed edges. Such an added edge is called *closed* as it is not possible to perform further Frink's reductions against it without

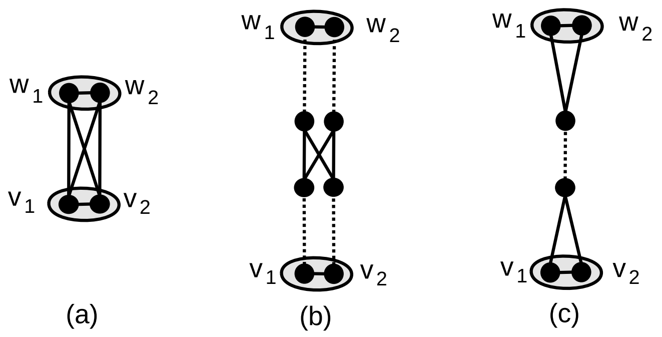

Figure 4.9: **(a)** Initial representation in G' graph of two vertices v and w connected by an edge. **(b)** General representation of an *open* edge — v_1 is connected by a path with w_1, v_2 with w_2. In addition the two paths are joined by a „crossing". **(c)** General representation of a *closed* edge.

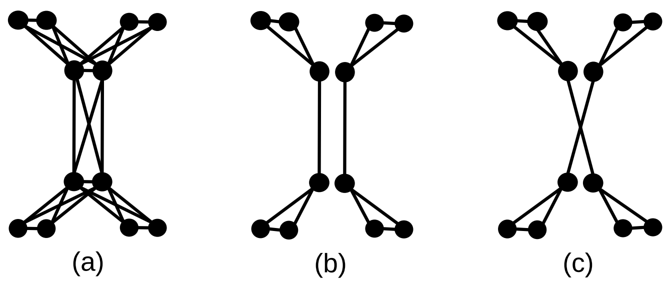

Figure 4.10: **(a)** State of graph G' for the corresponding graph G presented in the Figure 4.1(a). **(b)** State of graph G' after performing the first type of the Frink's reduction — newly added edges are represented using parts of removed edges. **(d)** State of graph G' after the second type of the Frink's reduction.

radical changes to the graph G'. A general representation of a *closed* edge is given in the Figure 4.9(c).

Just like in case of the matching algorithm from the previous section, we start by selecting an *excluded edge e*. A sequence of the Frinks' reductions is then performed. Apart from the way of querying and updating the dynamic connectivity data structure there are yet no differences in comparison with the algorithm utilizing fully-dynamic connectivity data structure. The only problem is encountered when it turns out that reduction to be performed affects a *closed* edge. In such a situation it is not possible to update G' to reflect changes in G. In order to go around this problem we extend the decremental dynamic data structure with restoration points. A restoration point allows for restoring the state of the data structure to the moment when the restoration point was created. Restoration points can be implemented by recording all modifications made to the data structure and reverting them upon execution of a restoration point. There is no additional performance cost (by means of asymptotic complexity) to record changes made to the data structure. The time required to execute a restoration point is amortized by the process of performing operations on the data structure. Before performing a single Frink's reduction a restoration point needs to be created. This way it is possible to restore the state of the data structure to the state before any Frink's reduction.

Two cases are possible when it turns out that the next reduction is to be performed against a *closed edge* f — f can equal e, i.e. f is the edge from which the reduction process started (see the Figures 4.11, 4.12), or not (see the Figure 4.13).

If f turns out to be e it means a cycle C was encountered. By executing a restoration point which was created prior to selecting an *excluded edge* e it is possible to represent all new edges of G in G' in the open form using the subset of removed edges. Figures 4.11 and 4.12 present the way of handling this situation depending on the parity of the length of C.

If f is not e it means that f must have been introduced to G' by one of the Frink's reductions, for which restoration point has not yet been executed. Denote by r_1, r_2, $\ldots$, r_k consecutive restoration points that have not yet been executed. Let r_m be the first restoration point for which G' contains a closed edge f. The next Frink's reduction to be performed affects edge f, so f was encountered twice by a sequence of reductions (see Figure 4.13). Reversion point r_m splits representation of the excluded edge e in G' into two parts — the part constructed prior to creation of r_m and the part generated afterwards. By executing the restoration point r_m it is possible to use representation of edges of G removed by the Frink's reductions after creation of r_m to turn all *closed* edges added to the graph after creation of r_m into an *open* form. Edge f is one of those edges, so it also becomes open. Now it is possible to perform the remaining Frink's reduction. Once the graph G is empty the rest of the algorithm is exactly the same as before.

The only difference in the complexity analyses compared to the algorithm from the previous section is the need of maintaining graph G' and executing restoration points. The time required to execute restoration points and reconstruct open edges is amortized by the cost of performing the Frink's reductions. Hence, the total execution time of the algorithm is $O(n \log n + nf(n))$ where $f(n)$ is the cost of a single operation on the decremental dynamic connectivity data structure. The $O(n \log n)$ component of the algorithm's complexity comes from the execution of Sleator/Tarjan's dynamic trees. If one designed a decremental dynamic connectivity data structure for sparse graphs running in $O(\log n)$ time per operation it would be possible to find perfect matching in biconnected cubic graphs in $O(n \log n)$ time.

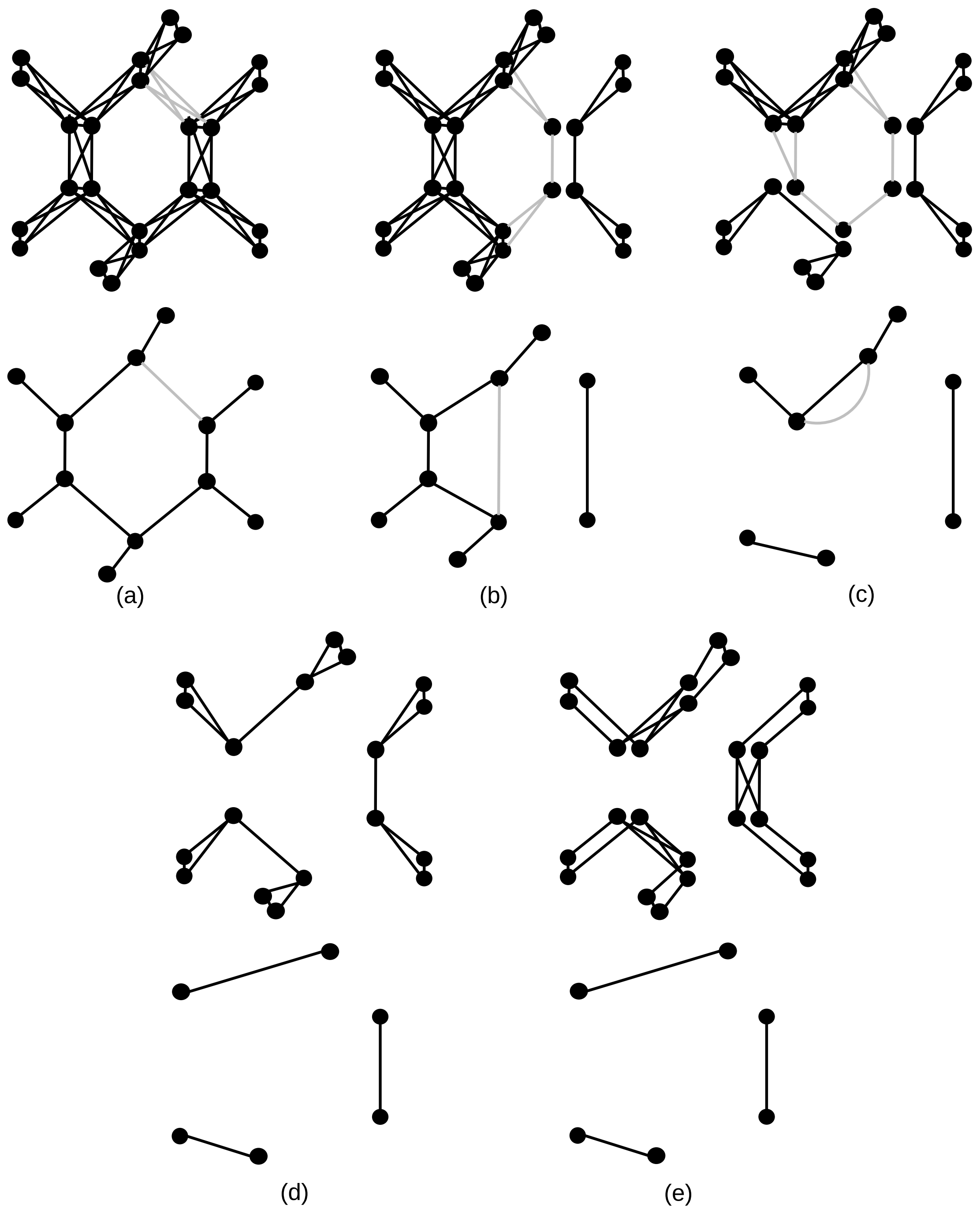

Figure 4.11: The process of performing reductions along an even cycle. The lower row presents changes in the graph G, while the upper row contains corresponding graph G'. Light edges represent edge e which is excluded from the matching. **(a)** Initial graph G and its corresponding graph G'. **(b)** G and G' after the first reduction along the cycle. **(c)** G and G' after the second reduction along the cycle. This reduction introduces a double edge which has to be reduced in the way depicted in the Figure 4.2(a). **(d)** State of G and G' after reduction of the double edge. **(e)** State of the graphs after executing a restoration point and converting all *closed* edges of G' into open edges.

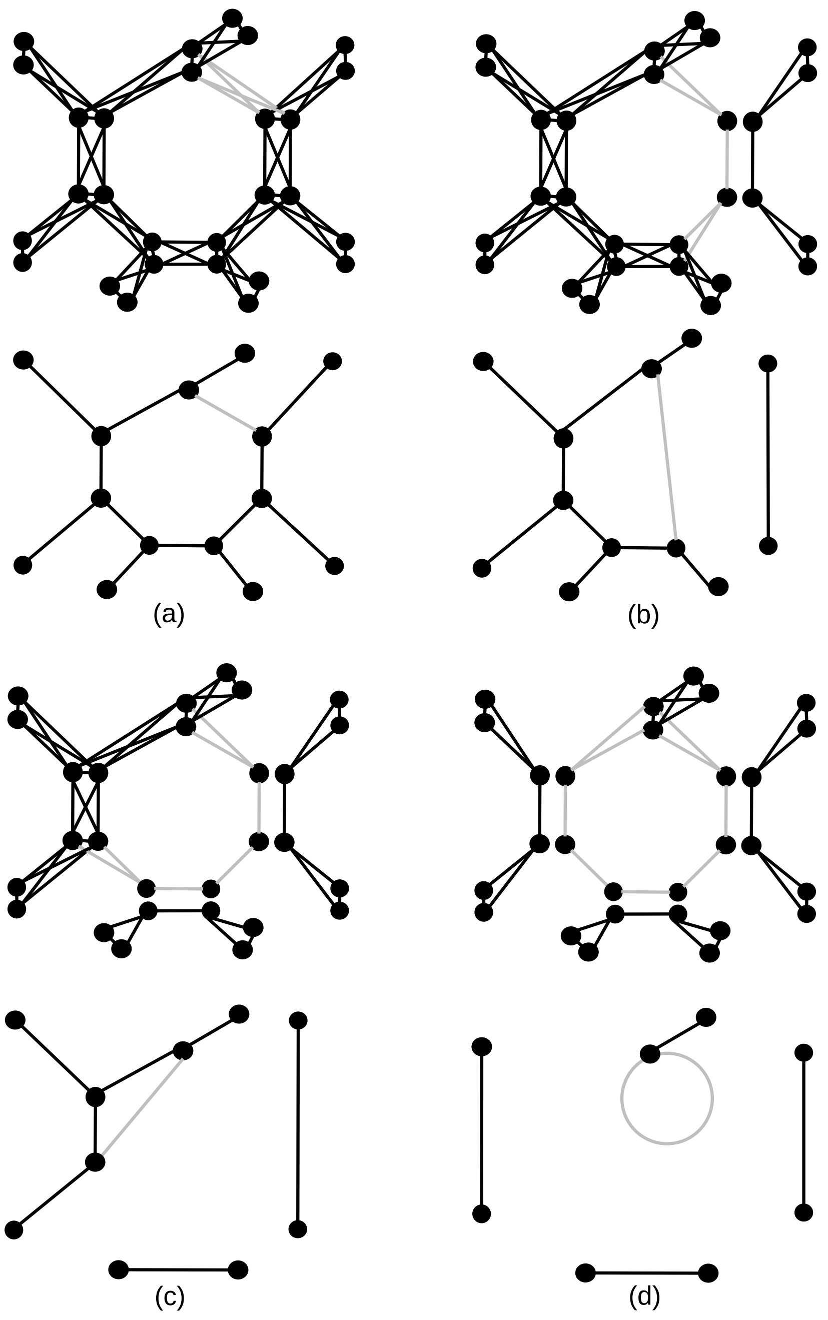

Figure 4.12: The process of performing reductions along an odd cycle. The lower row presents changes in graph G while the upper row contains corresponding graph G'. Light edges represent edge e which is excluded from the matching. **(a)** Initial graph G and its corresponding graph G'. **(b)** G and G' after the first reduction along the cycle. **(c)** G and G' after the second reduction along the cycle. **(d)** The last reduction along the cycle leads to not biconnected graph (a vertex incident to edge e is connected with the rest of the graph by a bridge), so this reduction is never performed by the algorithm.

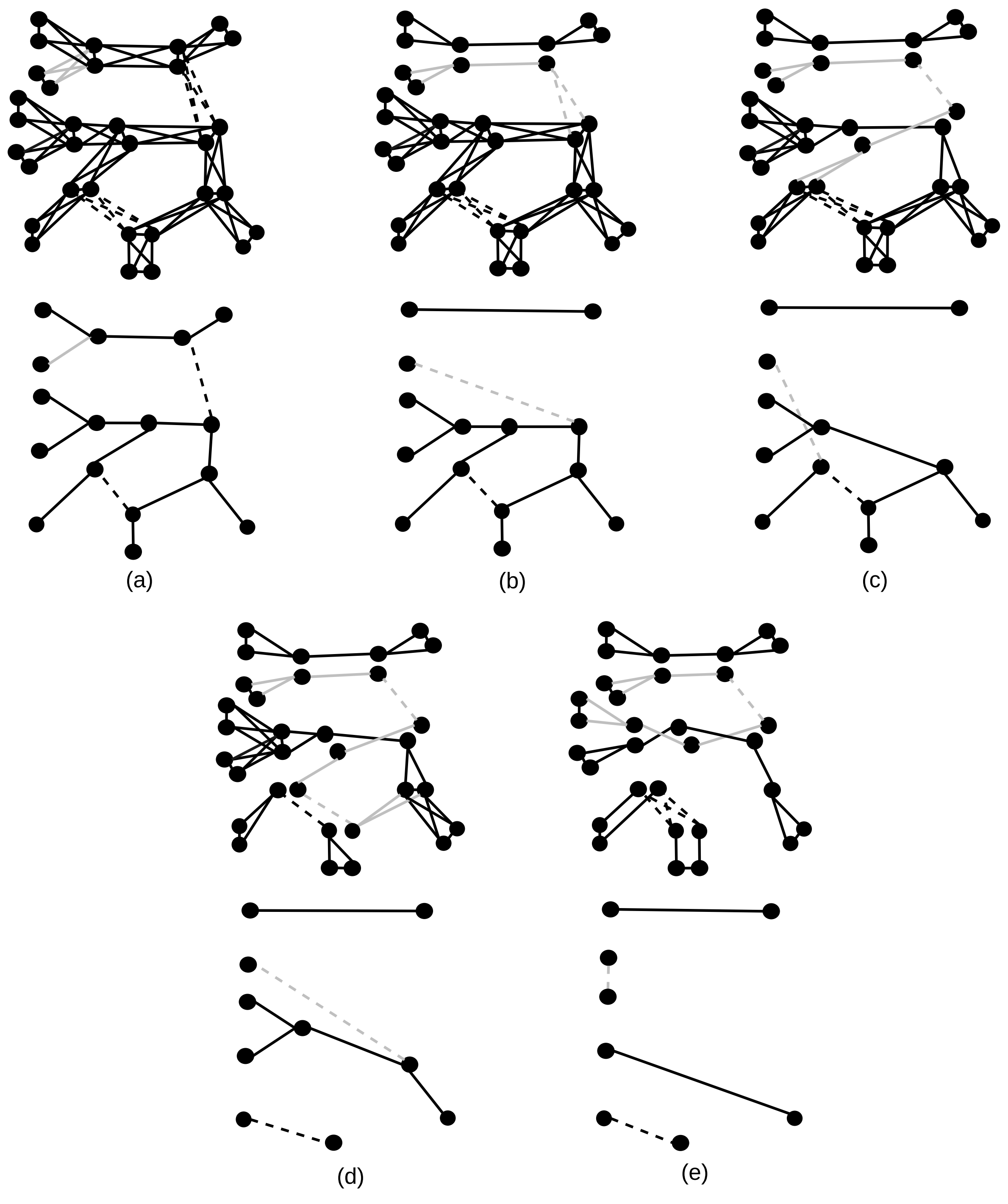

Figure 4.13: The process of reductions in case of encountering a *closed* edge f, different from e. The lower row presents changes in graph G while the upper row contains the state of corresponding graph G'. Light edges represent an excluded edge e. **(a)** Initial graph G and its corresponding graph G'. **(b)** The first sequence of reductions. **(c)** Reduction r_m which introduces a *closed* edge f. **(d)** The second sequence of reductions. The next reduction to be performed affects a closed edge f. **(e)** In order to make the next reduction, it is required to execute a restoration point created before stage (c). This way, all *closed* edges introduced into G after stage (b) can be turned into an *open* form (including f) and so the following reduction against f is possible.

Chapter 5

Parallel matching algorithms

In this chapter we introduce new parallel algorithms for constructing maximum matching for some broad classes of graphs. We start by presenting two NC algorithms for subclasses of cubic graphs. These algorithms are designed respectively for bipartite cubic and planar bipartite cubic graphs and their execution times are $O(\log^2 n)$ and $O(\log n \log^* n)$ respectively. We conclude this chapter with a parallel version of M. Mucha and P. Sankowski algebraic matching algorithm [43].

5.1 Bipartite cubic graphs

In this section we present a new algorithm for computing perfect matchings in bipartite cubic graphs. The general approach is similar to the Frink's sequential algorithm for biconnected cubic graphs [23]. Our algorithm runs in $O(\log^2 n)$ time and utilizes $O(n/\log n)$ processors. It's total work — $O(n \log n)$ — is near to optimal $O(n)$ work of sequential algorithm by A. Schrijver [52] and utilizes less processors than the fastest parallel algorithm (by a $O(\log^*(n))$ factor) for solving this problem known to date [54]. It is also much simpler to understand and implement.

Keys to our parallel algorithm are Lemma 2.9 and the Petersen's Theorem 4.1. Lemma 2.9 states that every bipartite cubic graph is biconnected and the Petersen's Theorem 4.1 implies that every such a graph has a perfect matching. When applying the Frink's reduction to a bipartite cubic graph, there is no need of verifying whether the resulting graph remains biconnected. Every Frink's reduction maintains bipartiness of a graph, hence also biconnectivity (according to Lemma 2.9). This observation allows us to provide a new sequential algorithm for bipartite cubic graphs that is based on the Biedl's algorithm. By disabling graph's biconnectivity validation, we obtain an optimal, linear time algorithm. The advantage of this approach over A. Schrijver's [51] algorithm is a possibility to relatively easily turn it into a parallel algorithm. There is also a disadvantage, however, it is not possible to adopt such an algorithm to regular graphs of degrees larger than 3.

Definition 5.1. *Let $G = (V, E)$ be a graph. A set of edges $E' \subseteq E$ is called* non-conflicting *if for any pair of distinct edges $e_1, e_2 \in E'$, they are not adjacent and they do not have a common adjacent edge ($(e_1 \cup N_e(G, e_1)) \cap (e_2 \cup N_e(G, e_2)) = \emptyset$).*

Example 5.2. An example set of *non-conflicting edges* is presented in the Figure 5.1.

Instead of executing a single reduction at a time, our new algorithm computes a set of *non-conflicting edges* and performs reductions against all of them in a single step. Once an empty graph is obtained, it is possible to select an initial empty matching and revert all reduction stages, executing each reversion stage in parallel. Three simple cases of reverting a reduction step from the Figure 5.1 are presented in the Figure 5.2. These simple reversion cases can be performed in constant time by executing each reversion independently from each other just like in the original Frink's algorithm.

The problem arises when it turns out that both edges introduced by a single reduction are matched (equivalent of the case (d) from the Figure 4.5). Before reverting such a case, we need to modify a perfect matching so that at least one of the edges is no more in the perfect matching. Only then we can apply a simple $O(1)$ reversion procedure.

Lemma 5.3. *Let $G(V, E)$ be a bipartite cubic graph and M a perfect matching of G. G contains a perfect matching M', which is disjoint with M.*

Proof. Consider $G[V, E - M]$. Every vertex of this graph has degree 2, hence it is a collection of cycles. All cycles are of even length, as G is bipartite. By selecting from every cycle every second edge we obtain a perfect matching M' disjoint with M. □

The proof of Lemma 5.3 gives us a general idea how to construct a matching M' disjoint with M. We can design a single reversion step in the following way:

- Revert all reductions where at most one of the edges being reverted is matched (according to the rules (a), (b) and (c) from the Figure 4.5).
- Construct a perfect matching M' disjoint with the current perfect matching M.

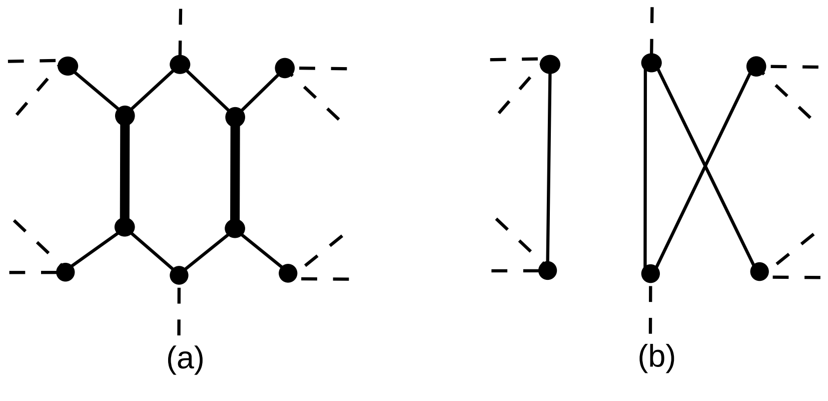

Figure 5.1: **(a)** An example cubic graph and a set of two *non-conflicting* edges (presented in bold). **(b)** The example graph after Frink's reductions performed against all *non-conflicting edges*.

- Revert all remaining reductions (this time none of the edges being reverted is matched, so rule (a) from the Figure 4.5 applies).

The exemplary reversion process requiring all three steps is presented in the Figure 5.3. The general structure of our new parallel algorithm is presented as an Algorithm 5.

The final complexity of the algorithm depends on the implementation of the process of finding *non-conflicting edges* and the performance of constructing the disjoint matching M'. The following two subsections present implementation of these two building blocks.

Non-conflicting edges **selection**

Depending on the number of *non-conflicting edges* that can be found at each step of the algorithm, the number of recurrence calls of *nc_matching* function is different. Hence, apart from the performance of constructing a set of *non-conflicting edges*, the crucial role plays the size of the found set.

Lemma 5.4. *For a given bounded degree graph $G = (V, E)$ it is possible to construct an independent set of vertices of linear size in $O(\log^* n)$ time utilizing $O(n)$ processors.*

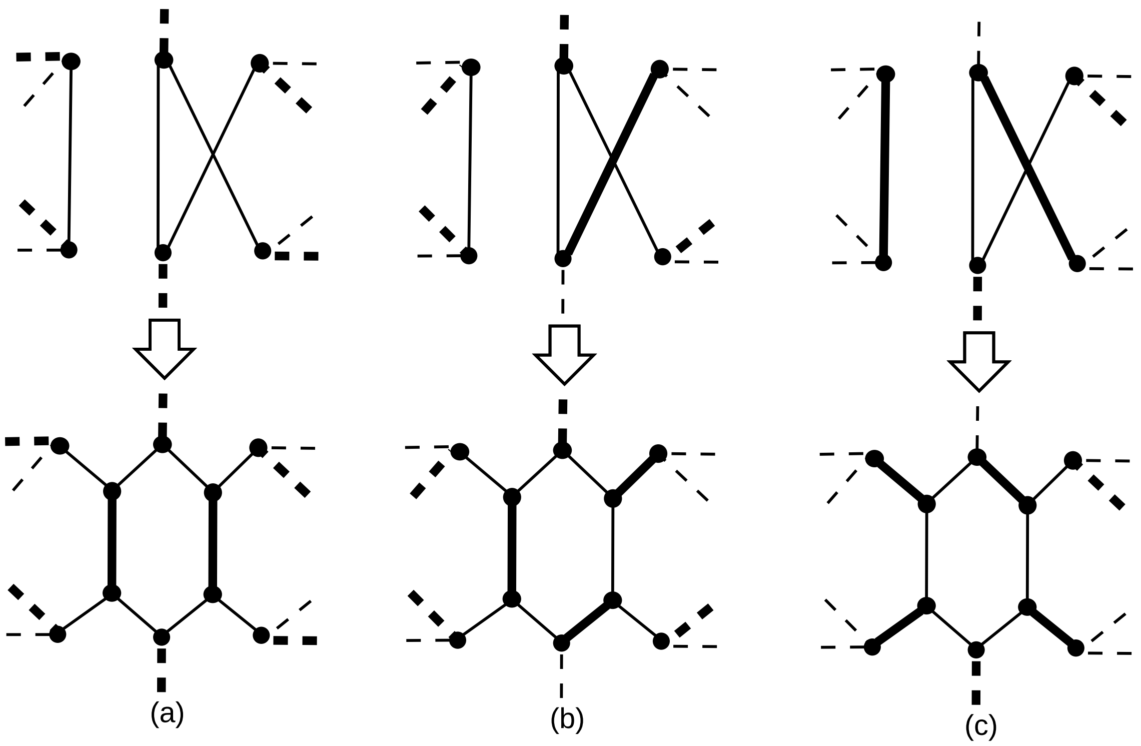

Figure 5.2: Reversion process of the simple case reductions of *non-conflicting edges* in parallel. Edges from the perfect matching are presented in bold. **(a)** None of the edges being reverted is matched. **(b)** One of the edges being reverted is matched. **(c)** Two of the edges being reverted are matched.

Proof. It is sufficient to find a maximal independent set. Since G is a graph of bounded degree, it can be done in $O(\log^* n)$ time using $O(n)$ processors [26]. □

Lemma 5.5. *For a given cubic graph $G = (V, E)$ it is possible to reduce the* non-conflicting edges *problem to the maximal independent set of vertices problem in $O(1)$ time utilizing $O(n)$ processors.*

For a given biconnected cubic graph G the pseudo-code presented as Algorithm 6 constructs a *non-conflicting edge* set of linear size in $O(\log^* n)$ time utilizing $O(n)$ processors. It calls *independent_set* function, which implements the parallel maximal independent set algorithm.

Disjoint matching construction

For a given bipartite cubic graph G with a perfect matching M it is relatively easy to construct another perfect matching M' disjoint with M. To do so, the following procedure can be applied:

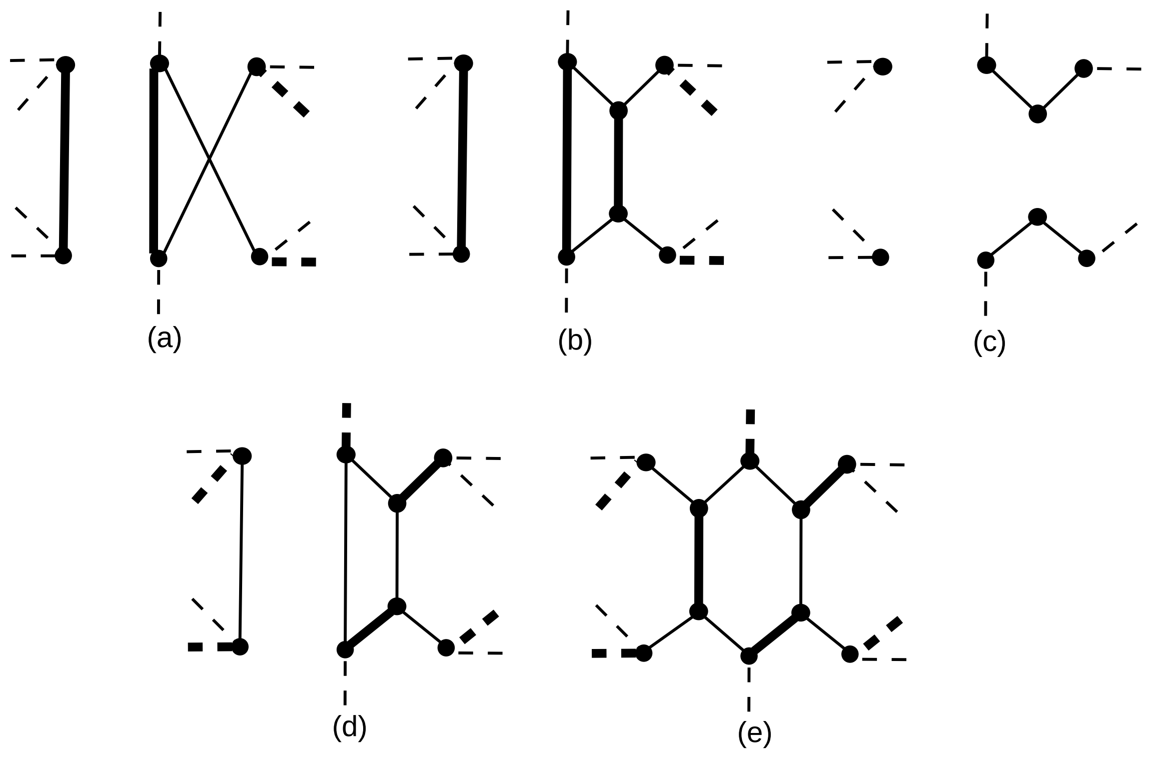

Figure 5.3: The process of reverting reductions when some reductions with both edges being reverted are in the perfect matching occur. **(a)** Initial state before reversion process, one reduction has two edges matched, the other none. **(b)** Reduction without matched edges is reverted, matching M is obtained. **(c)** Matching M is removed from G leading to a collection of even cycles. **(d)** Every second edge from all even cycles is selected constructing a matching M' ($M \cap M' = \emptyset$). **(e)** The remaining reduction is reverted.

Algorithm 5 *nc_matching*(G)

Require: Cubic bipartite graph $G = (V, E)$.
Ensure: Perfect matching of G.

```
1:  if $|V| = 0$ then
2:      return $\emptyset$
3:  else
4:      $E' \leftarrow$non-conflicting_edges$(G)$
5:      $G' \leftarrow G$
6:      for all $v - w$ in $E'$ do
7:          $G' \leftarrow G'[V - \{v - w\}] \cup reductions(G, v, w)[0]$
8:      end for
9:      $M \leftarrow nc\_matching(G')$
10:     for all $v - w$ in $E'$ do
11:         $R \leftarrow reductions(G, v, w)[0]$
12:         if $|R \cap M| \neq 2$ then
13:             $M \leftarrow (M - R) \cup simple\_reversion(G, v, w, R[0], M)$
14:         end if
15:     end for
16:     $M' \leftarrow disjoint\_matching(G, M)$
17:     for all $v - w$ in $E'$ do
18:         $R \leftarrow reductions(R, v, w)[0]$
19:         if $|R \cap M| = 2$ then
20:             $M' \leftarrow M' \cup simple\_reversion(G, v, w, R[0], M')$
21:         end if
22:     end for
23:     return $M'$
24: end if
```

- Remove all edges of M from G obtaining a collection of even cycles G'.
- Orient arbitrarily all cycles of G'.
- As G is bipartite, its vertices are contained in two sets V' and V'', $(V' \cup V'' = V)$ such that every edge joins vertex of V' with vertex of V''. Matching M' can be obtained by selecting all edges of G' outgoing from the vertices of V'.

It is obvious how to perform the first and the third step of the algorithm in constant time using $O(n)$ processors. The only nontrivial part is the parallel orientation of the cycles. It can be completed in $O(\log n)$ time with $O(n)$ processors by applying a doubling technique:

- For each cycle determine a representing vertex (for example a vertex with the highest ID).
- For each representing vertex remove an arbitrary edge incident with it. This turns all cycles of G' into paths starting in the representing vertices.

Algorithm 6 $non-conflicting\_edges(G)$

Require: Bounded degree graph $G = (V, E)$.
Ensure: Set of *non-conflicting edges* of G.

```
$G'(V', E') \leftarrow (\emptyset, \emptyset)$
for all $v - w$ in $E$ do
    $V' \leftarrow V' \cup \{v - w\}$
end for
for all $v - w$ in $V'$ do
    for $x \in N_v(G, v) - \{w\}$ do
        $E' \leftarrow E' \cup \{v - w, v - x\}$
        for $y \in N_v(G, x) - \{v\}$ do
            $E' \leftarrow E' \cup \{v - w, x - y\}$
        end for
    end for
    for $x \in N_v(G, w) - \{v\}$ do
        $E' \leftarrow E' \cup \{v - w, w - x\}$
        for $y \in N_v(G, x) - \{w\}$ do
            $E' \leftarrow E' \cup \{v - w, x - y\}$
        end for
    end for
end for
$set \leftarrow independent\_set(G')$
$result \leftarrow \emptyset$
for all $v - w$ in $set$ do
    $result \leftarrow result \cup \{v - w\}$
end for
return $res$
```

- By applying a doubling technique compute for each vertex of each cycle a distance *dist* from the representing vertex.

- Based on a computed distance function *dist* orient all edges of all cycles away from the representing vertices (for a given edge $v - w$ with $dist[v] = dist[w] - 1$ orient it from v to w).

- For every representing vertex v introduce the previously removed edge and orient it to v.

Orientation of cycles is obtained in $O(\log n)$ time with $O(n)$ processors, hence the disjoint matching construction phase has the same time complexity and processors requirement.

Algorithm 7 *disjoint_matching*(G, M)

Require: Bipartite cubic graph $G = (V, E)$.
Require: Perfect matching M of G.
Ensure: Perfect matching M' of G disjoint with M.

```
1: $G \leftarrow G[V, E - M]$
2: $D \leftarrow compute\_distance(G)$
3: $M' \leftarrow \emptyset$
4: for all $v \in V$ do
5:     if $D[v]$ is even then
6:         for all $u \in D_v(G, v)$ do
7:             if $D[v] + 1 = D[u]$ then
8:                 $M' \leftarrow M' \cup \{u - v\}$
9:             end if
10:        end for
11:    end if
12: end for
13: return $M'$
```

Algorithm 8 *compute_distance*(G)

Require: Bipartite regular graph $G = (V, E)$ of degree 2.
Ensure: Consecutive numbering of vertices of all cycles of a graph G.

```
1: $R \leftarrow representative(G)$
2: for all $v \in V$ do
3:     if $v = R[v]$ then
4:         $E \leftarrow E - \{D_e(G, v)[0]\}$
5:     end if
6: end for
7: $D : V \to \mathbb{Z} \leftarrow (\emptyset \to \emptyset)$
8: for all $v \in V$ do
9:     if $v = R[v]$ then
10:        $D[v] \leftarrow 0$
11:    else
12:        $D[v] \leftarrow \infty$
13:    end if
14: end for
15: $D \leftarrow distance(G, D)$
16: return $D$
```

5.1.1 Complexity analysis

In this subsection we analyze the time complexity and total work of the *nc_matching* algorithm. We prove that the presented implementation executes in $O(\log^2 n)$ time with $O(n)$ processors. Latter we show that it is possible to reduce the required number

Algorithm 9 *distance*(G, D)

Require: A graph $G = (V, E)$ which is a set of paths.
Require: A distance function $D : V \to \mathbb{Z}$ initialized to 0 for each head of a path, to ∞ otherwise.
Ensure: A distance function D representing the distance of each vertex from its path's head.

```
 1: $step \leftarrow 1$
 2: while $step < |V|$ do
 3:    for all $v \in V$ do
 4:       for all $u \in D_v(G, v)$ do
 5:          $D[v] \leftarrow min(D[v], D[u] + step)$
 6:       end for
 7:    end for
 8:    $M' = \emptyset$
 9:    for all $v \in V$ do
10:       if $|D_v(G, v)| = 2$ then
11:          $M' \leftarrow M' \cup \{D_v(G, v)[0] - D_v(G, v)[1]\}$
12:       end if
13:    end for
14:    $M \leftarrow M'$
15:    $step \leftarrow step * 2$
16: end while
17: return $D$
```

of processors to $O(n/\log n)$ maintaining the same time complexity.

Lemma 5.6. *For a given bipartite cubic graph G, Algorithm 5 computes a perfect matching in $O(\log^2 n)$ time using $O(n)$ processors.*

Proof. As in each recurrence execution of *nc_matching* function, the computed set of *non-conflicting edges* is of linear size, there exists a constant $0 < g < 1$, such that the size of the graph under consideration reduces each time at least by a factor g. Algorithm executes k recurrence levels ($k \leq \log_{1/g} n$). In the k-th recursion level the graph contains not more than ng^k vertices, hence *nc_matching* function requires $O(\log(ng^k))$ time to process k-th level (computation of *non-conflicting edges* takes $O(\log^*(ng^k))$, construction of the disjoint matching M' takes $O(\log(ng^k))$ time, other steps require $O(1)$ time). By summing the time required by all levels we obtain:

$$O(\log n + \log ng + \log ng^2 + \cdots + \log ng^k) \leq O(k \log n) \leq O(\log_{1/g} n \log n) = O(\log^2 n)$$

□

Only the first level of recursion of the *nc_matching* function uses all n processors. Once graph is being reduced the number of utilized processors decreases. If we execute *nc_matching* algorithm on $O(n/\log n)$ processors, we can reduce the total work of the

Algorithm 10 *representative*(G)

Require: Bipartite regular graph $G = (V, E)$ of degree 2.
Ensure: A mapping $R : V \to V$ from vertices of a graph G to cycles' representatives.

```
$R : V \to V \leftarrow (\emptyset \to \emptyset)$
for all $v \in V$ do
    $R[v] \leftarrow v$
end for
$step \leftarrow 1$
while $step < |V|$ do
    for all $v \in V$ do
        for all $u \in D_v(G, v)$ do
            $R[v] \leftarrow min(R[v], R[u])$
        end for
    end for
    $M' = \emptyset$
    for all $v \in V$ do
        $M' \leftarrow M' \cup \{D_v(G, v)[0] - D_v(G, v)[1]\}$
    end for
    $M \leftarrow M'$
    $step \leftarrow step * 2$
end while
return $R$
```

algorithm by better utilization of processors. As Lemma 5.7 states, such a reduction of the number of processors can be obtained without sacrificing the overall performance of the algorithm.

Lemma 5.7. *For a given bipartite cubic graph G with n vertices, Algorithm 5 can be executed with $O(n/\log n)$ processors in $O(\log^2 n)$ time.*

Proof. As the number of processors available is less than the number of vertices in the input graph G, it is required for a number of initial recurrence levels to share a single processor by a number of vertices. The very first recurrence call of the *nc_matching* function needs to analyze $O(\log n)$ vertices per processor. The following calls process $O(\log ng)$, $O(\log ng^2)$, $\ldots$ vertices per processor. This increases the time required to complete the k-th level from $O(\log ng^k)$ to $O(max(\log ng^k, g^k \log ng^k \log n))$. By summing up the times required by all recurrence levels we obtain:

$$\Sigma_{m=1}^{k} max(\log ng^k, g^k \log ng^k \log n) \le \Sigma_{m=1}^{k} \log ng^k + \Sigma_{m=1}^{k} g^k \log ng^k \log n \le$$
$$\le \log^2 n + \log n \Sigma_{m=1}^{k} g^k \log ng^k \le \log^2 n + \log^2 n \Sigma_{m=1}^{k} g^k \le$$
$$\le \log^2 n + \log^2 n \frac{1}{1-g} = O(\log^2 n)$$

Hence the reduction of the number of processors from $O(n)$ to $O(n/\log n)$ does not affect the overall asymptotic algorithm's performance. □

5.2 Planar bipartite cubic graphs

The main complexity factor of the algorithm from the previous section is the construction of a disjoint perfect matching. A bipartite cubic graph, after removal of the perfect matching M, becomes a graph consisting of a set of even length cycles G'. In the worst case it may turn out that it consists of a single cycle of length n. In such a case orientation of the cycle takes $\theta(\log n)$ time. If only we were able to limit the length of the cycles by a constant, it would give us a faster algorithm. We are not able to obtain such an upper bound in general case, but if we restrict ourselves to planar graphs, it turns out that it is possible to derive a faster solution.

Lemma 5.8. *Let G be a planar bipartite cubic graph and P be an arbitrary embedding of G in a plane. Then G has at least $\frac{n}{2}$ faces in P.*

Proof. The Euler's formula 2.4 states that for any connected planar graph $n-m+f=2$ holds, where n is the number of vertices, m — the number of edges, f — the number of faces. As G is cubic $m=\frac{3}{2}n$. Let k be the number of connected components of G and $n_1, n_2, \ldots, n_k$ be the numbers of vertices in connected components, respectively. The total number of faces of G, according to Euler's formula, is:

$$f+k-1=\sum_{l=1}^{l=k}\frac{3}{2}n_l-n_l+2=\frac{n}{2}+2k$$

This proves that G has at least $\frac{n}{2}$ faces. □

Lemma 5.9. *Let G be a planar bipartite cubic graph and let P be an arbitrary embedding of G in the plane. Then G contains a linear number of faces of size less than or equal* 6.

Proof. An n-vertex bipartite planar cubic graph has $\frac{3}{2}n$ edges. The total length of all faces of G is $3n$, as each edge of the graph contributes twice. As G has at least $\frac{n}{2}$ faces (Lemma 5.9) the average length of the face is not larger than $\frac{3n}{n/2}=6$. As there are linearly many faces in G, the number of faces of length shorter or equal 6 is also linear. □

What we are trying to accomplish is an algorithm which, instead of reducing single non-conflicting edges, reduces entire cycles at once. Consider an example presented in the Figure 5.4. It shows a way of eliminating a cycle from graph G resulting in a smaller bipartite planar cubic graph G'. Once a perfect matching M' is computed for the graph G', it is possible to revert the reduction in a constant time and compute a perfect matching M for G. Some examples of reversion process are presented in the Figure 5.5.

Definition 5.10. *Let $G=(V,E)$ be a bipartite planar cubic graph. The set of disjoint cycles C of G is called* non-conflicting *if for any pair of cycles $C_1, C_2 \in C$ they do not have an adjacent edge in common ($(C_1 \cup N_e(G,C_1)) \cap (C_2 \cup N_e(G,C_2))=\emptyset$).*

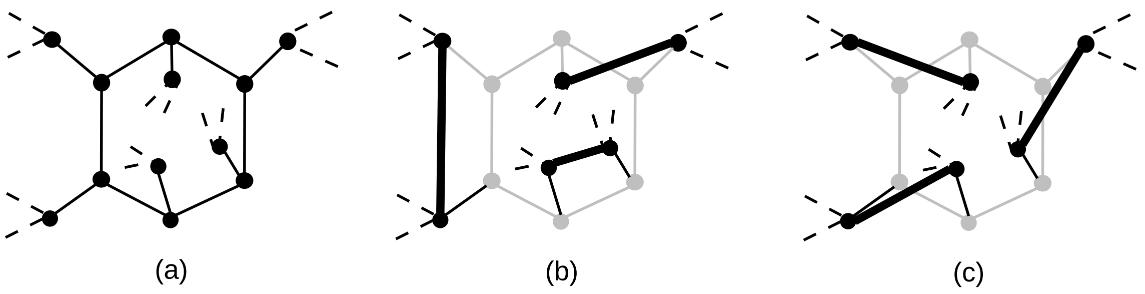

Figure 5.4: **(a)** An initial graph with a cycle of length 6 to be reduced. **(b)** The first possible reduction of the cycle. All removed vertices and edges are presented in gray. Edges added to the graph are bold. **(c)** The second possible reduction of the cycle. Please notice that reductions performed preserve planarity of the graph.

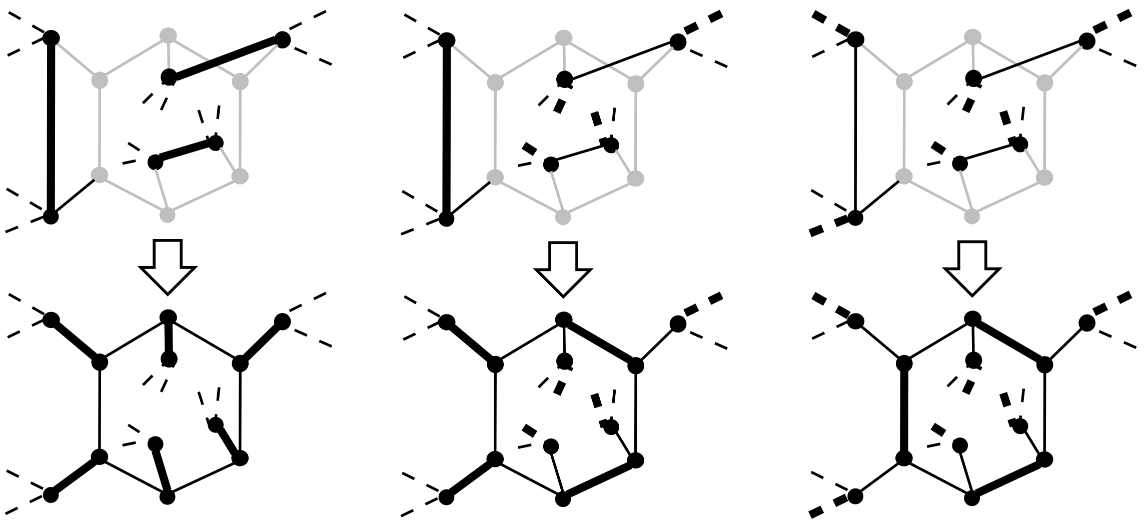

Figure 5.5: Reversion process of the reduction presented in the Figure 5.4(b). Three cases are presented: exemplary perfect matchings M' (top row) and a reverted reduction together with reconstructed perfect matching M (bottom row).

Just like in case of the algorithm from the previous section, which was reducing the set of *non-conflicting edges* at once, it is possible to perform a parallel reduction against all *non-conflicting cycles* at once. As we consider cycles of length up to 6, each cycle can be reduced by a single processor in a constant time. The only question is how to construct such a set of *non-conflicting cycles* efficiently. For this purpose we introduce the following approach:

- For each vertex v find a set of all cycles not longer than 6 containing v.
- Construct a graph G', in which vertices represent cycles found in the previous step (cycles found multiple times need to be added to the graph only once). Two vertices of G' are connected by an edge if cycles of G corresponding to that

Algorithm 11 *nc_planar_matching*(G)

Require: Planar bipartite cubic graph $G = (V, E)$.
Ensure: Perfect matching of G.

```
 1: $C \leftarrow independent\_cycles(G)$
 2: $G'(V', E') \leftarrow G(V, E)$
 3: for all $c \in C$ do
 4:    $G' \leftarrow G[C' - c, E' \cup cycle\_reduction(G, c)]$
 5: end for
 6: $M \leftarrow nc\_planar\_matching(G')$
 7: for all $c \in C$ do
 8:    $M \leftarrow (M - cycle\_reduction(G, c)) \cup cycle\_reversion(G, c, M)$
 9: end for
10: return $M$
```

Algorithm 12 *independent_cycles*(G)

Require: Planar bipartite cubic graph $G = (V, E)$.
Ensure: Set of independent cycles of G.

```
 1: $G'(V', E') \leftarrow (\emptyset, \emptyset)$
 2: $S \leftarrow \emptyset$
 3: for all $v \in V$ do
 4:    for all $P = v - v_2 - \cdots - v_k - v \in G, k \leq 6$ do
 5:       if $p$ is a simple cycle and $v = min\{w : w \in p\}$ and $v_2 < v_k$ then
 6:          $V' \leftarrow V' \cup \{P\}$
 7:          for all $w \in P$ do
 8:             $S[w] \leftarrow S[w] \cup \{P\}$
 9:          end for
10:       end if
11:    end for
12: end for
13: for all $v \in V$ do
14:    for all $w \in N_v(G, v) \cup \{v\}$ do
15:       for all $P \in S[v]$ do
16:          for all $Q \in S[w]$ do
17:             $E' \leftarrow E' \cup \{P - Q\}$
18:          end for
19:       end for
20:    end for
21: end for
22: return $independent\_set(G')$
```

vertices are adjacent.

- Compute an independent set of vertices in G'. The result corresponds to the

Algorithm 13 *cycle_reduction*(G, C)

Require: Planar bipartite cubic graph $G = (V, E)$.
Require: A simple cycle $C = (V', E') \subset G$. Edges of C are ordered by the occurrence on the cycle.
Ensure: Set of edges R such that $G[V - V', (E - E') \cup R]$ is a bipartite cubic graph.

```
1: $R \leftarrow \emptyset$
2: for $x = 0$ to $|C| - 1$ do
3:   if $x$ is even then
4:     $v - w \leftarrow E'[x]$
5:     $E' \leftarrow N_v(G, v) - V'$
6:     $E'' \leftarrow N_v(G, w) - V'$
7:     if $|E'| = 1$ and $|E''| = 1$ then
8:       $R \leftarrow R \cup \{E'[0] - E''[0]\}$
9:     end if
10:  end if
11: end for
12: return $R$
```

Algorithm 14 *cycle_reversion*(G, C, M)

Require: Planar bipartite cubic graph $G = (V, E)$.
Require: Simple cycle $C = (V', E')$ of G.
Require: Perfect matching M of G after reduction of cycle C.
Ensure: Set of edges R which extends M to the perfect matching of G.

```
1: $R \leftarrow \emptyset$
2: for $x = 0$ to $|C| - 1$ do
3:   if $x$ is even then
4:     $v - w \leftarrow E'[x]$
5:     $E' \leftarrow N_v(G, v) - V'$
6:     $E'' \leftarrow N_v(G, w) - V'$
7:     if $|E'| = 1$ and $|E''| = 1$ and $|\{E'[0] - E''[0]\} \cap M| = 1$ then
8:       $R \leftarrow R \cup \{v - E'[0], w - E''[0]\}$
9:     else
10:      $R \leftarrow R \cup \{v - w\}$
11:    end if
12:  end if
13: end for
14: return $R$
```

set of *non-conflicting cycles* in G.

Function *independent_cycles(G)* presented as Algorithm 12 provides an implementation of this approach. The entire matching algorithm for bipartite planar cubic graphs is implemented as *nc_planar_matching(G)* function presented as Algorithm

11.

Lemma 5.11. *For a given planar bipartite cubic graph G, Algorithm 12 computes a set of* non-conflicting cycles *of linear size.*

Proof. The graph G' representing all cycles of G not longer than 6 contains all faces of G not longer than 6, hence it follows From Lemma 5.9 that G' is of linear size. As G is cubic, each vertex of G' is of bounded degree (cycles of bounded length are incident to a constant number of cycles of bounded length), so by applying the maximal independent set algorithm, it is possible to compute an independent set of linear size in G'. This independent set corresponds to a set of *non-conflicting cycles* in G of linear size. □

Lemma 5.12. *It is possible to implement the Algorithm 11 so that it runs on $O(n)$ processors in $O(\log n \log^* n)$ time.*

Proof. As the sets of *non-conflicting cycles* computed by the algorithm are of linear size there are $O(\log n)$ recursion levels. The maximal independent set algorithm takes $O(\log^* n)$ time to compute a single set of *non-conflicting cycles*, hence the total execution time of this algorithm is $O(\log n \log^* n)$. Each parallel reduction and reversion step of *non-conflicting cycles* is performed in $O(1)$ time, as a single reduction/reversion of a bounded-size cycle can be performed by a single processor in constant time. The only remaining part is the construction of G' graph. It is easy to find all cycles not longer than 6 in a constant time. As graph G is cubic, the brute-force approach to analyze all paths of length up to 6 from a given source vertex v takes constant time. Each processor is assigned a different source vertex. Once a cycle is encountered, it can be added as a vertex to the graph G'. In order not to create more than one vertex for the same cycle, we can impose a linear ordering on vertices of G. If brute-force search was initiated from vertex v, a new vertex v' is to be added to G' only if v is the smallest vertex on a cycle corresponding to v' and the second vertex on a cycle is smaller than the last one. Such a limitation uniquely identifies each cycle of G. To construct all edges of graph G' it is sufficient for each vertex v of G' corresponding to a cycle C in G to add edges $v - w$ where w iterates over all vertices of G' corresponding to cycles C' in G adjacent to C. As all cycles are of bounded length it is possible to perform this step in constant time. □

5.3 Algebraic algorithm for general graphs

M. Mucha and P. Sankowski [39] proposed in 2004 a new randomized algebraic maximum matching algorithm for bipartite graphs running in $O(n^\omega)$ time[1]. They also managed to generalize this algorithm to all types of graphs preserving $O(n^\omega)$ execution time, but the generalization greatly complicated the algorithm. N. J. A. Harvey [29], by utilizing some algebraic properties of matrices, managed to simplify the algorithm by Mucha and Sankowski. The $O(n^\omega)$ algebraic maximum matching algorithm

[1]$O(n^\omega)$ is an optimal matrix multiplication time. It is known that $\omega < 2.376$.

applied to dense graphs is asymptotically faster than $O(n^{\frac{5}{2}})$ classical approach, however, because of a great complexity of utilized fast matrix multiplication procedure, it turns out to be impractical.

By replacing fast matrix multiplication with classical multiplication the idea proposed by Mucha and Sankowski leads to a simple $O(n^3)$ algorithm for arbitrary graphs. We will use it as a starting point for designing relatively simple parallel maximum matching algorithm for arbitrary graphs running in $O(n \log n)$ time with $O(n^2)$ processors. Such an approach has two advantages. Firstly, its total work of $O(n^3 \log n)$ is relatively small compared to other parallel matching algorithms. The second advantage is that it is possible to efficiently implement it on a GPU. We will provide such an implementation and analyze it's performance in the following chapter.

The presentation of our parallel algebraic algorithm is partitioned into four subsections. The first subsection presents an algebraic algorithm for computing perfect matching in bipartite graphs. The following subsection extends the algorithm to the class of general graphs. The third subsection presents a way of computing a maximum matching. We conclude this section with sequential and parallel implementation of the algorithm.

5.3.1 Perfect matchings in bipartite graphs

Consider a bipartite graph $G = (V, E)$ with bipartition (V', V'') and assume that $n = |V'| = |V''|$ (if it does not hold, a smaller set of vertices can be extended with adequate number of isolated vertices). The *Tutte matrix* T [60], [44] of size $n \times n$ representing the graph G is defined as follows:

$$T[x, y] = \begin{cases} 0 & \text{if } V'[x] - V''[y] \notin E \\ e_{x,y} & \text{if } V'[x] - V''[y] \in E \end{cases}$$

where $e_{x,y}$ is a unique variable. Each perfect matching M of G maps to a unique set of n non-zero elements of matrix T corresponding to the edges of M. *Determinant* of the matrix T ($det(T)$) is given by the following formula[2]:

$$det(T) = \sum_{\sigma \in S_n} sgn(\sigma) \Pi_{i=0}^{n-1} T[i, \sigma(i)]$$

Observe that graph G has a perfect matching if and only if $det(T) \neq 0$. As $det(T)$ is a symbolic polynomial (with variables $e_{x,y}$) we can replace all variables $e_{x,y}$ with random numbers obtaining a numerical matrix T'. If determinant of T' is not equal 0 (it is possible to compute the determinant of T' using classical $O(n^3)$ algebraic algorithm), it means that graph G has a perfect matching. It is not necessarily true that $det(T') = 0$ means that G does not have a perfect matching, but if we perform all computations in $\mathbb{Z}_p$ for sufficiently large prime p, the relation "G has a perfect matching if and only if $det(T') \neq 0$" holds with high probability. For the exact probability analyses please refer to [39].

[2] S_n is the set of all permutations σ of the numbers $\{0, 1, \ldots, n-1\}$. $sgn(\sigma)$ denotes the sign of the permutation σ: $+1$ if σ is an even permutation and -1 if it is odd.

By computing $det(T')$ we can convince ourselves that G has a perfect matching, but we don't know yet how this perfect matching looks like. To construct it we need to determine a permutation σ', for which $\Pi_{i=0}^{n-1} T'[i, \sigma'(i)] \neq 0$.

One way to accomplish this task is to start with empty matching M' and check every edge $V'[x] - V''[y] \in E$ if M' can be extended to a perfect matching using edge $V'[x] - V''[y]$. If $det(T' \setminus [x, y]) \neq 0$ it means that $V'[x] - V''[y]$ is an *allowed edge*, hence there exists a perfect matching in G containing it. We extend M' with the edge $V'[x] - V''[y]$ and remove vertices $V'[x]$ and $V''[y]$ from G. T' is also updated by removing the x-th column and the y-th row. The process of selecting edges continues until a perfect matching is constructed.

Naive computation of the determinant for each edge of G leads to the $O(n^5)$ algorithm (in the worst case it is required to examine $O(n^2)$ edges). Fortunately, the inverse matrix T'^{-1}, which can be computed in $O(n^3)$, can greatly reduce the computations. The following dependence between the determinant of T and the inverse of T holds:

$$T^{-1}[x, y] = (-1)^{x+y} \frac{det(T \setminus [y, x])}{det(T)}$$

For any non-singular matrix T, $det(T) \neq 0$, so by computing T^{-1} we get a fast probabilistic mechanism for checking in constant time whether a given edge $V'[x] - V''[y]$ is allowed (by making sure that $T^{-1}[y, x] \neq 0$). This improvement reduces time complexity of the algorithm from $O(n^5)$ to $O(n^4)$, as an inverse matrix has to be recomputed every time a matching being constructed is extended with a new edge.

If graph G had exactly one perfect matching, recomputation of the inverse matrix would not be required, as all edges $V'[x] - V''[y]$ for which $T^{-1}[y, x] \neq 0$ construct a single perfect matching. It turns out, however, that after extending a matching with an allowed edge $V'[x] - V''[y]$, it is possible to update T^{-1} in $O(n^2)$ time. All we need to do is a single Gaussian-elimination step of the y-th column and the x-th row of the T' matrix:

$$T^{-1} \leftarrow T^{-1} - \frac{T^{-1}[y, *] \cdot T^{-1}[*, x]}{T^{-1}[y, x]}$$

This leads to the $O(n^3)$ algebraic algorithm for constructing a perfect matching in a bipartite graph.

5.3.2 Perfect matching in general graphs

In order to find a perfect matching in an arbitrary graph $G = (V, E)$, some modifications to the algorithm from the previous section are required. First of all construction of matrix T needs to be different. In case of a bipartite graph, each vertex from V' has a unique column in matrix T assigned, while a vertex from V'' has a unique row. Such a construction makes it impossible to match a vertex from V' (V'') with a vertex from the same set. For bipartite graphs it is fine, as there are no edges between vertices in V' (V''). In general case, however, any pair of vertices can be connected by an edge, so construction of the matrix T needs to take this into account. Matrix T for general graphs has $n = |V|$ columns and rows — vertex $V[x], x \in \{0, 1, \ldots, n-1\}$, is assigned x'th column and x'th row. Tutte's matrix is defined as follows:

$$T[x,y] = \begin{cases} 0 & \text{if } V[x]-V[y] \notin E \\ e_{x,y} & \text{if } V[x]-V[y] \in E \text{ and } x<y \\ -e_{y,x} & \text{if } V[x]-V[y] \in E \text{ and } x>y \end{cases}$$

In case of bipartite graphs, a single non-zero determinant's product maps one-to-one to a perfect matching of a graph. In general case such a mapping does not hold, as each edge has two corresponding entries in the matrix. It turns out, however, that there is a mapping between perfect matchings and cycle-covers[3] of a graph, which makes it possible to use the determinant of matrix T for a construction of a perfect matching in a similar way as in case of bipartite graphs. The only modification required, when an allowed edge $V[x]-V[y]$ is detected, is an inverse matrix update process — in addition to a single Gaussian elimination step against row x and column y we need to perform elimination against row y and column x.

5.3.3 Maximum matching in general graphs

If a graph under consideration does not have a perfect matching, the determinant of matrix T is 0 and it is not possible to compute the inverse matrix. To find a maximum matching using algebraic methods, we need to remove from G some vertices, so that the remaining graph has a perfect matching. It can be achieved in terms of matrix operations by finding a maximum non-singular sub-matrix of T. Perform the Gaussian elimination of the matrix T and select all rows and columns against which eliminations were performed. The entire process takes $O(n^3)$ time.

5.3.4 Sequential implementation

In this section we define all implementation stages of the $O(n^3)$ sequential algebraic maximum matching algorithm for general graphs. It will be used in the following section as a starting point for an implementation of a parallel version of the matching algorithm that runs in $O(n \log n)$ time using $O(n^2)$ processors.

construction of the matrix T: For an input graph $G=(V,E)$ with n vertices, construct an empty matrix T with n columns and n rows. For each edge $V[x]-V[y] \in V$ pick a random integer $k \in \mathbb{Z}_p - \{0\}$ and update two entries of the matrix T in the following way:

$$T[x,y]=k,\ T[y,x]=k^{-1}$$

This stage takes $O(n^2)$ time.

construction of the non-singular sub-matrix T': Construct two empty lists A and B. Scan matrix T in the row-major order for non-zero elements. Once a non-zero element $T[x,y]$ is encountered, add vertices $V[x]$ and $V[y]$ to the lists A and B, respectively and perform the Gaussian elimination against $T[x,y]$

[3]To read more about perfect matchings and *cycle covers* please refer to [29].

and $T[y,x]$. Continue the scan process as long as there are some non-zero elements left. Construct a sub-matrix T' of T by selecting from T only columns corresponding to vertices from concatenated lists $[A,B]$ and rows corresponding to vertices from concatenated lists $[B,A]$. Make sure that order of columns and rows of T' correspond to the order of elements in the concatenated lists (this way no pivoting will be required in the next step of the algorithm). Up to n non-zero elements are found during the scan process, hence the total complexity of this stage is $O(n^3)$. Algorithm 20[4] presents a pseudo-code of this phase.

computation of the inverse of T': By performing the Gaussian elimination of the T' matrix we obtain three matrices — L, R and U, where $L\,R\,U = T'$, L is a lower triangular matrix, R is a diagonal matrix and U is an upper triangular matrix. T'^{-1} can be computed as follows:

$$T'^{-1} = U^{-1}\,R^{-1}\,L^{-1}.$$

LRU factorization: This step performs a linear number of Gaussian elimination steps against diagonal elements of the matrix T', recording at the same time coefficients of linear row and column combinations in L and U matrices respectively. A single elimination step takes $O(n^2)$ time, so it is possible to compute LRU factorization in $O(n^3)$ time. Algorithm 21 presents a pseudo-code of this phase.

inversion of the L matrix: Inversion of each row of the lower triangular matrix can be done independently. The process of computing the inverse is presented as Algorithm 22.

inversion of the R matrix: To inverse matrix R it is sufficient to compute the inverse of each diagonal entry of R. This step takes $O(n)$ time. The pseudo-code is presented as Algorithm 23.

inversion of the U matrix: To inverse matrix U, we can first transpose it, use L matrix inversion algorithm and then transpose the result again. Transposition is cheap (requires $O(n^2)$ operations). The pseudo-code is given by Algorithm 24.

$U^{-1}\,R^{-1}\,L^{-1}$ multiplication: First compute $S = U^{-1}\,R^{-1}$. R is diagonal, so it is sufficient to scale each column of U^{-1} by a constant (it requires $O(n^2)$ time). Afterwards compute $T'^{-1} = S\,L^{-1}$. It takes $O(n^3)$. The pseudo-code of the multiplication function is presented as Algorithm 25.

construction of matching M: Scan edges of graph G for the corresponding non-zero elements in the matrix T'^{-1}. If for a given edge e, $T'^{-1}[x,y] \neq 0$, it means that e is allowed. Add e to the matching M being constructed and perform the Gaussian elimination of T'^{-1} against rows and columns corresponding to vertices adjacent to e:

$$T^{-1} \leftarrow T^{-1} - \frac{T^{-1}[*,x]\cdot T^{-1}[y,*]}{T^{-1}[y,x]}$$

[4]Algorithms 20 — 35 are collected in the Appendix.

$$T^{-1} \leftarrow T^{-1} - \frac{T^{-1}[*,y] \cdot T^{-1}[x,*]}{T^{-1}[x,y]}$$

The pseudo-code of this phase is presented as Algorithm 26. The entire sequential algebraic matching algorithm is presented as Algorithm 27

5.3.5 Parallel implementation

In this section we turn the sequential algebraic maximum matching algorithm from the previous section into a PRAM [34] version running in $O(n \log n)$ time and utilizing $O(n^2)$ processors. We accomplish this by turning each step of the sequential algorithm into a parallel version.

construction of the T matrix: By assigning a single processor to each pair of elements $T[x,y]$ and $T[y,x]$, construction of the matrix T can be completed in $O(1)$ time. Each processor assigned to element $T[x,y]$ verifies whether $x-y \in E$ and if so two elements of matrix T are updated — $T[x,y] = k$, $T[y,x] = k^{-1}$, where k is a random integer from $\{1,\ldots,p-1\}$

construction of the T' sub-matrix: By assigning a single processor to each element of the matrix T it is possible to find successive non-zero elements of T in $O(1)$ time. In the similar way a single Gaussian elimination step can be performed. Construction of the T' matrix also takes $O(1)$ time. The total execution time of this phase is hence $O(n)$. Sample pseudo-code is presented as Algorithm 28.

inversion of the T' matrix: Matrix inversion can be computed in $O(n \log n)$ time. The bottleneck is computation of L^{-1} and U^{-1}.

LRU factorization: The parallel factorization in $O(n)$ time is similar to the parallel Gaussian elimination applied to the matrix T. The only difference is the additional requirement — construction of matrices L and U, which can be easily obtained. The sample pseudo-code is presented as Algorithm 29.

inversion of the L matrix: Fast parallel inversion of a lower triangular matrix is not so simple, as there are strong dependencies between matrix entries. By investigating the pseudo-code of the sequential implementation of this phase, one can notice that computation of entries from the different columns is independent. This allows to compute all columns in parallel. The pseudo-code is given as Algorithm 30. Such an approach uses $O(n)$ processors and takes $O(n^2)$ time. The other phases of the algorithm run in $O(n)$ time, so it is worth trying to further speed up the inversion of the L matrix. It is possible to compute consecutive entries of the result matrix with a parallel approach. Instead of spending $O(n)$ time on sequential sum computation (lines $5-7$ of Algorithm 30), $O(\log n)$ parallel phases can be executed, each of them reducing the size of the sum being computed by half. This provides an algorithm for computing the inverse of the L matrix in $O(n \log n)$ time using $O(n^2)$ processors.

inversion of the R matrix: All entries of the diagonal matrix, as already noticed in the previous section, can be computed independently of each other, which leads to a simple $O(1)$ time algorithm utilizing $O(n)$ processors (the pseudo-code is presented as Algorithm 31).

$M'^{-1} = U^{-1}\,R^{-1}\,L^{-1}$ **multiplication:** Multiplication of two $n \times n$ matrices can be computed in $O(n)$ time using $O(n^2)$ processors. This is a simple result following from the fact that computation of each entry of the result matrix is independent. Example pseudo-code is given as Algorithm 33. Matrix multiplication can also be computed in polylogarithmic time, but we are interested in an approach that provides an efficient execution on a GPU.

Putting all phases of the PRAM algorithm together we get $O(n \log n)$ total execution time utilizing $O(n^2)$ processors. This is much faster from the sequential $O(n^3)$ time algorithm, however, the total work is larger by a $O(\log n)$ factor.

Chapter 6

Practical aspects of the matching problem

The matching problem has been analyzed for over a century from many different perspectives. A number of approaches have been utilized — alternating paths, flows, randomization, algebraic algorithms, just to name a few. Depending on the type of a problem being solved numerous types of graphs were considered — general graphs, bipartite, biconnected, planar, regular, cubic... Over time a lot of efficient algorithms were proposed to solve subclasses of the general matching problem. In practice, however, theoretically efficient algorithms sometimes turn out to be slow or even hardly implementable. In this chapter we analyze the practical value of some matching algorithms and we try to answer the question how different subclasses of the matching problem can be solved effectively. We start with an introduction of one more algorithm — reduction of the perfect matching problem to the boolean satisfiability problem. Description of a porting of the parallel algebraic maximum matching algorithm from section 5.3 to a GPU architecture follows. The chapter is concluded with performance evaluation of different matching algorithms.

6.1 Perfect matching construction via boolean satisfiability

When searching for an efficient solution to the perfect matching problem in biconnected cubic graphs we tried many different approaches. One of them was a reduction of the perfect matching problem to the SAT problem. Despite the lack of the theoretical upper bound on the complexity of such an algorithm, it turns out to be quite effective in practice. We have decided to present this algorithm and provide a performance comparison against other matching algorithms.

If a given graph $G = (V, E)$ has a perfect matching it is possible to construct an instance of the SAT problem, which solution encodes a perfect matching of G. For a given graph G we are going to construct an instance of the SAT problem with $|E|$ variables where each edge is assigned a single variable. Variable assigned to an edge e is evaluated to TRUE if the perfect matching being constructed contains e. For each vertex $v \in V$ with incident edges $N_e(G, v) = \{e_1, e_2, \ldots, e_k\}$ we generate the following set of clauses:

$$e_1 \vee e_2 \vee \cdots \vee e_k$$

$$\forall_{e_1, e_2 \in N_e(G,v), e_1 \neq e_2} \neg e_1 \vee \neg e_2$$

The first clause makes sure that for each vertex $v \in V$ at least one edge incident to v is matched. This way if SAT problem formule is satisfiable we are guaranteed that each vertex is incident to an edge from the constructed matching. The rest of the clauses make sure that for each vertex $v \in V$ at most one edge incident to it is matched (otherwise the set of edges for which SAT variables are evaluated to TRUE might not form a valid matching). This way any satisfiable evaluation of the SAT problem formule maps uniquely to a perfect matching of G.

The instance of the SAT problem generated for graph $G = (V, E)$ consists of

$$|V| + \sum_{v \in V} \frac{N_e(G, v) \cdot (N_e(G, v) - 1)|}{2} \text{ clauses}$$

and contains

$$|E| + \sum_{v \in V} |N_e(G, v)| \cdot (|N_e(G, v)| - 1) \text{ literals}$$

In case of bounded degree graphs an instance of the SAT problem is of $O(n)$ size, which is of the same asymptotic size as the input graph. In case of dense graphs, however, the size of the SAT instance grows to $\theta(n^3)$. We cannot expect an algorithm with $\theta(n^3)$ input to outperform other competing matching algorithms, as the size of the SAT instance is bigger than the time complexity of the state of the art general matching algorithms.

We can overcome this problem by first constructing a graph G' with maximum vertex degree 3, such that G has a perfect matching if and only if G' has one. In addition it must be possible to efficiently convert a perfect matching of G' to a perfect

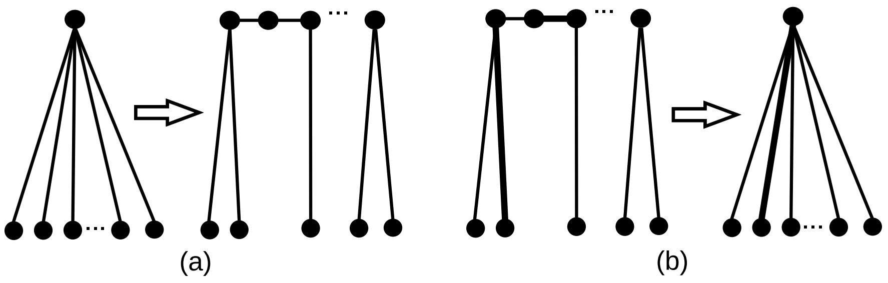

Figure 6.1: **(a)** The process of reducing degree of a vertex. To reduce a vertex of degree k ($k > 3$) it is required to introduce $2(k-3)$ new vertices and edges. Reduction of the entire graph leads to a new, almost-cubic graph with $O(n + m)$ vertices and edges. **(b)** The process of reconstructing a matching from a matching found for the almost-cubic graph (matched edges are presented in bold).

matching of G. The Figure 6.1 demonstrates the process of reducing a degree of a vertex to up to 3 and the process of reconstructing a matching of G from a matching of G'. Almost cubic graph G' constructed for an input graph $G = (V, E)$ consists of

$$|V| + \sum_{v \in V} max(0, 2 \cdot |N_e(G, v)| - 6) \text{ vertices}$$

and

$$|E| + \sum_{v \in V} max(0, 2 \cdot |N_e(G, v)| - 6) \text{ edges}$$

As all vertices of G' are of degree up to 3, size of the SAT problem formule generated for G' is asymptotically the same as the size of G ($O(|V| + |E|)$). Some more details on reducing the degree of a graph can be found in [5].

In general, solving the SAT problem is an NP-complete task. In case of our problem most of the clauses are in 2-CNF form (2-CNF is solvable in linear time), which gives us some hope for relatively fast processing (however, still exponential). The difficulty to solve an instance of the SAT problem also depends on the number of satisfiable evaluations. In general, more satisfiable solutions of the SAT problem makes it easier to find one of them. In our case each solution to the SAT problem maps directly to the perfect matching of the graph, hence the number of perfect matchings is important to us. In the 1970's Lovasz and Plummer [38] made the following conjecture:

Conjecture 6.1 (Lovash-Plummer). *Let G be an arbitrary n-vertex cubic graph without a cut-edge. There exists a fixed constant c such that G has at least e^{cn} perfect matchings.*

This conjecture has been proven for two important classes of cubic graphs — bipartite and planar. It is deeply believed that it also holds for all cubic bridgeless graphs. This is crucial for a SAT solver to find solution to the matching problem in a reasonable amount of time.

In case a graph does not have a perfect matching at all, it may take a very long time to determine that an instance of the SAT problem is unsatisfiable. We have executed our SAT solver based matching algorithm against a number of graphs without a perfect matching. For many input graphs, SAT solver was able to quickly find a contradiction in the input formula. In some cases, however, it failed to find such a contradiction which resulted in very long execution. For one of the extreme cases an input graph had 21 vertices and 120 edges (after reduction to an almost-cubic graph it had 375 vertices and 474 edges). It took 42 seconds to determine there is no perfect matching in this graph. In case of graphs with many perfect matchings, however, SAT technique provides an efficient algorithm for constructing a perfect matching.

6.2 Maximum matching on a GPU

Nowadays parallelism is becoming the only tool for increasing performance of computational expensive tasks. It is widely practiced to apply parallelism to speed up many

Table 6.1: Performance of a CPU implementation of the algebraic matching algorithm

Graph size (# of vertices / # of edges)	Submatrix computation	Matrix inverse	Matching construction	Total
256 / 13 107	0.015 s	0.029 s	0.135 s	0.179 s
512 / 52 428	0.071 s	0.140 s	1.091 s	1.302 s
1 024 / 209 715	0.363 s	0.730 s	9.164 s	10.257 s
2 048 / 838 860	2.540 s	4.597 s	74.988 s	82.125 s
4 096 / 3 355 443	14.209 s	25.760 s	419.292 s	459.261 s

real-life graph based problems as well. Some of them can be easily parallelized — breadth-first search is a flagship example. However, there is a class of graph problems for which efficient sequential algorithms have not yet been outperformed by applying parallelism. Maximum matching is one of them.

In this section we present a practical parallel matching algorithm adopted to a GPU architecture. Our starting point is the PRAM algorithm presented in section 5.3 which runs in $O(n \log n)$ time using $O(n^2)$ processors. We will adopt this algorithm for GPU (Graphics Processor Unit) by taking into account nowadays hardware architecture and by reducing the total algorithm's work to $O(n^3)$. We are not pursuing the highly tuned implementation. We just want to prove that it is possible for some groups of graphs to outperform efficient sequential matching algorithms by applying parallel computation on nowadays hardware.

Before porting the parallel algebraic matching algorithm to a GPU we first implement it on a CPU using FFLAS-FFPACK library [3]. This library provides linear algebra algorithms over a finite field which runs almost as fast as floating point equivalents. In order to implement maximum matching algorithm we utilized three functions from this library. To compute a maximum non-singular sub-matrix LQUP matrix factorization implemented as a *FFPACK::LUdivine* function was utilized. *FFPACK::Invert* was used to compute the matrix inverse, while *FFPACK::fgemm* was used for matrix-matrix multiplication. We have executed the algorithm against random graphs of different sizes. The results obtained are presented in the Table 6.1.

CPU-based matching algorithm indeed has $O(n^3)$ time complexity — doubling the number of vertices increases the execution time of the algorithm 8 times. It turns out that most of the time ($\sim 90\%$) is spent in the last phase of the algorithm — matching construction. This phase consists of two parts — searching for non-zero elements in the inverse matrix (to find the allowed edges) and performing the inverse matrix update, which utilizes matrix-matrix multiplication function. The cumulative execution time of the search process is $O(n^2)$, so it is marginal compared to the matrix update. The multiplication phase, on the other hand, is quite complex — not only it performs the total work of $O(n^3)$, but it also executes matrix-vector multiplication $O(n)$ times, which does not allow to reach the peak performance of a CPU. To confirm this statement we have performed a small experiment. We have done a single multiplication of two 2048×2048 matrices vs 2048 multiplications of a 2048×2048 matrix by 2048 elements vector. The total theoretical work in both cases

Table 6.2: Execution time of the third phase of the algebraic matching algorithm depending on the value of parameter g

Graph size / g	1	2	4	8	16	32	64	128
256 / 13 107	0.13 s	0.07 s	0.04 s	0.02 s	0.01 s	0.01 s	0.01 s	0.04 s
512 / 52 428	1.09 s	0.58 s	0.28 s	0.15 s	0.09 s	0.07 s	0.08 s	0.12 s
1 024 / 209 715	9.16 s	4.41 s	2.28 s	1.22 s	0.76 s	0.59 s	0.62 s	1.00 s
2 048 / 838 860	74.99 s	35.74 s	18.01 s	9.42 s	5.50 s	3.98 s	3.74 s	4.72 s
4 096 / 3 355 443	549 s	277 s	142 s	74 s	42 s	29 s	24 s	68 s

Table 6.3: Performance comparison of the algebraic algorithm after optimization

Graph size (# of vertices / # of edges)	Old algorithm	Optimized algorithm	Speed up
256 / 13 107	0.179 s	0.060 s	3.0
512 / 52 428	1.302 s	0.291 s	4.5
1 024 / 209 715	10.257 s	1.712 s	6.0
2 048 / 838 860	82.125 s	10.894 s	7.5
4 096 / 3 355 443	591.943 s	65.670 s	9.0

is the same, however the first algorithm executes over 80 times faster. Before porting our algebraic algorithm to a GPU architecture we need to address this performance issue.

Instead of updating the inverse matrix for each edge detected it is possible to group g updates and apply them at once. When searching for the next edge to be added to the matching not yet applied changes need to be taken into account. This way the cost of scanning for non-zero entries increases to $O(\mathrm{g}\,n^2)$ time (checking each matrix entry requires taking into account up to g not yet applied updates), while the number of matrix multiplications is reduced g times. The table 6.2 presents execution times of the third phase of the algorithm depending on the value of g.

By analyzing the results of the experiment it turns out that it is optimal to select $g = \theta(\sqrt{n})$. By applying the optimization, the original algorithm accelerates up to 9 times (see the performance comparison presented in the Table 6.3).

Implementations of many matrix based algorithms (like matrix-matrix multiplication or triangular solvers) for CPUs reach almost 100% of the theoretical hardware performance. In case of GPUs this kind of computations turn out to be memory bandwidth bounded which is due to the very limited size of GPU caches. Apart from this limitation it has been recently shown [62] that it is possible to implement many matrix based algorithms very efficiently. V. Vasily and J. Demmel managed to obtain over 180 Gflop/s for GEMM (matrix-matrix multiplication) problem utilizing a GPU with theoretical peak of 346 Gflop/s. They also managed to accelerate TRSM (triangular solver) and SYR2K (symmetric update) reaching 30% of GPU's peak performance. LU factorization reaches 140 Gflop/s. Recent GPUs exceed 1 Tflop/s, so by applying them to the algebraic maximum matching problem, it should be possible to obtain 100x speed-up in comparison with CPU based implementation. Unfortunately, we are not aware of any finite field linear algebra package for GPUs, so we developed a simple implementation of matrix-based parallel algebraic maximum matching algorithm from the scratch, which is presented in the following subsection.

6.2.1 Parallel maximum matching with GPU

Asymptotic complexity of our PRAM algorithm is much better than $O(n^{\frac{5}{2}})$ Edmond's based algorithms or $O(n^{\omega})$ of Mucha and Sankowski algorithm. However, it doesn't mean that the parallel algorithm will run faster in practice. In order to outperform sequential algorithms we need to take into account a number of limitations imposed by GPU architecture:

Non-optimal work. Our PRAM algorithm performs the total work of $O(n^3 \log n)$, which is worse than $O(n^{\frac{5}{2}})$ Edmond's approach-based algorithm or even simple $O(n^3)$ matrix-based approach. In contrast to the PRAM theoretical model where time complexity is the only factor of the algorithm's efficiency, in the real world total work also plays a crucial role. This is not only because of limited number of available processors, but also because of communication limitations which are not taken seriously in many theoretical models.

Insufficient processors number. Maximum matching algorithms can be applied to graphs with thousands of vertices. It means the PRAM algorithm presented in section 5.3 to run with full speed requires millions of processors. Nowadays GPUs are equipped with hundreds of computation units (128 for Nvidia GeForce 8800) that can run up to thousands of interleaved threads. It is not sufficient for our input graphs to be processed with full parallelism. In many cases it is possible, by reducing the number of processors utilized by the algorithm, to reduce the total work and decrease the total execution time.

Global memory utilization. Nowadays GPUs can execute a tremendous number of floating point operations per second. The most recent units exceed 1 Tflop/s. At the same time the speed of GPUs' global memory stays behind — it is hardly possible to go beyond 120 GB/s. In order to obtain the maximum memory bandwidth one needs to adhere to many hardware restrictions — sequential memory access is a must. If we wanted to run the matrix multiplication algorithm presented in the Section 5.3.5, we would need to transfer $8 \cdot n + 4$ bytes to compute a single matrix entry. This accounts for $8 \cdot n^3$ bytes to multiply 2 matrices. Assuming the perfect global memory access pattern, we wouldn't be able to go beyond 15 Gflop/s. This is less than 3% of the peak computational performance of GPU. In order to increase the performance we need to reduce the global memory usage by utilizing GPU's shared memory and registers.

Processor synchronization. The SIMD model we used for designing the PRAM algorithm assumes that all processors run with a synchronized clock. This guarantees that all processors execute exactly the same instruction at the same time. In case of GPUs this is not true. GPUs are a combination of SIMD and MIMD model — they consist of a number of asynchronous multiprocessors, each of them containing a number of SIMD processors. In addition, each SIMD processor runs a number of interlaced threads in order to reduce global memory latency. Straightforward application of the PRAM algorithm would require additional synchronization mechanisms not only within a single multiprocessor but also between different multiprocessors, which is extremely expensive.

Thread execution coherence. Each multiprocessor within a GPU is a SIMD machine executing interlaced warps of threads. SIMD model assumes that every thread executes exactly the same instruction at the same time. GPU architecture allows for conditional instructions, loops, etc. In practice, in case of a different conditional statement evaluation by threads within the same warp, both paths of the conditional statement are executed sequentially — once for a group of threads evaluating the condition to true, once for the rest of threads. In case of multi-level conditional statements and loops this situation gets even worse. In order to obtain an efficient algorithm it is a good practice to split work in such a way that threads within a single warp evaluate all conditional statements in the same way. As an example — if one wants to divide threads into two equally sized groups, it is not a good idea to assign even threads to the first set, while odd threads to the second set. In case of our algebraic maximum matching algorithm it is not hard to do an optimal threads assignment, as there is just a little conditional logic involved.

Thread synchronization. __sync function used in the pseudo-codes of GPU implementation is responsible for synchronizing execution of all threads running within a single multiprocessor. This instruction does not enforce cross multiprocessor synchronization. However, it still has a negative performance impact, as it makes efficient pipelining impossible. We already mentioned that GPU executes a group of threads (a warp of size m, usually 16 or 32) synchronously — in SIMD manner. If our GPU algorithm performs group calculations in $m \times m$ blocks it is possible to assign threads to block elements in a way eliminating the need for some of the synchronization operations.

Shared memory organization. Shared memory within a single multiprocessor is organized in m banks (the number of banks is equal to the size of the warp). The i'th bank contains integers with addresses given by the formula $m \cdot n + i, n \in \mathbb{N}$. Each bank can perform a single memory operation at a time, so if a number of threads from a warp access shared memory from the same bank, it causes some additional time penalty. The best situation is when there are no bank conflicts. In order to optimize threads synchronization computations are performed in $m \times m$ blocks, hence simultaneous access to m elements of a single column causes a m-threads bank conflict. This problem can be eliminated by allocating shared memory matrices of size $m + 1 \times m + 1$, so that m consecutive elements of a single column, as well as of a single row, are located in different banks and no conflicts are encountered.

Section 6.2.2 presents an implementation of the PRAM maximum matching algorithm from section 5.3.5, which has been tuned for a GPU architecture. In order to reduce algorithm's total work, inversion of L and U matrices are computed in $O(n^2)$ time using $O(n)$ processors (it saves an $O(\log n)$ factor of the algorithm's total work). In spite of the fact that implementation of our algorithm aims at clarity rather than performance, it is sufficient to outperform sequential matching algorithms for some classes of graphs as presented in the performance evaluation section.

It is possible to further optimize our GPU algorithm. For more details on efficient implementations of matrix-based algorithms please refer to [13], [50], [37], [63].

6.2.2 Maximum matching algorithm implementation on GPU

In this section we present an adoption of the PRAM algebraic maximum matching algorithm to a GPU architecture. Restrictions of the real hardware are taken into account in order to present a fast, yet simple algorithm. The algorithm consists of the same phases as the PRAM algorithm — finding a subgraph G' of a graph G containing a perfect matching, computation of the inverse matrix and construction of the matching. As the general design of the algorithm is similar to the PRAM version, we only present GPU implementation of the following "building blocks":

- square matrix multiplication
- Gaussian elimination,
- inversion of the lower triangular matrix.

Square matrix multiplication on GPU

By analyzing the matrix multiplication algorithm for the PRAM model it is easy to see that the main performance issue in case of GPU is a global memory utilization. Matrix multiplication in the PRAM approach does not require any processors' synchronization — each processor computes one matrix entry and its work is independent from the work of the others. Each processor reads $2 \cdot n$ elements from the global memory which accounts for the total $2 \cdot n^3$ global memory accesses. In case of GPU this number can be reduced by the proper utilization of shared memory and registers. As an example, let us consider architecture of Geforce 8800 GTX. Each multiprocessor within the GPU can run simultaneously (interlaced) up to 512 threads. Each thread is assigned the same number of registers for storing local variables and parameters (8192 4-byte registers are available within a single multiprocessor). All threads within the same multiprocessor can also use 16 KB of shared memory with the same access speed as registers. Multiprocessor runs a group of consecutive threads synchronously (SIMD model) — such a group of threads is called a warp and is of size 16 in case of Geforce 8800 GTX. Multiprocessor switches a group of executed threads periodically. Hardware architecture makes it possible to synchronize execution of all threads running on a single multiprocessor — we will use *__sync* instruction to denote this operation. *__sync*, otherwise known as a barrier, blocks execution until all threads within a single multiprocessor reach a barrier. Synchronization of threads running on different multiprocessors is not supported by the hardware and can be realized by global memory synchronization algorithms. Such synchronization is very expensive and should be rarely used. In order to reduce the effect of global memory transfers our algorithm will copy small parts of the input matrices to the shared memory and perform multiplication operations internally without accessing global memory. The pseudo-code presented as the Algorithm 15 computes the multiplication of two input matrices. By MUL_CNT we denote the number of multiprocessors within a GPU, by THR_CNT the number of threads per multiprocessor. We assume that $THR\_CNT = 256$ and that the size of a warp is 16 (there is no need to synchronize execution of each group of 16 consecutive threads).

Algorithm 15 $gpu\_multiplication(A, B)$

```
Require: Two matrices $A$ and $B$ of size $n \times n$ to be multiplied
Ensure: Multiplication of matrices $A$ and $B$
 1: $size \leftarrow \sqrt{THR\_CNT}$
 2: $C \leftarrow 0_{n \times n}$ /* located in global memory */
 3: $A' \leftarrow 0_{size \times size}$ /* located in shared memory */
 4: $B' \leftarrow 0_{size \times size}$ /* located in shared memory */
 5: for all $blk$ such that $0 \leq blk < MUL\_CNT$ do
 6:   for all $thr$ such that $0 \leq thr < THR\_CNT$ do
 7:     $x\_id \leftarrow thr/size$
 8:     $y\_id \leftarrow thr\%size$
 9:     for $x = size \cdot blk$; $x < n$; $x += size \cdot MUL\_CNT$ do
10:       for $y = 0$; $y < n$; $y += size$ do
11:         $sum \leftarrow 0$
12:         for $z = 0$; $z < n$; $z += THR\_CNT$ do
13:           $A'[x\_id][y\_id] \leftarrow A[x\_id + z][y\_id + y]$
14:           $B'[x\_id][y\_id] \leftarrow B[x\_id + x][y\_id + z]$
15:           __sync
16:           for $i = 0$ to $size - 1$ do
17:             $sum \leftarrow sum + A'[i][y\_id] \cdot B'[x\_id][i]$
18:           end for
19:           __sync
20:         end for
21:         $C[x + x\_id][y + y\_id] = sum\%P$
22:       end for
23:     end for
24:   end for
25: end for
26: return $C$
```

The matrix multiplication algorithm divides the output matrix into 16×16 blocks and assigns to each block 256 threads of a single multiprocessor (lines 5—10 of Algorithm 15, a single block is identified by its first matrix entry $[x, y]$). Notice that size of a block is equal to the number of threads assigned to it, so each thread can take care of computing the value of one output matrix entry — just like it was in case of PRAM implementation. All 256 threads share memory regions A' and B' in order to reduce the global memory load. Each thread stores the result being computed in a local variable *sum* (register). Consecutive blocks of A and B matrices are being loaded to the shared memory in lines 12—14, A' and B' sub-matrices are then used to update partial sums (lines 15—18). Thread synchronization executed in lines 15 and 19 is required to make sure that A' and B' have a correct context before using them for computation of the partial sums.

In order to compute 16×16 block of the result matrix this algorithm loads $32 \cdot n$ words from global memory (in comparison with $512 \cdot n$ words for PRAM algorithm). The whole matrix multiplication requires $\frac{n^3}{8}$ loads compared to $2 \cdot n^3$ for the PRAM approach. What is more, all global memory accesses are sequential (threads within a warp access consecutive memory addresses) allowing for high global memory bandwidth. In order to further reduce global memory usage one can increase the size of blocks to 32×32 (64×64 blocks are too big to fit 16 KB shared memory region). It is also possible to utilize GPU registers for storing one of the matrices which can additionally increase the algorithm's performance [1].

Gaussian elimination on GPU

The Gaussian elimination process can be seen as repeatedly performing two stages — search for a non-zero matrix entry and the matrix update. The total execution time of the sequential search is $O(n^2)$, as each matrix entry has to be analyzed not more than once. The single matrix update process, on the other hand, requires $O(n^2)$ time and can be described in the following way:

$$M \leftarrow M - C \cdot e \cdot R,$$

where C is a column matrix $(1 \times n)$, e is a scalar, R is a row matrix $(n \times 1)$. Even, if we reorganized the way PRAM algorithm shares work among threads, we will not be able to reduce global memory usage, as each update phase needs to load and store the whole matrix M.

To reduce the global memory load we need to apply the same trick as in our experimental FFLAS-FFPACK implementation from section 6.2 — perform matrix updates in batches. Once a number of rows and columns to be eliminated are found, all updates can be applied at once. The pseudo code presented as Algorithm 16 implements this idea.

The main loop in lines $9 - 33$ performs elimination of consecutive matrix rows (from 0 to $n - 1$). Loop $10 - 15$ searches for a non-zero matrix entry within the row

[1]The performance increase in case of utilizing registers is not only due to possibility of using larger blocks. It is also caused by GPU architecture limitations which makes it impossible for a single thread to access two shared memory locations within a single clock cycle. Some experiment results in this area can be found in [49].

Algorithm 16 *gpu_elimination*(M)

```
Require: $n \times n$ matrix being eliminated
Ensure: Eliminated matrix
 1: $size \leftarrow \sqrt{THR\_CNT}$
 2: $C \leftarrow 0_{size \times n}$
 3: $R \leftarrow 0_{n \times size}$
 4: $E \leftarrow 0_{size \times size}$
 5: $[P.x, P.y] \leftarrow [-1, -1]$ /* matrix index */
 6: $[P'.x, P'.y] \leftarrow [-1, -1]$ /* shared memory matrix index */
 7: for all $thr$ such that $0 \leq thr < size$ do
 8:   $to\_reduce \leftarrow 0$
 9:   for $y = 0$ to $n - 1$ do
10:     for $x = thr; x < n; x+ = size$ do
11:       $val \leftarrow gpu\_compute(M, E, C, R, to\_reduce, x, y)$
12:       if $val \neq 0$ then
13:         $[P'.x, P'.y] \leftarrow [x, y]$
14:       end if
15:     end for
16:     __sync
17:     if $[P'.x, P'.y] \neq [-1, -1]$ then
18:       use $P'$ element for matching / non-singular matrix construction
19:       $E[to\_reduce] \leftarrow M[P']$
20:       for $z = thr; z < n; z+ = size$ do
21:         $val = gpu\_compute(M, E, C, R, to\_reduce, z, P'.y)$
22:         $C[to\_reduce, z] \leftarrow val$
23:         $val = gpu\_compute(M, E, C, R, to\_reduce, P'.x, z)$
24:         $R[z, to\_reduce] \leftarrow val$
25:       end for
26:       $to\_reduce \leftarrow to\_reduce + 1$
27:     end if
28:     if $to\_reduce = size$ then
29:       $P \leftarrow P'$
30:       $gpu\_update()$
31:       $to\_reduce = 0$
32:     end if
33:   end for
34: end for
```

being analyzed and stores its index into P variable. This loop is executed using only one multiprocessor. If such an element is found matrices used for the lazy update are computed in lines $20-25$. Afterwards, if those matrices are full the lazy elimination is performed (lines $28-32$). The pseudo-code makes use of two macros — *gpu_compute* and *gpu_update*. *Gpu_compute* macro calculates the current value of a given entry of the matrix M taking into account updates stored in C, R and E. The pseudo-code of *gpu_compute* function is presented as Algorithm 17.

Algorithm 17 *gpu_compute*$(M, E, C, R, reduced, x, y)$

```
Require: An inverse of a Tutte matrix representing a graph
Require: Vector with Gaussian elimination coefficients
Require: Matrix with Gaussian elimination coefficients
Require: Matrix with Gaussian elimination coefficients
Require: The number of not yet applied Gaussian eliminations
Require: Row and column index of the matrix element to be computed
Ensure: Value of the matrix entry under question after taking into account not yet
    applied eliminations
1: $val \leftarrow M[x, y]$
2: for $z = 0$ to $reduced$ do
3:     $val \leftarrow val + E[z] \cdot C[z, y] \cdot R[x, z]$
4: end for
5: return $val$
```

Notice that *gpu_compute* macro is executed by all threads of a single multiprocessor (line 11, 21, 23 of Algorithm 16). A single execution of this macro performs up to 48 global memory access and is executed $3 \cdot n^2$ times, so it does $144 \cdot n^2$ global memory operations.

Gpu_update macro is responsible for applying a number of pending updates to the matrix M. It runs 16 updates at a time so similar approach as in the matrix multiplication can be utilized to reduce the global memory load. *Gpu_update* macro is executed by all threads within all multiprocessors (line 30 of Algorithm 16). To update a single 16×16 block of a matrix M it is required to load 768 global memory words, hence the total number of global memory operations performed by *gpu_update* macro is $3 \cdot n^2$. It is called up to $\frac{n}{16}$ times, so the total number of global memory operations is $\frac{3 \cdot n^3}{16}$. The PRAM algorithm needs $3 \cdot n^3$ memory accesses — 16 times more. The pseudo-code of *gpu_update* function is presented as Algorithm 18.

Inversion of the lower triangular matrix on GPU

By investigating the PRAM implementation of the lower triangular matrix inversion it turns out that in order to compute $[x, y]$ entry of the inverse matrix it is required to compute all $[x, k]$ entries for $k \in \{0, \ldots, y - 1\}$ first. There are no dependencies between different columns of the matrix, so each multiprocessor can be assigned a number of columns to process without the need of synchronization. The PRAM lower matrix inversion algorithm does not require synchronous execution of processors, so it is possible to apply that algorithm to GPU without additional modifications. However, such a solution is not optimal in terms of global memory usage. By applying the idea presented in the matrix multiplication algorithm for the GPU, the memory load can be reduce in the following way:

The algorithm uses 256 threads on each multiprocessor to compute 16 consecutive columns of the result matrix. The pseudo-code is presented as Algorithm 19. Line 9

Algorithm 18 *gpu_update*(M, E, C, R)

Require: An inverse of a Tutte matrix representing a graph
Require: Vector with Gaussian elimination coefficients
Require: Matrix with Gaussian elimination coefficients
Require: Matrix with Gaussian elimination coefficients
Ensure: An input matrix with all updates applied

```
 1: $size \leftarrow \sqrt{THR\_CNT}$
 2: $C' \leftarrow 0_{size \times size}$ /* located in shared memory */
 3: $R' \leftarrow 0_{size \times size}$ /* located in shared memory */
 4: $E' \leftarrow 0_{size \times size}$ /* located in shared memory */
 5: for all $blk$ such that $0 \leq blk < MUL\_CNT$ do
 6:   $x\_id \leftarrow thr/size$
 7:   $y\_id \leftarrow thr\%size$
 8:   if y_id $= 0$ then
 9:     $E'[x\_id] = E[x\_id]$
10:   end if
11:   for $x = size \cdot blk; x < n; x+ = size \cdot MUL\_CNT$ do
12:     for $y = 0; y < n; y+ = size$ do
13:       $val \leftarrow M[x + x\_id][y + y\_id]$
14:       $C'[x\_id][y\_id] \leftarrow C[x + x\_id][y\_id]$
15:       $R'[x\_id][y\_id] \leftarrow R[x\_id][y + y\_id]$
16:       __sync
17:       for $z = 0$ to $size - 1$ do
18:         $val \leftarrow val + C'[z, y\_id] \cdot R'[x\_id, z] \cdot E'[z]$
19:       end for
20:       $M[x + x\_id, y + y\_id] \leftarrow val\%P$
21:       __sync
22:     end for
23:   end for
24: end for
25: return M
```

divides groups of 16 columns between different multiprocessors while line 10 computes consecutive 16×16 blocks. First, in lines $12-20$, for each element $[x, y]$ of the block we compute a $sum = \sum_{i=0}^{16 \cdot \lfloor \frac{y}{16} \rfloor - 1} L[i, y] \cdot L^{-1}[x, i]$. Later on, in lines $22-30$, the remaining $\sum_{i=16 \cdot \lfloor \frac{y}{16} \rfloor}^{y} L[i, y] \cdot L^{-1}[x, i]$ is computed, taking additional care, as L^{-1} entries used for this computation are just being computed.

The PRAM version of the lower matrix inversion algorithm executed n^3 global memory accesses. The GPU version accesses the global memory only $\frac{n^3}{16}$ times, which allows for efficient GPU utilization. This last "building block" completes GPU parallel implementation of the maximum matching algorithm for arbitrary graphs.

Algorithm 19 *gpu_L_inverse*(G)

```
Require: $n \times n$ lower triangular matrix $L$ to be inverted
Ensure: Inversion of the input matrix
 1: $size \leftarrow \sqrt{THR\_CNT}$
 2: $L^{-1} \leftarrow I_n$ /* located in global memory */
 3: $A \leftarrow 0_{size \times size}$ /* located in shared memory */
 4: $B \leftarrow 0_{size \times size}$ /* located in shared memory */
 5: for all $blk$ such that $0 \leq blk < MUL\_CNT$ do
 6:   for all $thr$ such that $0 \leq thr < THR\_CNT$ do
 7:     $x\_id \leftarrow thr/size$
 8:     $y\_id \leftarrow thr\%size$
 9:     for $x = size \cdot blk; x < n; x+ = size \cdot MUL\_CNT$ do
10:       for $y = 0; y < n; y+ = size$ do
11:         $val \leftarrow -L^{-1}[x + px, y + py]$
12:         for $zz := 0; zz < y; zz+ = size$ do
13:           $A[px, py] \leftarrow L[px + zz, py + y]$
14:           $B[px, py] \leftarrow L^{-1}[px + x, py + zz]$
15:           __sync
16:           for $i := 0$ to $size - 1$ do
17:             $val \leftarrow val + A[i, py] \cdot B[i, px]$
18:           end for
19:           __sync
20:         end for
21:         $A[py, px] = L[px + y, py + y]$
22:         __sync
23:         for $yy := 0$ to $size - 1$ do
24:           if $yy = py$ then
25:             $B[yy, px] \leftarrow -val$
26:           end if
27:           __sync
28:           $val \leftarrow val + A[py, yy] \cdot B[yy, px]$
29:         end for
30:         $L^{-1}[px + x, py + y] = B[py, px]$
31:         __sync
32:       end for
33:     end for
34:   end for
35: end for
36: return $L^{-1}$
```

6.3 Performance evaluation

In the following sub-sections we present results of our performance evaluations of the different matching algorithms. We start by introducing algorithms to be compared, presenting obtained results for each of them and discussing their pros and cons. At the end of this section we compare performance of all algorithms in the context of different subclasses of graphs. Algorithms under consideration have been evaluated against following 10 groups of graphs:

- Group A: *The University of Florida Sparse Matrix Collection* [47].
- Group B: *A Database of Graphs in Combinatorica Format* [12].
- Group C: *Mathematica's GraphData collection* [40].
- Group D: Collection of regular graphs generated with *High productivity software for complex networks* [46].
- Group E: Collection of bipartite regular graphs generated with *High productivity software for complex networks* [46].
- Group F: Dense random graphs.
- Group G: Dense graphs of different sizes with a general structure presented in the Figure 6.2.
- Group H: Dense graphs of different sizes with a general structure presented in the Figure 6.3.
- Group I: Random cubic graphs generated with *High productivity software for complex networks* [46].
- Group J: Cubic graphs of different sizes with a general structure presented in the Figure 6.4.

Table 6.4 presents general characteristics of all groups of graphs.

All tests have been performed on the following machine:

- 4 cores Intel(R) Xeon(R) CPU E5420, 2.50GHz with 6 MB of cache

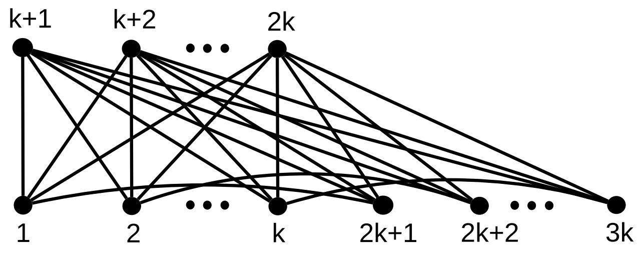

Figure 6.2: General structure of graphs from group G. A graph with $3k$ vertices is presented. Vertex $l \in \{k+1, k+2, \ldots, 2k\}$ is connected with vertices $1, 2, \ldots k$, $2k+1, 2k+2, \ldots, 3k$. In addition vertex $l \in \{1, 2, \ldots, k\}$ is connected with vertex $2k+l$

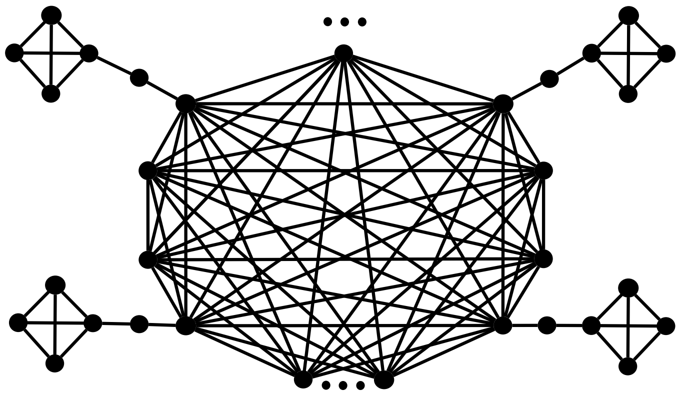

Figure 6.3: General structure of test graphs from group H. For a given n, the graph consists of a clique of size $11n$ connected to n cliques of size 4 by paths of length 2.

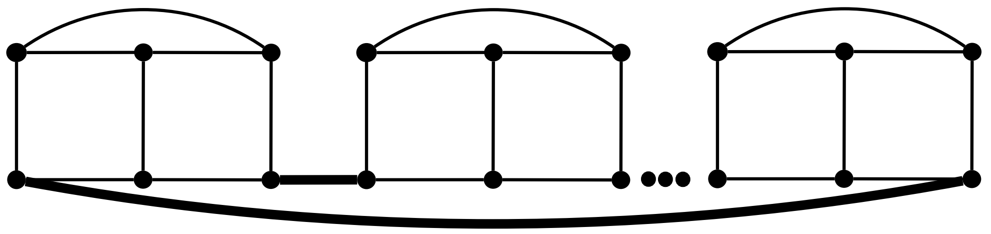

Figure 6.4: General structure of biconnected cubic graphs from group I. The edges in bold are contained in only one matching.

Table 6.4: The general characteristics of graph sets used as test data

characteristic	A	B	C	D	E	F	G	H	I	J
# of graphs	1952	399	3025	229	100	30	25	39	102	49
min. # of vertices	3	2	2	744	2596	256	330	160	10000	6000
max. # of vertices	530428	64960	6877	496790	98762	4096	3159	3200	981314	580602
av. # of vertices	23610	2646	63.99	91789	51827	1587.2	1269	1680	201183	129138
min. # of edges	1	1	1	8676	2596	65	24310	6075	15000	9000
max. # of edges	933343	97440	225226	745185	245400	8.0 mln	2.2 mln	2.4 mln	1.4 mln	870903
av. # of edges	95086	5433	465	217683	122471	812713	510255	858858	301775	193708
min. vertex degree	0.92	1.83	0.69	2.00	2.00	0.50	147	75.93	3.00	3.00
max. vertex degree	291	13.50	330	209	34.00	3932	1404	1512	3.00	3.00
av. vertex degree	10.02	3.65	5.72	9.65	6.99	576.8	564.6	793	3	3
# of regular graphs	30	265	981	229	100	0	0	0	102	49
# of bipartite graphs	1952	91	824	0	100	2	0	0	0	0
min. matching size	1	1	1	372	1298	48	165	80	5000	3000
max. matching size	169399	32479	2380	248395	49381	2048	1579	1600	490657	290301
av. matching size	9292	1278	28.45	45894	25913	782.3	634	840	100591	64569
# of perfect matchings	1078	306	1856	218	100	24	12	39	102	49
graph characteristics after converting to almost-cubic form										
min. # of vertices	3	2	2	6666	2596	256	95590	23520	10000	6000
max. # of vertices	3.7 mln	287518	894079	1.6 mln	565732	32.1 mln	8.8 mln	9.6 mln	981314	580602
av. # of vertices	288799	9215	1593	417297	254462	3.2 mln	2.0 mln	3.4 mln	201183	129138
min. # of edges	1	1	1	9999	2596	65	119570	29435	15000	9000
max. # of edges	4.6 mln	363836	1.1 mln	2.0 mln	712042	40.2 mln	11.0 mln	12.0 mln	1.4 mln	870903
av. # of edges	360274	12002	1995	543191	325106	4054291.6	2.5 mln	4.2 mln	301775	193708

- 4 GB RAM memory
- Nvidia GeForce 9800 GX 2, 1 GB

6.3.1 Boost matching algorithm with an empty initial matching

The first algorithm that we compare is the Boost library [1] implementation of Edmond's approach [17] to constructing maximum matching in general graphs. Edmond's approach has been introduced in Section 3.1. The Boost library algorithm prior to searching for augmenting paths computes the initial matching using some fast heuristics. Such an approach greatly increases the performance of the entire algorithm, as smaller number of augmenting paths need to be found.

In the current section we present the analysis of the Boost matching algorithm with disabled heuristics (the initial matching is empty). We will refer to this algorithm as a Boost matching algorithm with an empty initial matching or *boost_empty* for short. Figures 6.5 and 6.6 present, respectively, execution time of the Boost algorithm as a function of the number of vertices and edges in the test graphs. Both charts have logarithmic scale to visualize in more detail the spread of execution time for different input graphs. The pessimistic execution time of the Boost algorithm, as claimed by the Boost library documentation, is $O(m\,n\,\alpha(m,n))$. After analyzing the results presented in the Figure 6.5 it turns out that for the majority of graphs the Boost algorithm requires $\theta(n^2)$ time, where n stands for the number of vertices. This behavior is expected, as most of the graphs from the test set are sparse, so $m = O(n)$ and $O(m\,n\,\alpha(m,n))$ becomes $O(n^2\alpha(m,n))$, which from practical point of view is equal to $O(n^2)$. The algorithm is also doing pretty well for many dense graphs — runs in time $o(n^3)$. This is also expected, as it is easier to find short augmenting paths in dense graphs. Only for graphs from groups G and H the worst case performance of $O(n^3)$ is hit. When analyzing execution time of the algorithm as a function of the number of edges, it once again turns out that for the majority of graphs computation of the maximum matching takes quadratic time. Again, this stems from the fact that most graphs are sparse ($O(m) = O(n)$).

6.3.2 Boost initial matching heuristics

Maximum matching algorithm from the Boost library comes together with two relatively simple heuristics for constructing an initial matching. The first heuristic is called *greedy_matching*. It starts with an empty matching M and scans the set of all edges of the graph in an arbitrary order. Once an edge $e \in E$ incident to not yet matched vertices is identified, M is extended with e. This algorithm runs in $O(m)$ time. As the matching M computed with this approach is maximal (it can't be extended by adding any edge $e \in E$), according to the Lemma 6.2, *greedy_matching* heuristic is an 1/2 approximation of the maximum matching problem.

Lemma 6.2. *Let $G = (V, E)$ be an arbitrary graph, M — a* maximum matching *of G and M' — a maximal matching of G. M' is a $1/2$ approximation of M ($2|M'| \geq |M|$).*

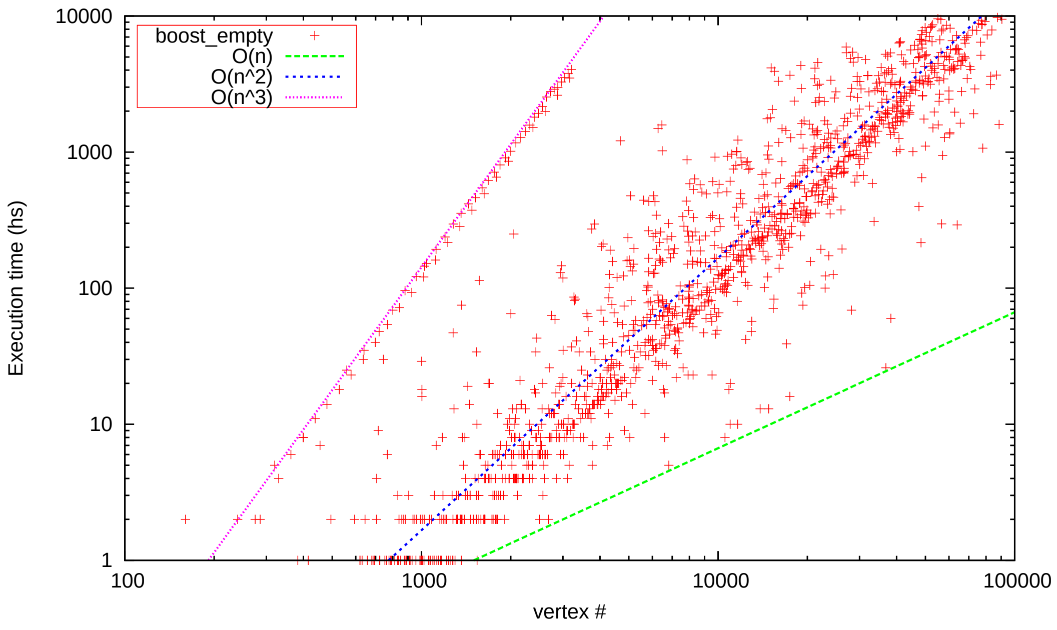

Figure 6.5: Execution time of the Boost maximum matching algorithm with the empty initial matching presented as a function of the number of vertices in the test graphs.

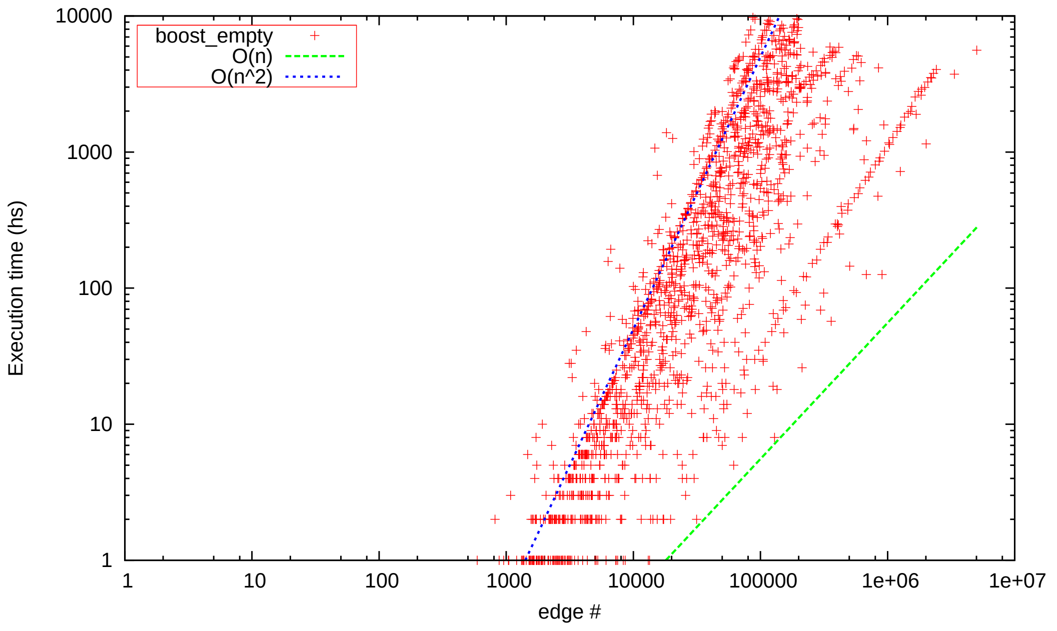

Figure 6.6: Execution time of the Boost maximum matching algorithm with the empty initial matching presented as a function of the number of vertices in the test graphs.

Proof. Consider $G' = G[V, M \otimes M']$. Every vertex of G' is of degree not greater than 2, hence G' is a collection of cycles C and paths P. Each cycle and path consists of alternative edges from M and M', hence each cycle $c \in C$ is of even length and consists of the same number of edges from M and M'. Each even-length path $p \in P$ also consists of the same number of edges from M and M'. The only case when the number of edges from sets M and M' differs are paths of odd length. Path of length $2k + 1$ consists of k edges from one of the matchings and $k + 1$ edges from the other. As there are no paths of length 1 (otherwise it would mean two adjacent vertices of G are not matched in one of M or M' contradicting the maximality of M or M'), the worst case scenario are paths of length 3 with 2 edges from M and 1 edge from M'. Hence $2|M'| \geq |M|$. $\square$

The second heuristic — *extra_greedy_matching* — also greedily extends the matching M being constructed with consecutively analyzed edges. The difference is that edges are not processed in an arbitrary order. *Extra_greedy_matching* prior to constructing the matching sorts all edges of the graph in increasing order with respect to degrees of their end vertices. The intuition behind such an approach is that it is easier to match a vertex with a high degree than a vertex with a low degree. By matching vertices of low degree first we increase our chances of constructing a matching which is a better approximation of the maximum matching. This heuristic runs in $O(n + m \log m)$ time (as sorting of edges occurs) and it is also a $1/2$ approximation of the maximum matching.

In addition to the two original Boost heuristics we have also introduced the third heuristic. We call it *extra2_greedy_matching*, as it is quite similar to the second Boost heuristic. It processes vertices of the graph in the increasing order of their degrees and extends the matching being constructed with not yet matched edges adjacent to vertices of lowest possible degrees. Once an edge $u - v$ is added to the matching, vertices u and v are removed from the graph and the degrees of vertices adjacent to them are updated. This algorithm is implemented with the clever use of bi-direction lists, hence it's execution time is $O(n + m)$. Figures 6.7, 6.9 and 6.11 present execution time of all three heuristics as a function of the number of vertices in the test graphs. Figures 6.8, 6.10 and 6.12 present execution times of heuristics as a function of the number of edges in the test graphs. Figure 6.13 presents an average dependence between the number of edges in the graphs and the execution time.

From the Figure 6.13 it is clear that the fastest heuristic is the *greedy_matching*. The slowest, on the other hand, is *extra_greedy_matching*. This observation is consistent with the theoretical complexity analysis of these three algorithms. *Greedy_matching* is linear and the simplest to implement, hence it turns out to be the fastest in practice. Slightly slower is the *extra2_greedy_matching*, as some additional operations on bidirectional lists are performed by this algorithm. The slowest one is *extra_greedy_matching*, which sorts edges of the graph with $O(m \log m)$ sorting algorithm. Which of these three heuristics works the best in connection with Boost implementation of Edmond's augmenting paths approach? Not only does the answer depend on the performance of the heuristic, but also on the size of the initial matching constructed. Figure 6.14 presents the sizes of matchings constructed by

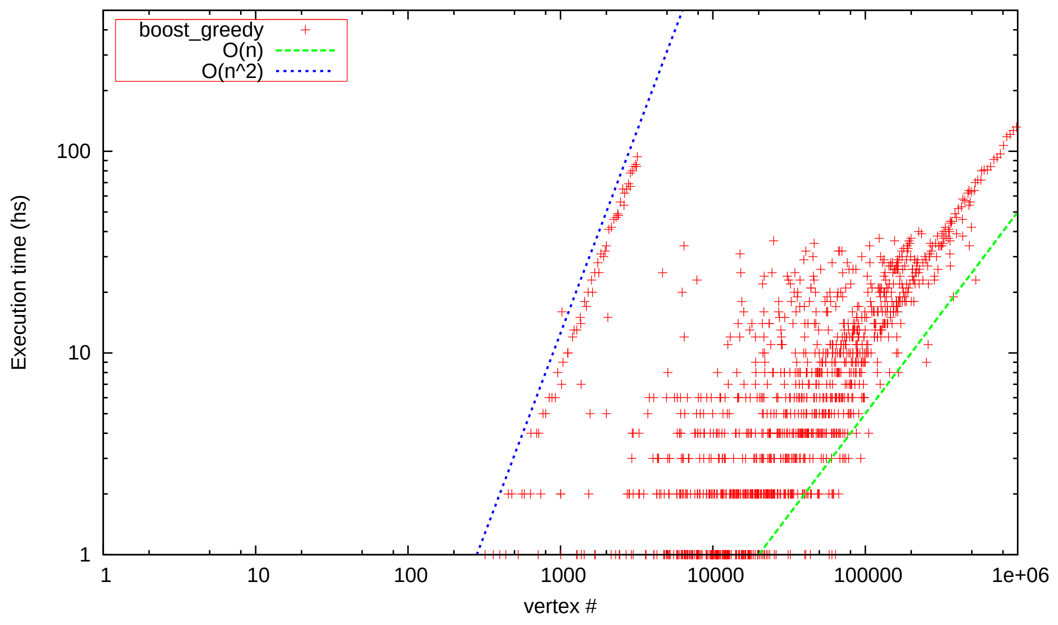

Figure 6.7: Execution time of *greedy_matching* heuristic presented as a function of the number of vertices.

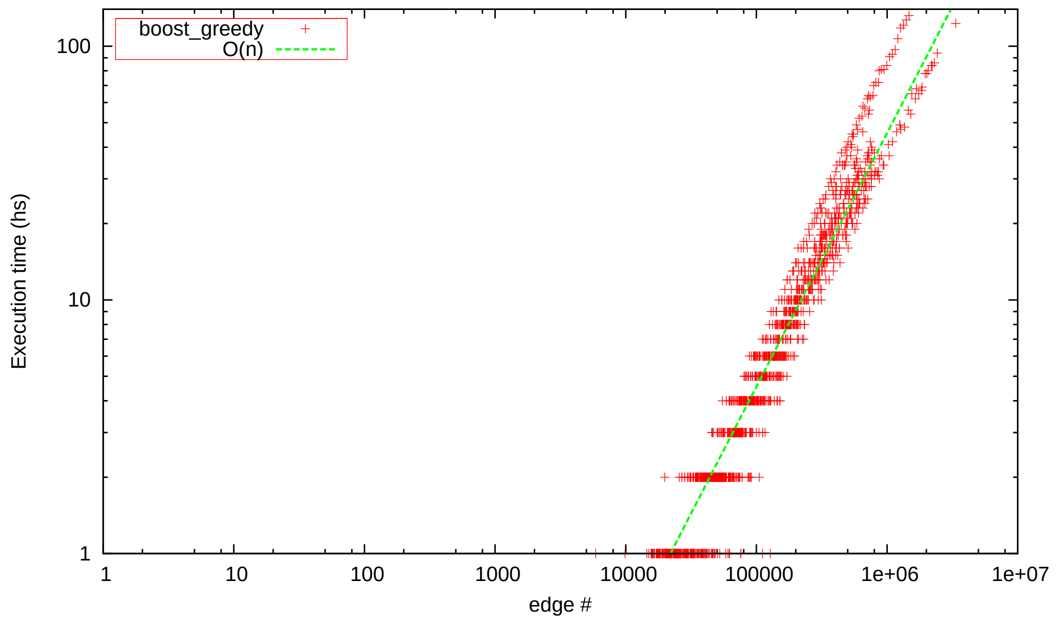

Figure 6.8: Execution time of *greedy_matching* heuristic presented as a function of the number of edges.

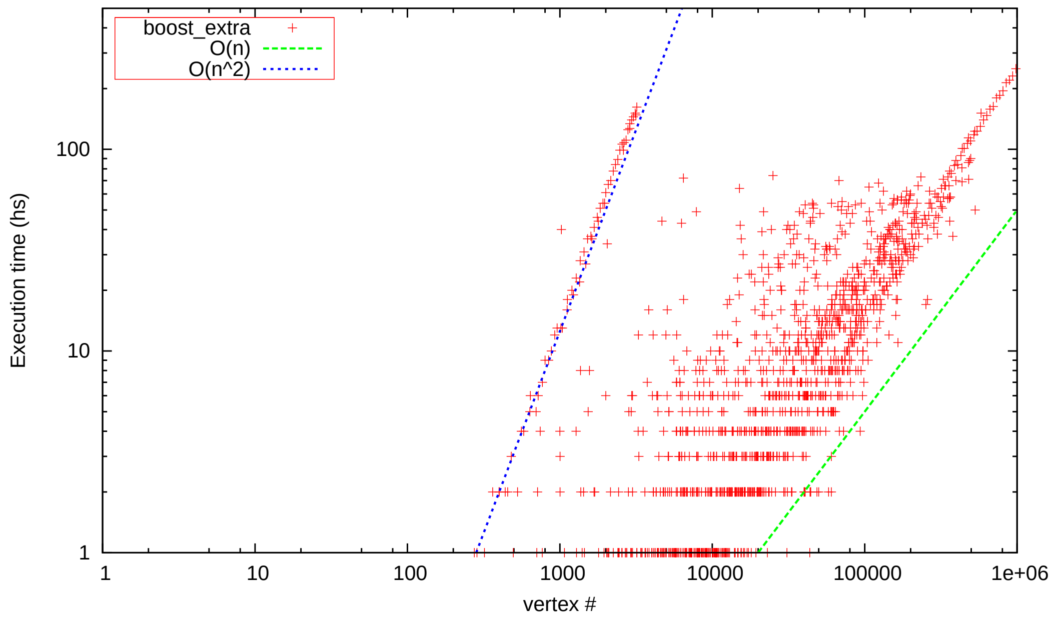

Figure 6.9: Execution time of *extra_greedy_matching* heuristic presented as a function of the number of vertices.

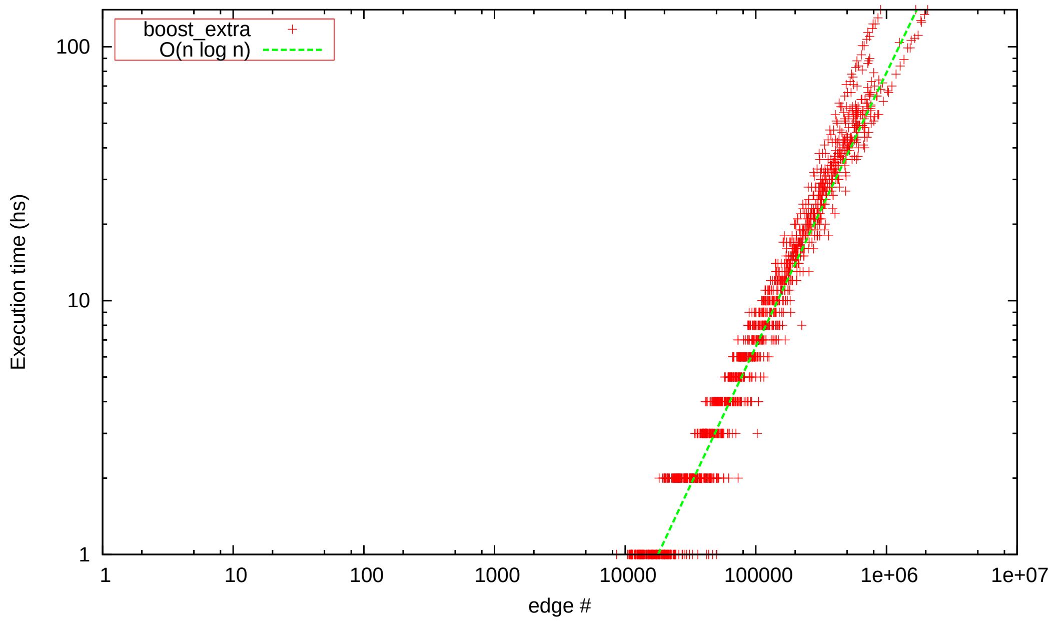

Figure 6.10: Execution time of *extra_greedy_matching* heuristic presented as a function of the number of edges.

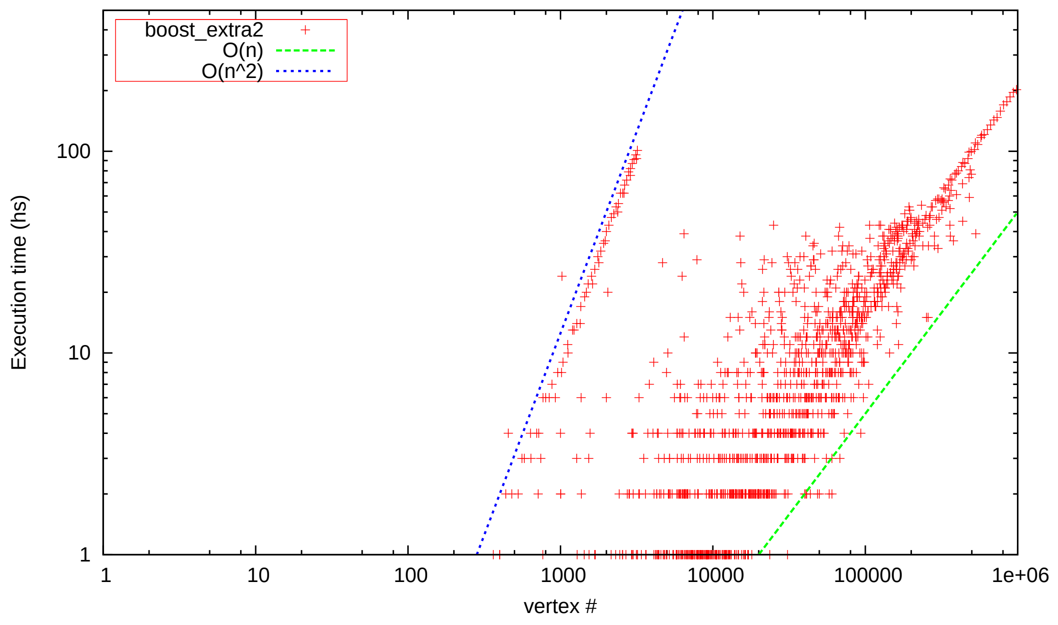

Figure 6.11: Execution time of *extra2_greedy_matching* heuristic presented as a function of the number of vertices.

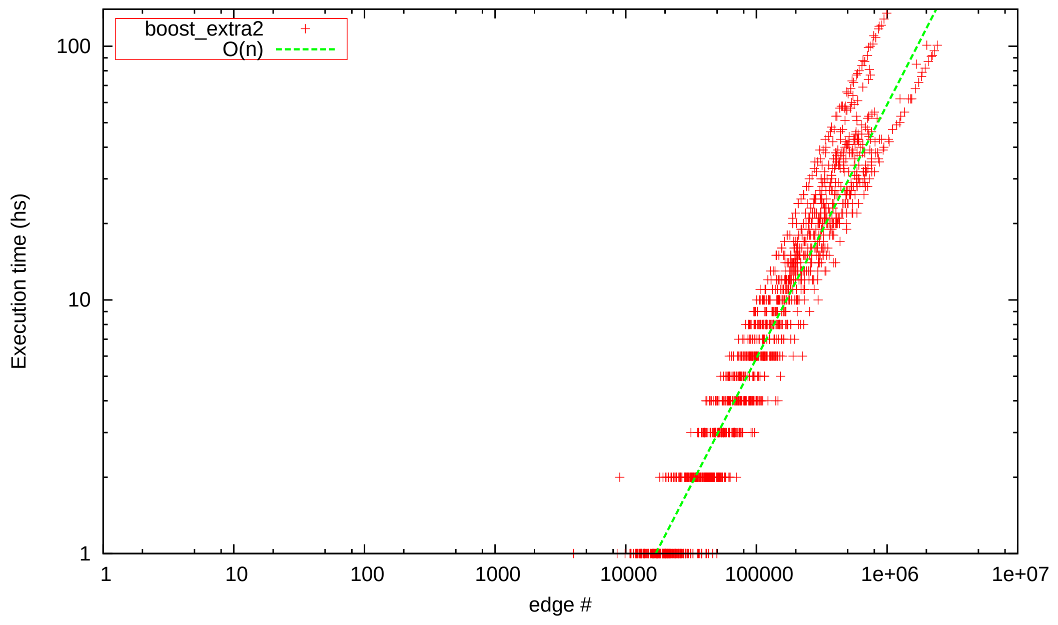

Figure 6.12: Execution time of *extra2_greedy_matching* heuristic presented as a function of the number of edges.

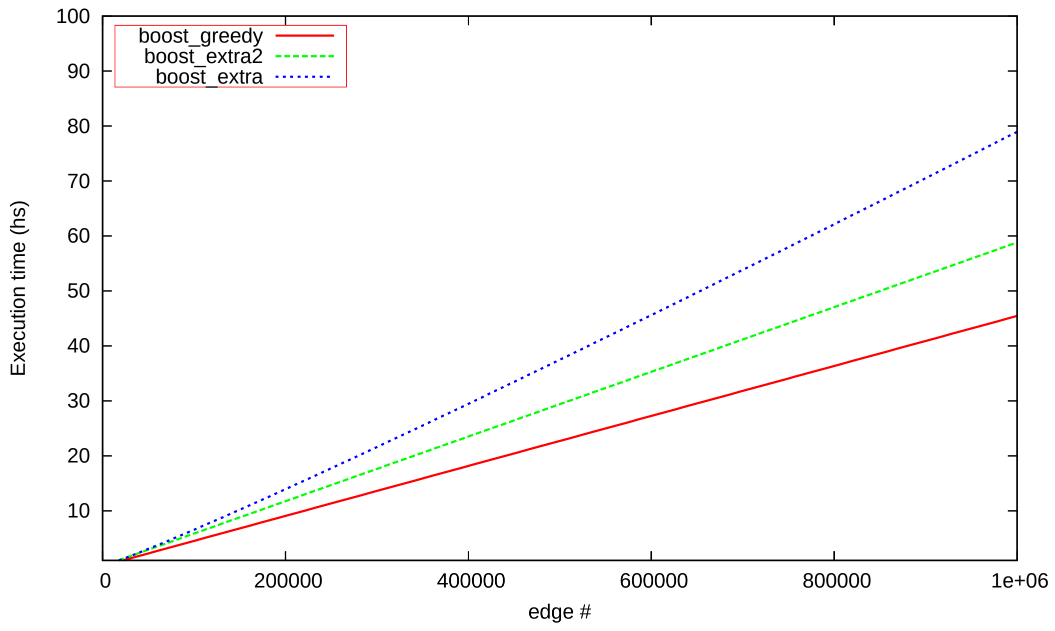

Figure 6.13: Average execution time of heuristics presented as a function of the number of edges.

the three heuristics as the percentage of the maximum matching size. After analyzing Figure 6.14 it is clear that *extra2_greedy_matching* gives much better results for majority of input graphs. The average sizes of the matchings constructed by *greedy_matching*, *extra_greedy_matching* and *extra2_greedy_matching* are respectively 92.8%, 94.8% and 99.8% of the maximum matching size. Figures 6.15 and 6.16 present execution time of the four Boost-based matching algorithms (Edmond's matching algorithm with an empty initial matching and with initial matchings computed by the three heuristics). Asymptotic complexity of the worst cases and average cases of all four algorithms is the same — $O(n^3)$ and $O(n^2)$ respectively. The use of heuristics, however, greatly accelerates the matching algorithm. The average speedup obtained with *greedy_matching* is 31.7, with *extra_greedy_matching* — 25, while *extra2_greedy_matching* accelerates matching algorithm over 104 times. *Extra_greedy_matching* heuristic gives slightly better results than *greedy_matching*, but as it executes longer the overall result is worse. In the following sections we are going to consider matching heuristics only in the context of Edmond's matching approach, so from now on the name of heuristic will refer to the Boost maximum matching algorithm with an initial matching computed by that heuristic. The Boost matching algorithm with an empty initial matching will be called *boost_matching_empty*, for short.

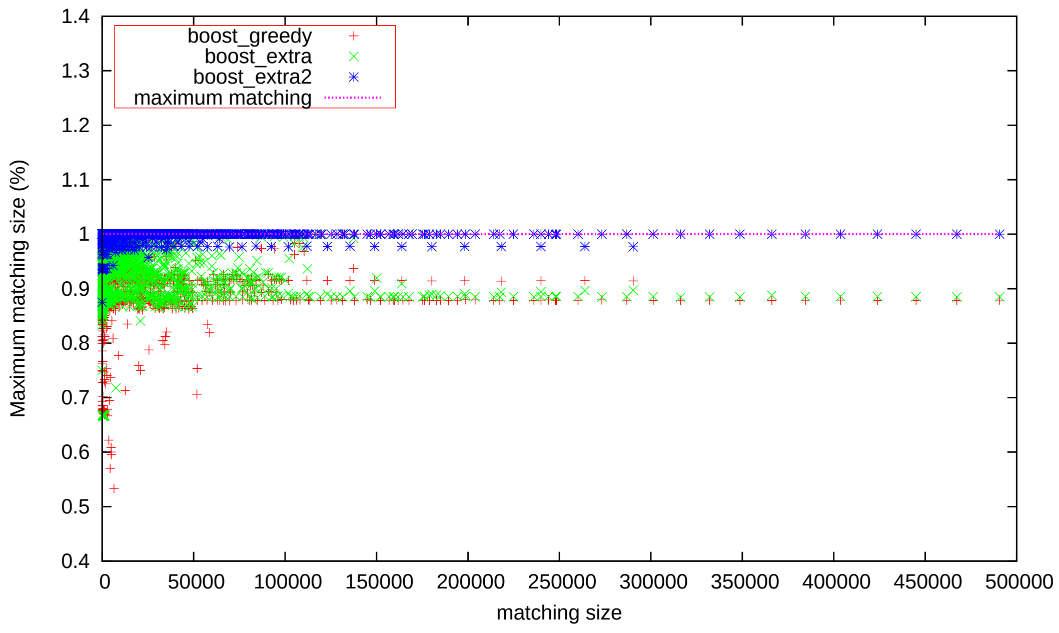

Figure 6.14: The size of the initial matching constructed by the heuristics presented as a percentage of the maximum matching size.

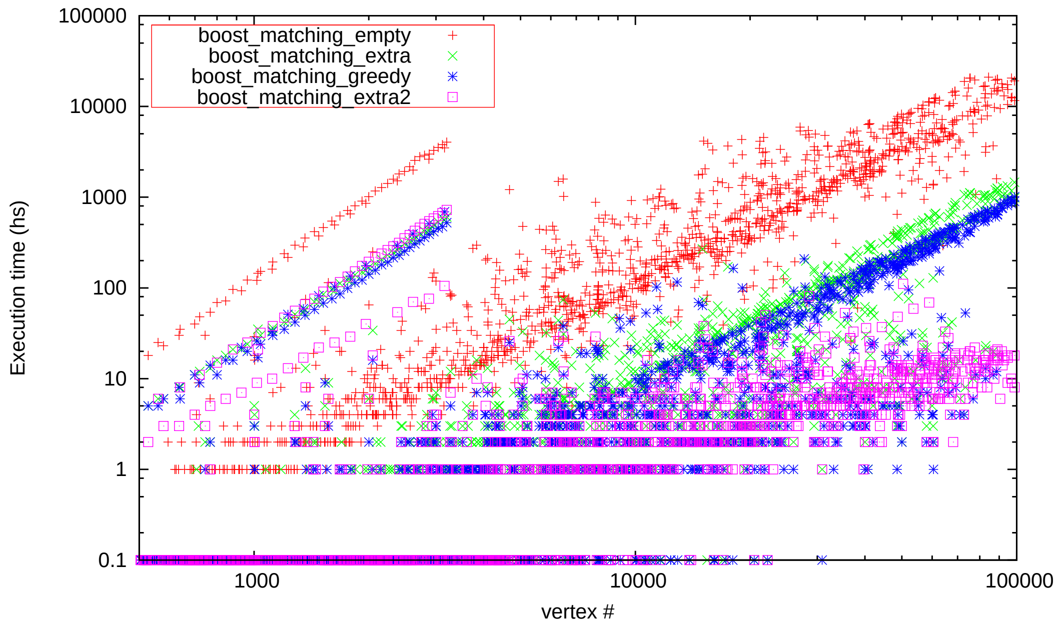

Figure 6.15: Execution time of the Boost matching algorithms presented as a function of the number of vertices in the graphs.

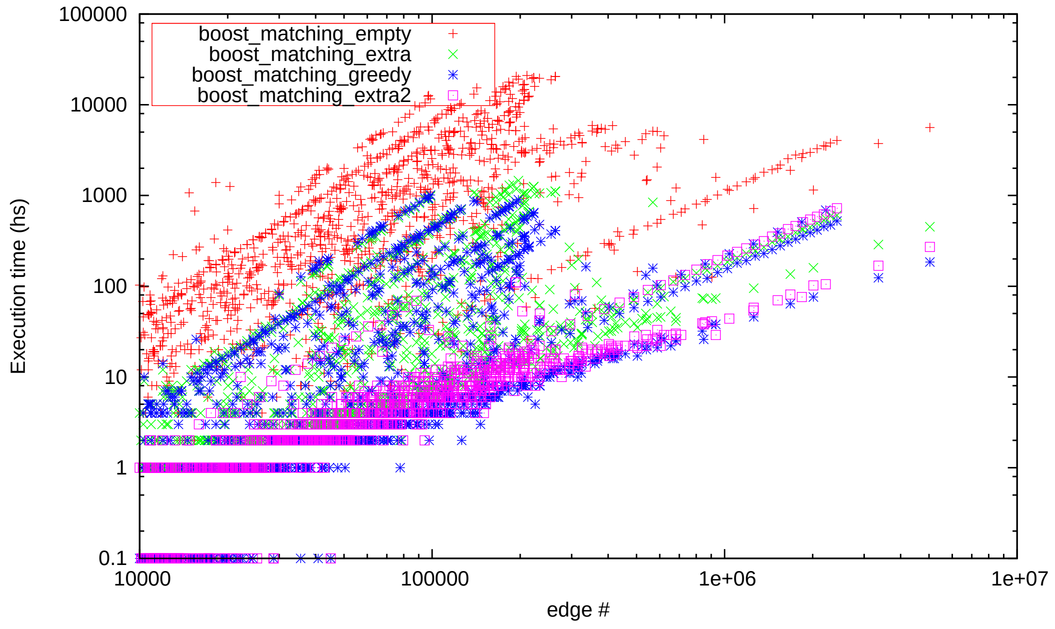

Figure 6.16: Execution time of the Boost matching algorithms presented as a function of the number of edges in the graphs.

6.3.3 Perfect matching via boolean satisfiability

Another algorithm that we compare is an algorithm which reduces the matching problem to an instance of the boolean satisfiability problem. The reduction was presented in Section 6.1. As this algorithm is only able to construct a perfect matching of a graph, we analyze it separately for graphs with and without a perfect matching. Our implementation of this algorithm utilizes MiniSat [45], a minimalistic, open-source SAT solver, developed to help researchers and developers alike to get started on SAT.

Solving the SAT problem in general case is NP-hard, however it turns out relatively efficient to use a SAT solver to verify whether a graph contains a perfect matching. For 92% of test graphs without a perfect matching SAT solver managed to come up with the conclusion, that a graph does not contain a perfect matching within a reasonable time (60 second time-limit). Among those 92% of graphs, SAT solver was faster than the Boost algorithm with an empty initial matching for 86% of graphs. The cumulative execution time against all graphs was over 13 times shorter than in case of the Boost algorithm. The maximal speed-up obtained with a SAT-based algorithm in comparison with Edmond's matching algorithm with an empty initial matching is over 2000 times, while the maximal slowdown — almost 27 times.

When compared to the Boost algorithms with an initial heuristic enabled — *greedy_matching*, *extra_greedy_matching* and *extra2_greedy_matching*, the results are less impressive. The speedup was obtained for only 29%, 32.7% and 18% of graphs respectively, while the cumulative execution time was 2.97, 2.67 and 18.34

times longer.

There is no well defined structure of graphs for which SAT solver is faster than the Boost algorithms. It is able to outperform Boost only when it is lucky to find an easy proof of non-existence of a perfect matching in an input graph (like an isolated vertex or odd-size connected component). Figures 6.17 and 6.18 present execution time comparison of a SAT-based algorithm and selected Boost algorithms against graphs without a perfect matching.

The second group of graphs that are considered for SAT-based matching algorithm are the graphs containing perfect matchings. SAT-based algorithm, within 60 seconds time-limit, managed to compute a perfect matching for 91% of those graphs (*empty_matching* computed result for 84.7% of graphs, *greedy_matching* for 98.3%, *extra_greedy_matching* for 98.2% and *extra2_greedy_matching* for 99% of graphs). Figures 6.19 and 6.20 present execution time comparison of SAT-based algorithm with the Boost algorithms against graphs with perfect matchings. SAT-based algorithm obtains speedup compared to the Boost algorithms for 85%, 55%, 50% and 21% of graphs respectively. The cumulative execution time of the SAT-based algorithm is almost 14 times shorter when compared to Boost algorithm with an empty initial matching and respectively 1.26, 1.96 and 37 times slower when compared to Boost algorithms with heuristics enabled. The maximal speedup of 1579, 135, 140 and 2 times respectively is obtained for a regular bipartite graphs of degree 2. There are graphs, however, for which SAT algorithm runs much slower than Edmond's algorithm. When compared to *empty_matching* the slowdown is 1028 times, while when

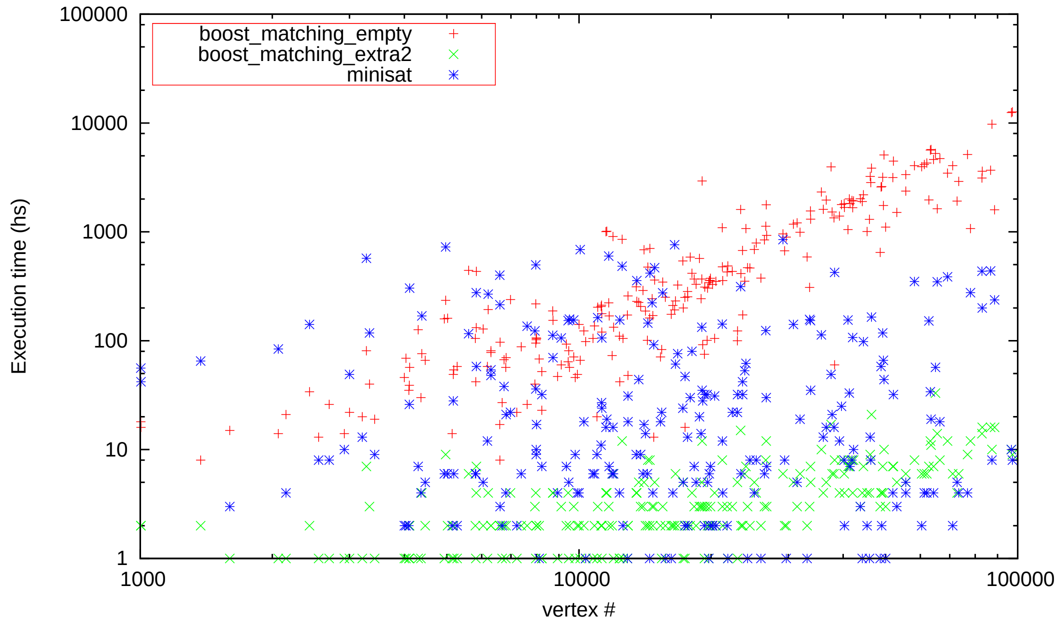

Figure 6.17: Execution time of the SAT-based algorithm and the Boost matching algorithms for graphs without a perfect matching presented as a function of the number of vertices.

compared to *extra2_greedy_mathcing* — 7197 times. Is it possible to characterize the class of graphs for which SAT algorithm does well in comparison with Edmond's approach? After analyzing the structure of the input graphs and the speedup obtained it turns out that SAT-based algorithm does well for regular and bipartite graphs of low degree. When limited to regular biconnected graphs SAT algorithm is doing very well. Figure 6.21 presents a comparison. The cumulative execution time of *extra_greedy_matching* algorithm is over 3.5 times longer than the SAT-based algorithm. If matching heuristic is disabled the cumulative execution time of Boost algorithm becomes over 89 times longer. *Extra2_greedy_matching* is still performing much better and runs 6 times faster than SAT-based algorithm.

Once a set of considered graphs is further limited to cubic graphs SAT-based algorithm speed-ups over 85% of cases for *greedy_matching* and *extra_greedy_matching* algorithms. Compared to *extra2_greedy_matching* the speedup is obtained for 82% of cases. The cumulative execution time is now over 170 times shorter when compared to *empty_matching* and around 12 times shorter compared to *greedy_matching* and *extra_greedy_matching*. The execution time of *extra2_greedy_matching* is still around 3 times faster. How is it possible that total execution time of the SAT-based algorithm is 3 times longer than the *extra2_greedy_matching* and at the same time SAT-based algorithm accelerates 82% of cases? For most graphs SAT-based algorithm runs faster than all Boost algorithms, however there are graphs for which the execution time is much longer. The worst case scenario from the Figure 6.22 presents 290 times slow-down. Figure 6.23 presents execution time of the considered algorithms

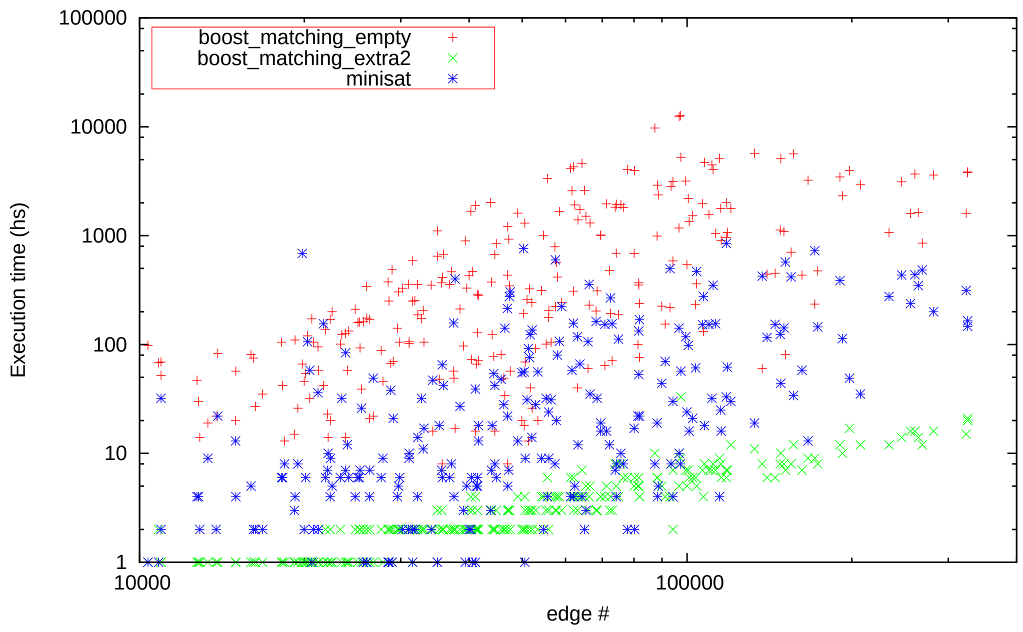

Figure 6.18: Execution time of the SAT-based algorithm and the Boost matching algorithms for graphs without a perfect matching presented as a function of the number of edges.

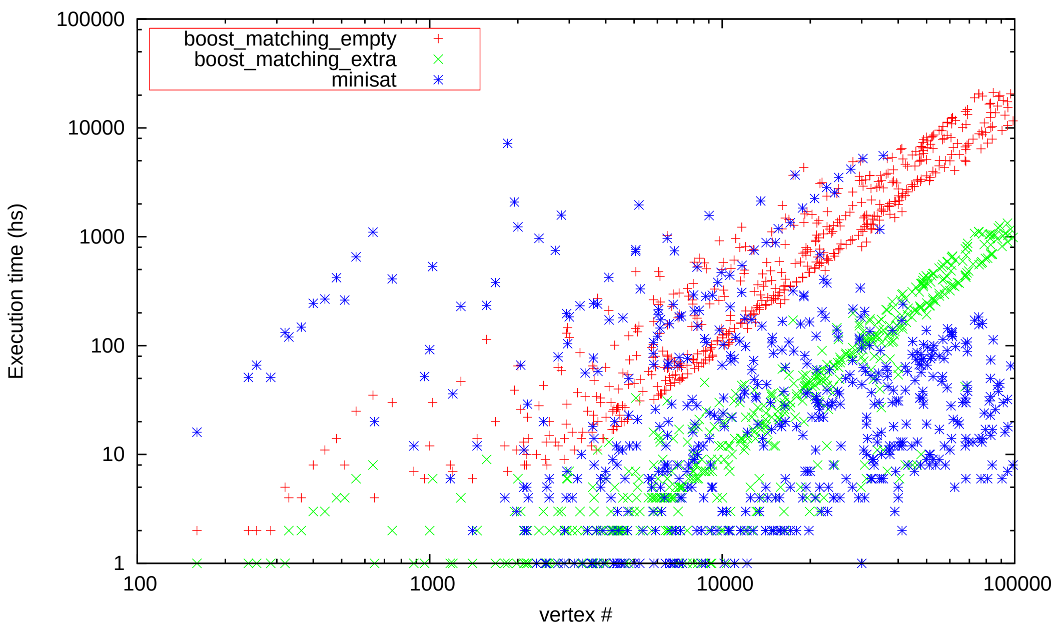

Figure 6.19: Execution time of the SAT-based algorithm and the Boost matching algorithms for graphs containing a perfect matching presented as a function of the number of vertices in graphs.

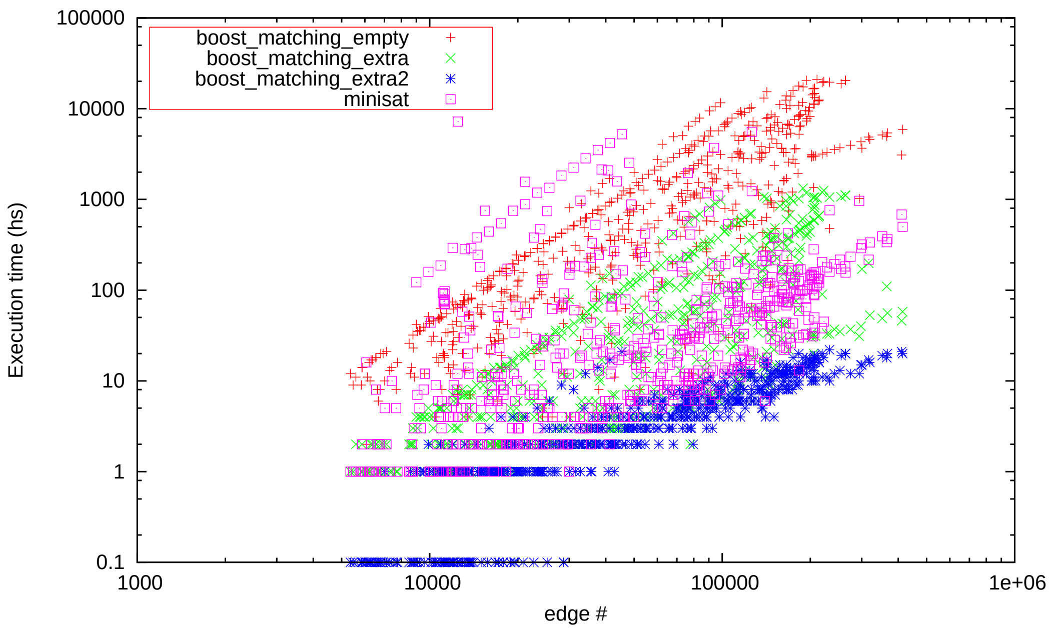

Figure 6.20: Execution time of the SAT-based algorithm and the Boost matching algorithms for graphs with a perfect matching presented as a function of the number of edges in graphs.

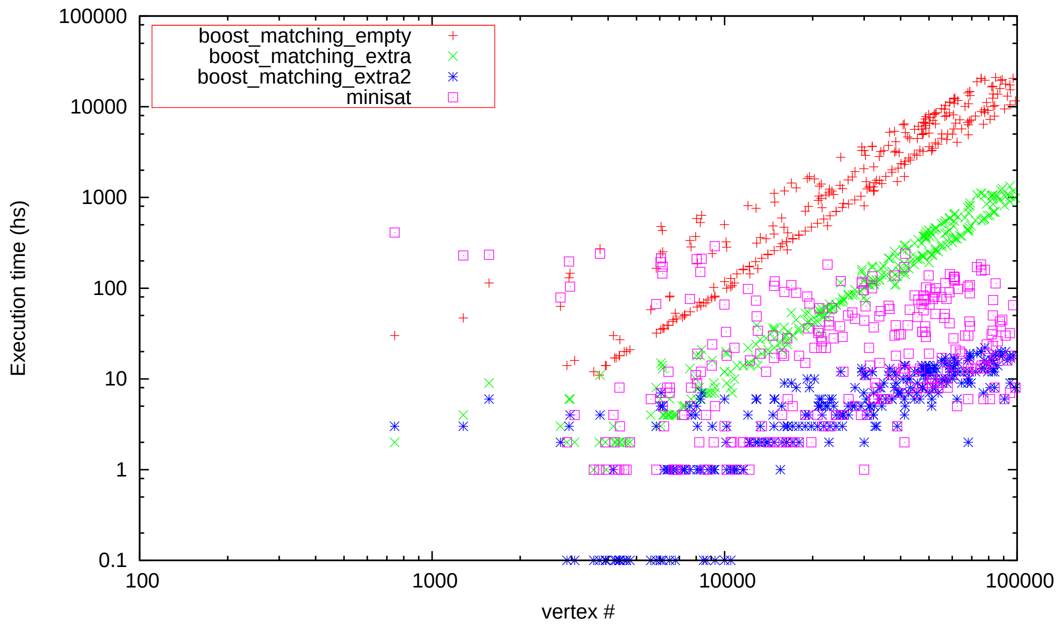

Figure 6.21: Execution time of the SAT-based algorithm and the Boost matching algorithms for regular bipartite graphs with a perfect matching presented as a function of the number of vertices. Input graphs do not include graphs from Group J.

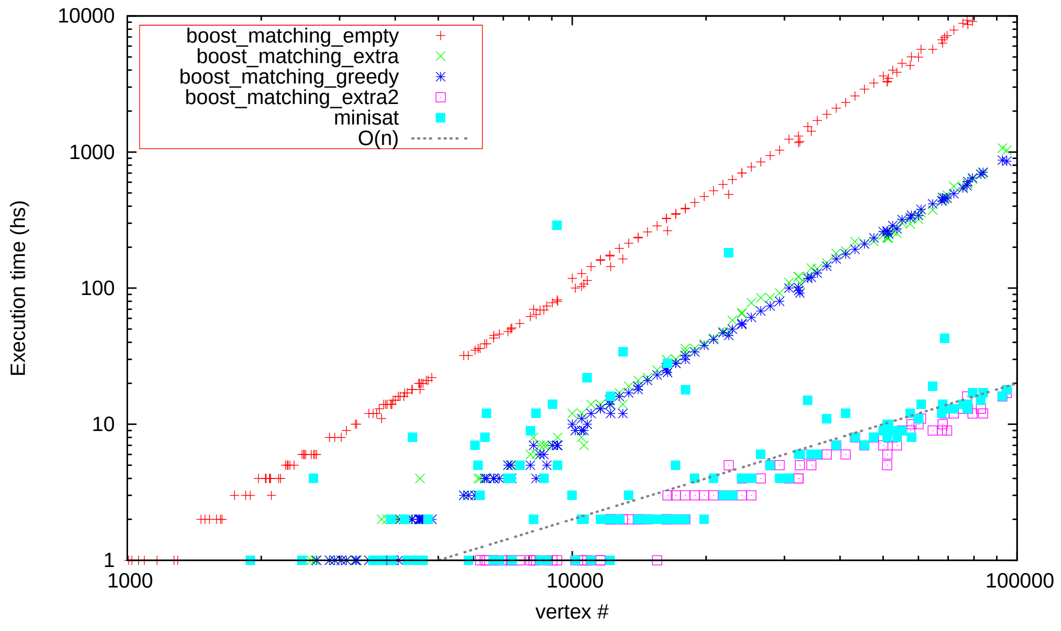

Figure 6.22: Execution time of the SAT-based algorithm and the Boost matching algorithms for biconnected cubic graphs presented as a function of the number of vertices. Input graphs do not include graphs from Group J.

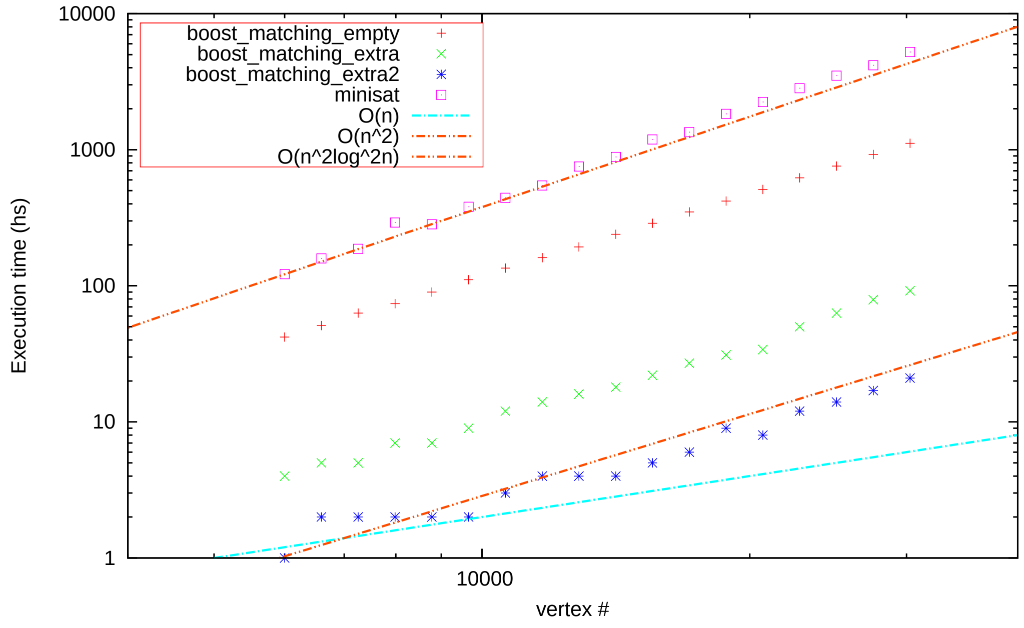

Figure 6.23: Execution time of the SAT-based algorithm and the Boost matching algorithms for graphs from group J presented as a function of the number of vertices in graphs.

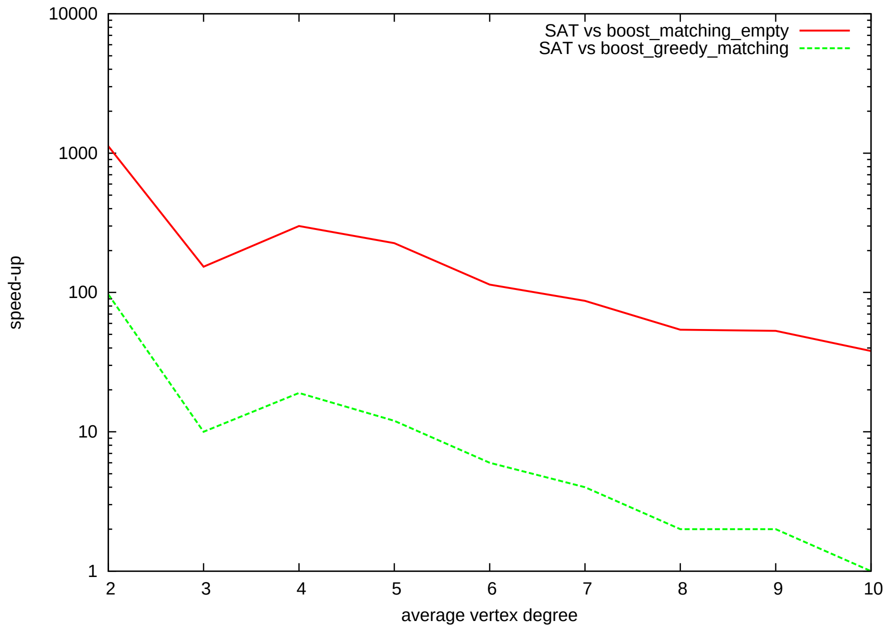

Figure 6.24: Average speed-up obtained by the SAT-based algorithm compared to the Boost matching algorithm as a function of average vertex degree of graphs.

against graphs from group J. For these graphs all Edmond's algorithms run in $O(n^2)$ time, while SAT-based algorithm runs in $O(n^2 \log^2 n)$.

The size of the matching problem converted to the SAT format depends in square on the average vertex degree of an input graph. For dense graphs with $\theta(n)$ vertices and $\theta(n^2)$ edges the input graph turns into $\theta(n^3)$-size SAT problem. The intuition suggests that once an average degree of vertices in a graph increases, the chance that SAT-based algorithm provides speed-up vanishes. The results of the experiments confirm this conjecture. Figure 6.24 presents the speedup obtained by SAT-based matching algorithm for regular biconnected graphs as a function of the degree of graphs.

6.3.4 GPU-based algebraic matching algorithm

The last but not least algorithm to be analyzed is a parallel maximum matching algorithm on GPU. This algorithm was presented in section 6.2. Its execution time depends on the number of vertices in a graph and the size of the maximum matching rather than the number of edges. Figures 6.25 and 6.26 present, respectively, execution time of this algorithm as a function of the number of vertices and edges. The theoretical cubic dependence between the number of vertices and execution time is confirmed in practice. This dependence is even more visible when graphs under consideration are limited to those with perfect matchings — see Figure 6.27. At the same time there is no dependence between the number of edges and the execution time (the scatter ranges from $O(m)$ to $O(m^3)$).

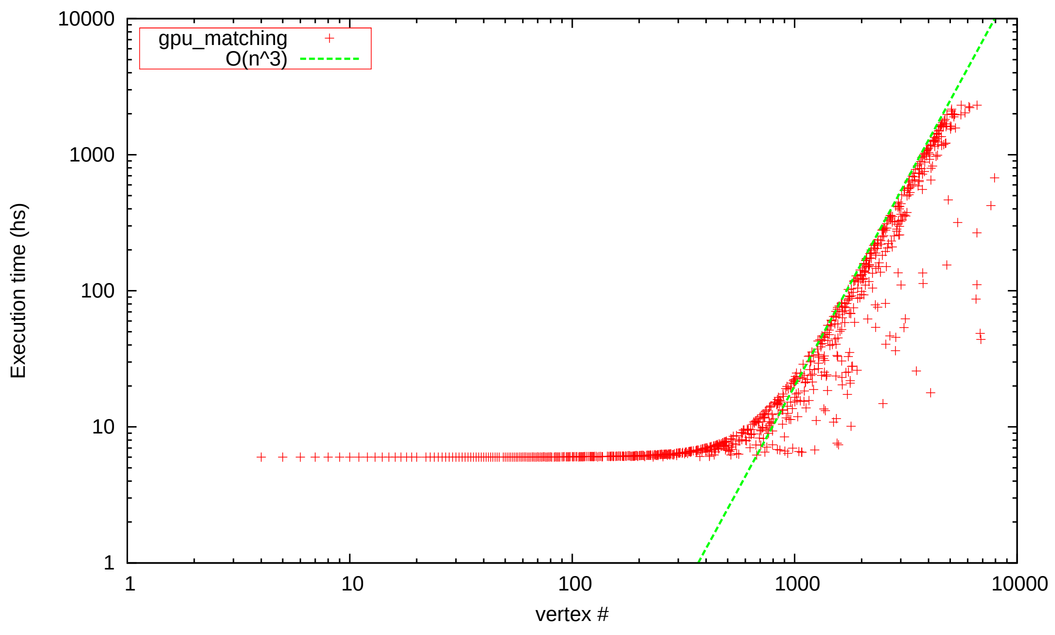

Figure 6.25: Execution time of the GPU-based algebraic matching algorithm presented as a function of the number of vertices.

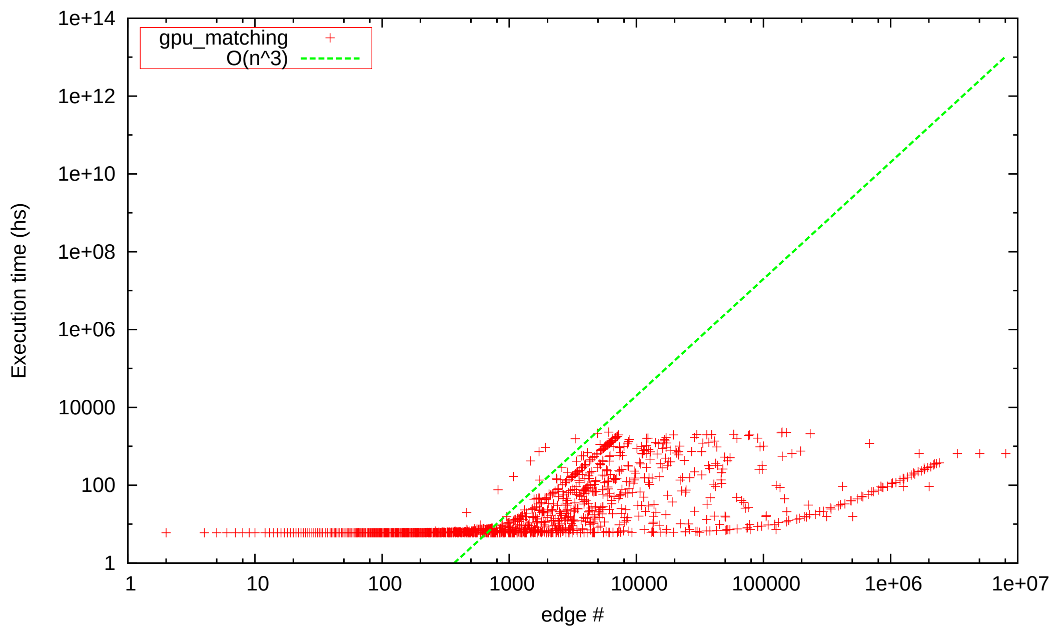

Figure 6.26: Execution time of the GPU-based algebraic matching algorithm presented as a function of the number of edges

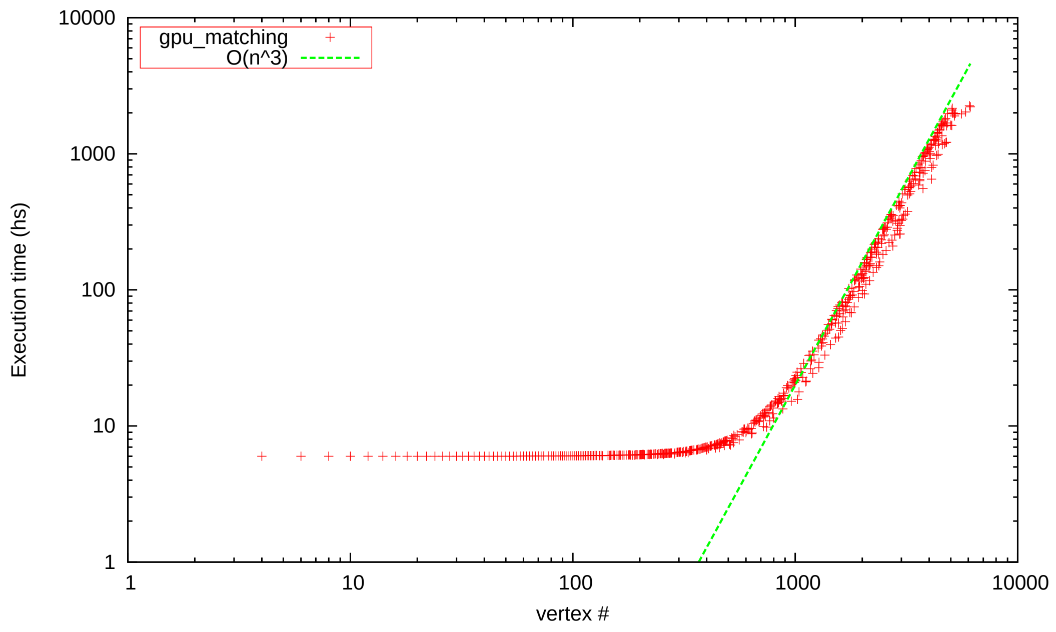

Figure 6.27: Execution time of the GPU-based algebraic matching algorithm against graphs with a perfect matching presented as a function of the number of vertices

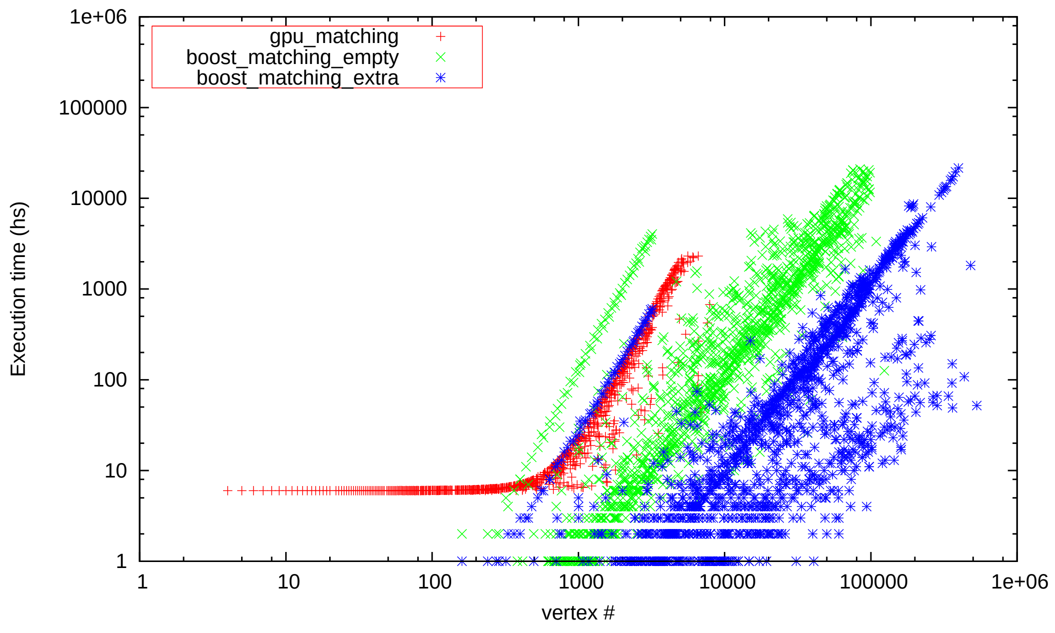

Figure 6.28: Comparison of the GPU-based algebraic matching algorithm and Boost algorithms against all test graphs presented as a function of the number of vertices

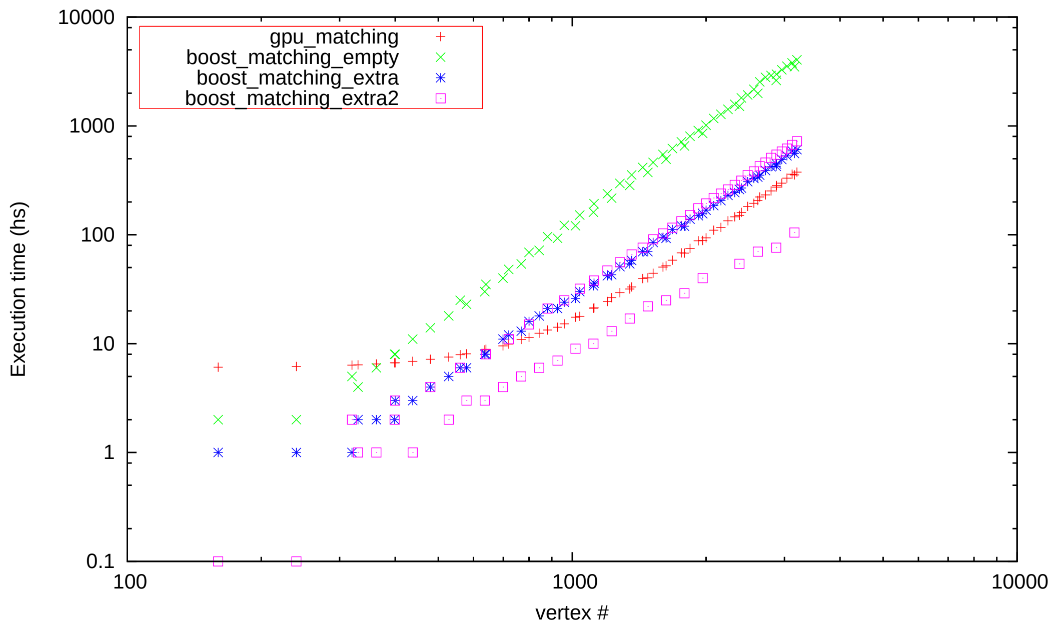

Figure 6.29: Comparison of the GPU-based algebraic matching algorithm and the Boost algorithms against graphs from groups G and H. In case of graphs from group G, GPU-based algorithm outperforms original Boost algorithms (both with and without initial heuristics), while it is still slower than *extra2_greedy_matching*. In case of graphs from group H, GPU-based algorithms outperforms all algorithms.

Figure 6.28 compares execution time of the GPU-based algebraic algorithm against three versions of Edmond's algorithm. It is not surprising that in general the Boost algorithms outperform matrix-based approach. As Figure 6.5 presents *empty_matching* algorithm runs for majority of cases in $O(n^2)$ time, while matrix-based approach is always $\theta(n^3)$. Parallel implementation of the algebraic algorithm is much faster from the sequential counterpart, however the number of processors available in nowadays GPU units is not polynomial in the number of graphs' vertices. In case of huge sparse graphs it is not possible to outperform the Boost sequential algorithms, which are asymptotically faster. As Figure 6.29 presents, however, there are graphs for which GPU-based algorithm outperforms all Boost-based algorithms.

6.4 Summary

Figure 6.30 presents asymptotic average case performance of analyzed matching algorithms in general case. Algebraic matching algorithm runs in $O(n^3)$, while the rest of algorithms in $O(n^2)$. The fastest algorithm in average case is *extra2_greedy_matching*. The worst case scenarios are presented in the Figure 6.31. This time all algorithm have the running time of $O(n^3)$. The order of Boost algorithms didn't change. The slowest is the *empty_matching*, while the fastest — *extra2_greedy_matching*. This time, however, GPU-based algebraic matching algorithm becomes the fastest.

When nothing is known about the structure of the input graphs the safest approach is to apply the classical matching algorithm based on Edmond's approach utilizing a good heuristic for computing initial matching. When, on the other hand, it is known that input graphs are dense and a stable computation time is required (for instance it might be crucial for some on-line problems to compute a matching within a given limited time) GPU-based algebraic algorithms might be useful. For dense graphs our implementation of the algebraic algorithm provides a speedup of only few times compared to *extra2_greedy_matching*, but highly tuned implementations should provide much more impressive results. Also, as we are witnessing a continues increase of GPU units performance, it is an additional source of speedup in the future. The algebraic algorithm can also be distributed to a number of GPU units.

Figures 6.32 and 6.33 present, respectively, asymptotic average and worst case performances of the matching algorithms against biconnected cubic graphs. This time algebraic algorithm is not presented, as it has no chances of being useful in case of sparse graphs. Instead, SAT-based algorithm and two specialized algorithms for biconnected cubic graphs are presented — Biedl et al. [6] $O(n \log^4 n)$ algorithm and our new matching algorithm presented in section 4.3. In the average case the fastest are *extra2_greedy_matching* and SAT-based algorithms. SAT-based algorithm is slightly slower, but both algorithms run in linear time. In case of SAT-based algorithm linear execution time can be easily explained. Each edge of biconnected cubic graph is allowed, hence once a SAT-based algorithm selects the first edge to the perfect matching being constructed, it never needs to change this decision. Removal of the two matched vertices from biconnected cubic graph generates four vertices with degree 2. As biconnected cubic graphs are believed to contain exponentially many perfect

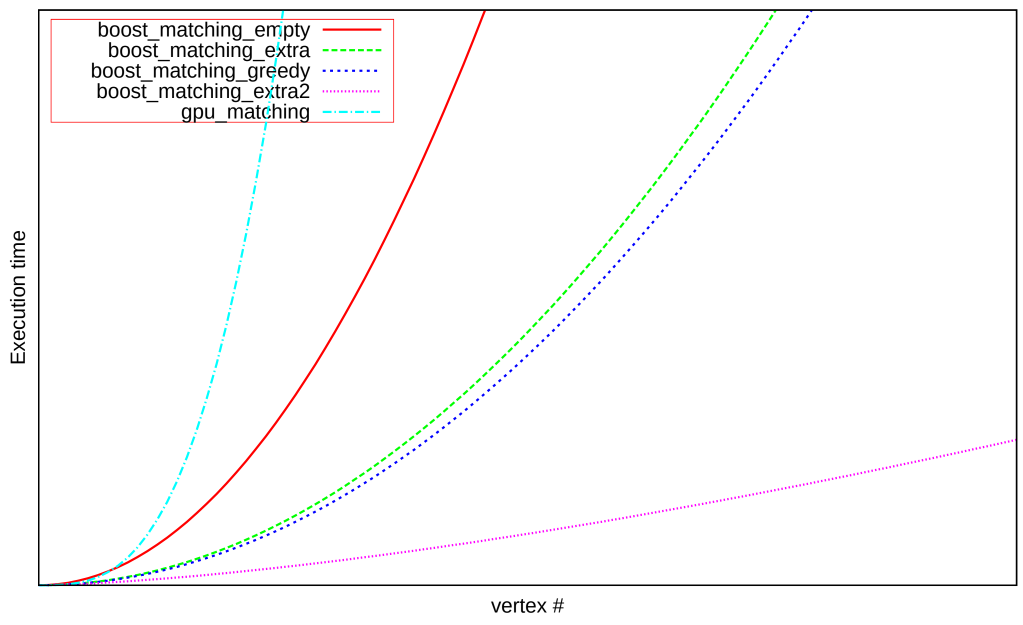

Figure 6.30: Average case performance of the matching algorithms in general case.

matchings, so with high probability any of the edges adjacent to those 4 vertices of degree 2 belong to some perfect matching. The following selections of edges into a perfect matching being constructed increase the number of vertices of degree 1 and 2, which further simplifies the next decisions. As *extra2_greedy_matching* heuristic analyses edges adjacent to vertices of lowest possible degrees and during the execution it updates the order of vertices to be analyzed, similar arguments as for SAT-based algorithm applies. The slowest algorithms on the average are the original Boost algorithms. *Greedy_matching* and *extra_greedy_matching* increase the performance of the Edmond's matching approach, but they still run in $O(n^2)$ time. Algorithm utilizing *greedy_matching* heuristic is slightly faster from *extra_greedy_matching*, as all vertices in the input graphs are of the same degree and the sorting of edges only increases the runtime of the algorithm. Somewhere in between those two performance extremes — $O(n)$ and $O(n^2)$ are found $O(n \log^4 n)$ and $O(n \log^2 n)$ matching algorithms for biconnected cubic graphs.

When the algorithms are executed against worst case graphs from the group J, SAT-based and *extra2_greedy_matching* do not run in linear time any longer. *Extra2_greedy_matching* reaches $O(n^2)$ execution time, but it remains the fastest among Boost-based algorithms. SAT-based algorithm becomes the slowest algorithm — its asymptotic time complexity matches $\theta(n^2 \log^2 n)$. The fastest algorithm for computing perfect matching in biconnected cubic graphs is our new $O(n \log^2 n)$ algorithm presented in section 4.3.

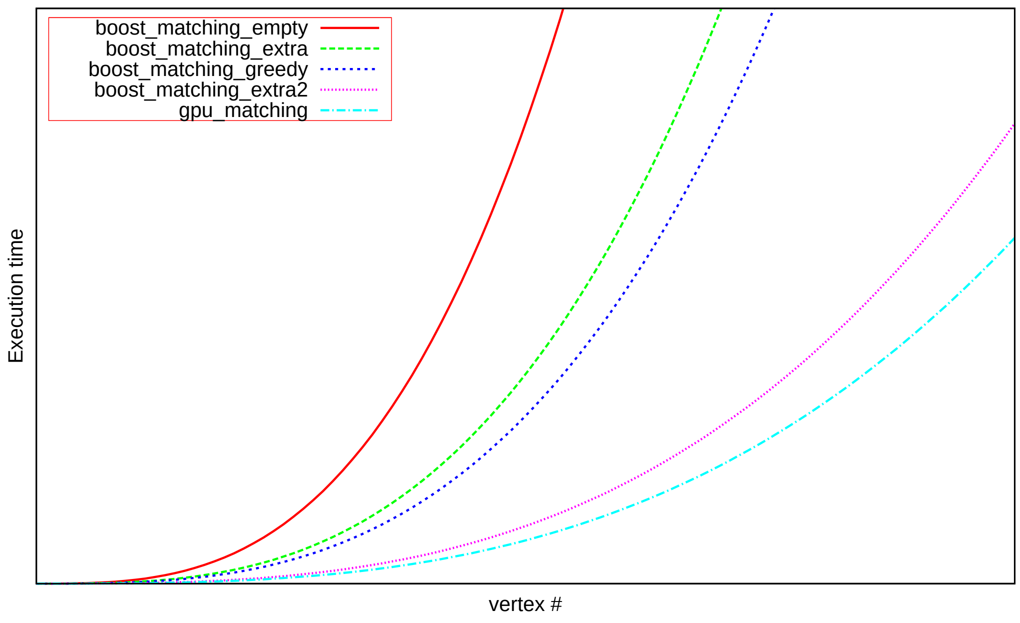

Figure 6.31: Worst case performance of the matching algorithms in general case.

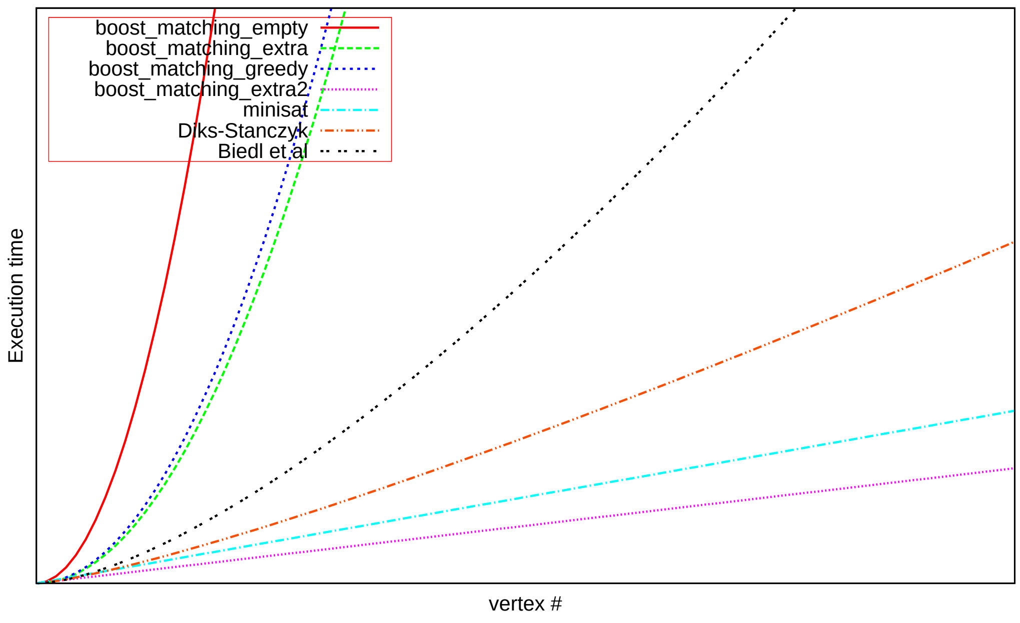

Figure 6.32: Average case performance of the matching algorithms against biconnected cubic graphs.

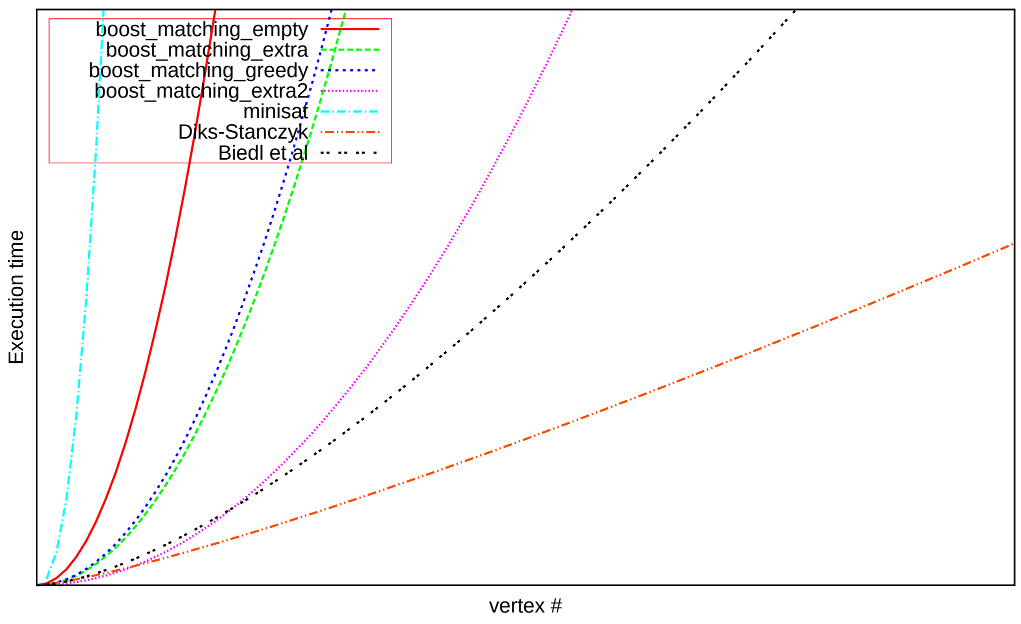

Figure 6.33: Worst case performance of the matching algorithms against biconnected cubic graphs.

Chapter 7

Open problems

In this chapter we present some open problems and other issues related to this work.

Nested dissection for planar graphs. M. Mucha and P. Sankowski in [42] proposed a modification to the algebraic algorithm for computing maximum matching. By applying the Separator Theorem for planar graphs it is possible to reorder the Tutte's matrix in such a way that the Gaussian elimination takes $O(n^{\frac{\omega}{2}})$ time. M. Mucha and P. Sankowski showed a maximum matching algorithm for planar graphs running in the same time. It would be interesting to try this approach with GPUs and obtain a fast parallel algorithm for solving maximum matching problem in case of planar graphs.

Fast algorithm for sparse graphs. The total work of the algebraic matching algorithm depends only on the number of vertices in an input graph. Is there a way to design a practical parallel algorithm with better running time for sparse graphs? Sparse matrix computation techniques might be applicable, see [8] [2].

Weighted matching. Is it possible to implement an efficient parallel algorithm for solving weighted maximum matching problem?

Multi GPU maximum matching. Our GPU matching algorithm was implemented with a single GPU in mind. It is interesting to try to distribute the algorithm to multi GPUs.

Linear time perfect matching algorithm for biconnected cubic graphs. The question posted by T. Biedl et al. whether there is a linear time algorithm for solving the perfect matching problem for biconnected cubic graphs remains unanswered.

Chapter 8

Appendix

Algorithm 20 $non-singular\_submatrix(T)$

```
Require: An $n \times n$ Tutte matrix $T$
Ensure: Maximal non-singular sub-matrix of $T$
 1: $A \leftarrow \emptyset$
 2: $B \leftarrow \emptyset$
 3: $T' \leftarrow T$
 4: for $x = 0$ to $n - 1$ do
 5:     for $y = 0$ to $n - 1$ do
 6:         if $T[x, y] \neq 0$ then
 7:             $A \leftarrow A \cup \{x\}$
 8:             $B \leftarrow B \cup \{y\}$
 9:             $T'' \leftarrow T$
10:             $r \leftarrow T[x, y]^{-1}$
11:             for $x' = 0$ to $n - 1$ do
12:                 for $y' = 0$ to $n - 1$ do
13:                     $T[x', y'] \leftarrow T[x', y'] - T''[x, y'] \cdot T''[x', y] \cdot r$
14:                 end for
15:             end for
16:         end if
17:     end for
18: end for
19: $A' \leftarrow A \cup B$
20: $B' \leftarrow B \cup A$
21: $R \leftarrow 0_{|A'| \times |B'|}$
22: for $x = 0$ to $|A'| - 1$ do
23:     for $y = 0$ to $|B'| - 1$ do
24:         $R[x, y] \leftarrow T'[A'[x], B'[y]]$
25:     end for
26: end for
27: return $R$
```

Algorithm 21 $LRU\_factorization(T)$

Require: An $n \times n$ matrix T
Ensure: LRU factorization of T

```
1:  $L \leftarrow I_{n\times n}$
2:  $R \leftarrow I_{n\times n}$
3:  $U \leftarrow I_{n\times n}$
4:  for $x = 0$ to $n-1$ do
5:      $rev \leftarrow T[i,i]^{-1}$
6:      for $x = i+1$ to $n-1$ do
7:          for $y = i+1$ to $n-1$ do
8:              $T[x,y] \leftarrow T[x,y] - T[x,i] \cdot T[i,y] \cdot rev$
9:          end for
10:     end for
11:     $R[i,i] \leftarrow -rev$
12:     for $x = i+1$ to $n-1$ do
13:         $U[x,i] \leftarrow T[x,i] \cdot rev$
14:     end for
15:     for $y = i+1$ to $n-1$ do
16:         $L[i,y] \leftarrow T[i,y] \cdot rev$
17:     end for
18: end for
19: return $L$, $R$, $U$
```

Algorithm 22 $L\_inversion(L)$

Require: An $n \times n$ lower triangular matrix L
Ensure: An inverse of L

```
1:  $L' \leftarrow 0_{n\times n}$
2:  for $y = 0$ to $n-1$ do
3:      for $x = 0$ to $y-1$ do
4:          $sum \leftarrow 0$
5:          for $z = 0$ to $y-1$ do
6:              $sum \leftarrow sum + L[z,y] \cdot L'[x,z]$
7:          end for
8:          $L'[x,y] \leftarrow -sum$
9:      end for
10: end for
11: return $L'$
```

Algorithm 23 $R\_inversion(R)$

```
Require: An $n \times n$ diagonal matrix $R$
Ensure: An inverse of $R$
1: $R' \leftarrow 0_{n \times n}$
2: for $i = 0$ to $n - 1$ do
3:     $R'[i, i] \leftarrow R[i, i]^{-1}$
4: end for
5: return $R'$
```

Algorithm 24 $transposition(T)$

```
Require: An $n \times n$ matrix $T$
Ensure: Transposition of matrix $T$
1: $T' \leftarrow 0_{n \times n}$
2: for $x = 0$ to $n - 1$ do
3:     for $y = 0$ to $n - 1$ do
4:         $T'[x, y] \leftarrow T[y, x]$
5:     end for
6: end for
7: return $T'$
```

Algorithm 25 $multiplication(A, B)$

```
Require: An $n \times n$ matrices $A$ and $B$
Ensure: Multiplication of $A$ and $B$
1: $C \leftarrow 0_{n \times n}$
2: for $x = 0$ to $n - 1$ do
3:     for $y = 0$ to $n - 1$ do
4:         $sum \leftarrow 0$
5:         for $z = 0$ to $n - 1$ do
6:             $sum \leftarrow sum + A[z, y] \cdot B[x, z]$
7:         end for
8:         $C[x, y] \leftarrow sum$
9:     end for
10: end for
11: return $C$
```

Algorithm 26 *edge_selection*(T, T^{-1})

```
Require: An $n \times n$ Tutte matrix $T$ representing a graph $G$ with a perfect matching
Require: An inverse of matrix $T$
Ensure: Perfect matching of $G$
 1: $M \leftarrow \emptyset$
 2: for $x = 0$ to $n - 1$ do
 3:    for $y = 0$ to $n - 1$ do
 4:       if $T[x, y] \neq 0$ and $T^{-1}[x, y] \neq 0$ then
 5:          $M \leftarrow M \cup \{x - y\}$
 6:          $T' \leftarrow T^{-1}$
 7:          $rev \leftarrow T^{-1}[x, y]^{-1}$
 8:          for $x' = 0$ to $n - 1$ do
 9:             for $y' = 0$ to $n - 1$ do
10:                $T^{-1}[x', y'] \leftarrow T^{-1}[x', y'] - T'[x, y'] \cdot T'[x', y] \cdot rev$
11:             end for
12:          end for
13:          $T' \leftarrow T^{-1}$
14:          $rev \leftarrow T^{-1}[y, x]^{-1}$
15:          for $x' = 0$ to $n - 1$ do
16:             for $y' = 0$ to $n - 1$ do
17:                $T^{-1}[x', y'] \leftarrow T^{-1}[x', y'] - T'[y, y'] \cdot T'[x', x] \cdot rev$
18:             end for
19:          end for
20:       end if
21:    end for
22: end for
23: return $M$
```

Algorithm 27 *algebraic_matching*(G)

```
Require: A graph $G = (V, E)$
Ensure: Maximum matching $M$ of $G$
 1: $T \leftarrow 0_{|V| \times |V|}$
 2: for $x = 0$ to $|V| - 1$ do
 3:     for $y = 0$ to $|V| - 1$ do
 4:         if $V[x] - V[y] \in E$ then
 5:             $k = rand(0, p - 1)$
 6:             $T[x, y] = k$
 7:             $T[y, x] = k^{-1}$
 8:         end if
 9:     end for
10: end for
11: $T' \leftarrow non-singular\_submatrix(T)$
12: $L, R, U \leftarrow LRU\_factorization(T')$
13: $L^{-1} \leftarrow L\_inversion(L)$
14: $R^{-1} \leftarrow R\_inversion(R)$
15: $U^{-1} \leftarrow transposition(L\_inversion(transposition(U)))$
16: $T'^{-1} \leftarrow multiplication(multiplication(L^{-1}, R^{-1}), U^{-1})$
17: $M \leftarrow edge\_selection(T', T'^{-1})$
18: return M
```

Algorithm 28 $parallel\_non-singular\_submatrix(T)$

```
Require: An $n \times n$ Tutte matrix representing an input graph
Ensure: Maximal non-singular sub-matrix of $T$
 1: $A \leftarrow \emptyset$
 2: $B \leftarrow \emptyset$
 3: $T' \leftarrow T$
 4: repeat
 5:    $[v, w] \leftarrow [-1, -1]$
 6:    for all $x$ such that $0 \leq x < n$ do
 7:       for all $y$ such that $0 \leq y < n$ do
 8:          if $T[x, y] \neq 0$ then
 9:             $[v, w] \leftarrow [x, y]$ /* write of a single arbitrary processor is successful */
10:          end if
11:       end for
12:    end for
13:    if $[v, w] \neq [-1, -1]$ then
14:       $A \leftarrow A \cup \{v\}$
15:       $B \leftarrow B \cup \{w\}$
16:       for all $x$ such that $0 \leq x < n$ do
17:          for all $y$ such that $0 \leq y < n$ do
18:             $T[x, y] \leftarrow T[x, y] - T[v, y] \cdot T[x, w] \cdot T[v, w]^{-1}$
19:          end for
20:       end for
21:    end if
22: until $[v, w] = [-1, -1]$
23: $A' \leftarrow A \cup B$
24: $B' \leftarrow B \cup A$
25: $T'' \leftarrow 0_{|A'| \times |B'|}$
26: for all $x$ such that $0 \leq x < |A'|$ do
27:    for all $y$ such that $0 \leq y < |B'|$ do
28:       $T''[x, y] \leftarrow T'[A'[x], B'[y]]$
29:    end for
30: end for
31: return $T''$
```

Algorithm 29 $parallel\_LRU\_factorization(T)$

```
Require: An $n \times n$ matrix $T$ to be factorized
Ensure: LRU factorization of $T$
 1: $L \leftarrow 0_{n\times n}$
 2: $R \leftarrow 0_{n\times n}$
 3: $U \leftarrow 0_{n\times n}$
 4: for $i = 0$ to $n-1$ do
 5:     $rev = T[i,i]^{-1}$
 6:     $R[i,i] \leftarrow -T[i,i]$
 7:     for all $x$ such that $i < x < n$ do
 8:         for all $y$ such that $i < y < n$ do
 9:             $T[x,y] \leftarrow T[x,y] - T[x,i] \cdot T[i,y] \cdot rev$
10:         end for
11:     end for
12:     for all $x$ such that $i < x < n$ do
13:         $U[x,i] \leftarrow T[x,i] \cdot rev$
14:     end for
15:     for all $y$ such that $i < y < n$ do
16:         $L[i,y] \leftarrow T[i,y] \cdot rev$
17:     end for
18: end for
19: return $L$, $R$, $U$
```

Algorithm 30 $parallel\_L\_Inversion(L)$

```
Require: An $n \times n$ lower triangular matrix $L$
Ensure: An inverse of $L$
 1: $L' \leftarrow 0_{n\times n}$
 2: for $y = 0$ to $n-1$ do
 3:     for all $x$ such that $0 \le x < y$ do
 4:         $sum \leftarrow 0$
 5:         for $z = 0$ to $y-1$ do
 6:             $sum \leftarrow sum + L[z,y] \cdot L'[x,z]$
 7:         end for
 8:         $L'[x,y] \leftarrow -sum$
 9:     end for
10: end for
11: return $L'$
```

Algorithm 31 *parallel_R_Inversion*(R)

```
Require: An $n \times n$ diagonal matrix $R$
Ensure: An inverse of $R$
1: for all $i$ such that $0 \leq i < n$ do
2:     $R[i,i] \leftarrow R[i,i]^{-1}$
3: end for
4: return $R$
```

Algorithm 32 *parallel_transposition*(T)

```
Require: An $n \times n$ matrix $T$
Ensure: A transposition of matrix $T$
1: $T' \leftarrow 0_{n\times n}$
2: for all $x$ such that $0 \leq x < n$ do
3:     for all $y$ such that $0 \leq y < n$ do
4:         $T'[x,y] \leftarrow T[y,x]$
5:     end for
6: end for
7: return $T'$
```

Algorithm 33 *parallel_multiplication*(A, B)

```
Require: An $n \times n$ matrices $A$ and $B$
Ensure: A multiplication of matrices $A$ and $B$
1: $C \leftarrow 0_{n\times n}$
2: for all $x$ such that $0 \leq x < n$ do
3:     for all $y$ such that $0 \leq y < n$ do
4:         $sum \leftarrow 0$
5:         for $z = 0$ to $n - 1$ do
6:             $sum \leftarrow sum + A[z,y] \cdot B[x,z]$
7:         end for
8:         $C[x,y] \leftarrow sum$
9:     end for
10: end for
11: return $C$
```

Algorithm 34 *parallel_edge_selection*(T, T^{-1})

```
Require: An $n \times n$ Tutte matrix $T$ representing a graph $G$
Require: An inverse of $T$
Ensure: A maximum matching of a graph $G$
 1: $M \leftarrow \emptyset$
 2: repeat
 3:    $[v, w] \leftarrow [-1, -1]$
 4:    for all $x$ such that $0 \leq x < n$ do
 5:       for all $y$ such that $0 \leq x < n$ do
 6:          if $T[x, y] \neq 0$ and $T^{-1}[x, y] \neq 0$ then
 7:             $[v, w] \leftarrow [x, y]$ /* write of a single arbitrary processor is successful */
 8:          end if
 9:       end for
10:    end for
11:    if $[v, w] \neq [-1, -1]$ then
12:       $M \leftarrow M \cup \{v - w\}$
13:       $rev \leftarrow T^{-1}[v, w]^{-1}$
14:       for all $x$ such that $0 \leq x < n$ do
15:          for all $y$ such that $0 \leq y < n$ do
16:             $T^{-1}[x, y] \leftarrow T^{-1}[x, y] - T^{-1}[v, y] \cdot T^{-1}[x, w] \cdot rev$
17:          end for
18:       end for
19:       $rev \leftarrow T^{-1}[w, v]^{-1}$
20:       for all $x$ such that $0 \leq x < n$ do
21:          for all $y$ such that $0 \leq y < n$ do
22:             $T^{-1}[x, y] \leftarrow T^{-1}[x, y] - T^{-1}[w, y] \cdot T^{-1}[x, v] \cdot rev$
23:          end for
24:       end for
25:    end if
26: until $[v, w] = [-1, -1]$
27: return $M$
```

Algorithm 35 *parallel_algebraic_matching*(G)

```
Require: A graph $G = (V, E)$
Ensure: A maximum matching $M$ of $G$
 1: $T \leftarrow 0_{|V| \times |V|}$
 2: for all $x$ such that $0 \leq x < |V|$ do
 3:    for all $y$ such that $0 \leq y < |V|$ do
 4:       if $V[x] - V[y] \in E$ then
 5:          $k = rand(1, p - 1)$
 6:          $T[x, y] = k$
 7:          $T[y, x] = k^{-1}$
 8:       end if
 9:    end for
10: end for
11: $T' \leftarrow parallel\_non - singular\_submatrix(T)$
12: $L, R, U \leftarrow parallel\_LRU\_factorization(T')$
13: $L^{-1} \leftarrow parallel\_L\_inversion(L)$
14: $R^{-1} \leftarrow parallel\_R\_inversion(R)$
15: $U^{-1} \leftarrow parallel\_transposition(parallel\_L\_inversion(parallel\_transposition(U)))$
16: $T'^{-1} \leftarrow parallel\_multiplication(parallel\_multiplication(L^{-1}, R^{-1}), U^{-1})$
17: $M \leftarrow parallel\_edge\_selection(T', T'^{-1})$
18: return M
```

Bibliography

[1] Boost, C++ Libraries. *http://www.boost.org/.*

[2] CGGPU: Sparse Matrix Data Structures on the GPU. *http://cml.berkeley.edu/∼bcox/cggpu.html.*

[3] FFLAS-FFPACK, Finite field linear algebra subroutines/package. *http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/FFLAS/.*

[4] M. G. Andrews, M. J. Atallah, D. Z. Chen, and D. T. Lee. Parallel algorithms for maximum matching in interval graphs. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 84–92, Washington, DC, USA, 1995. IEEE Computer Society.

[5] T. Biedl. Linear reductions of maximum matching. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 825–826, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[6] T. C. Biedl, P. Bose, E. D. Demaine, and A. Lubiw. Efficient algorithms for Petersen's matching theorem. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 130–139, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[7] N. Blum. A new approach to maximum matching in general graphs. In *Proc. 17th ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 586–597. Springer-Verlag, 1990.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22:917–924, July 2003.

[9] A. B. Borodin, J. Von zur Gathen, and J. E. Hopcroft. Fast Parallel Matrix and GCD Computations. Technical report, Ithaca, NY, USA, 1982.

[10] H. R. Brahana. A Proof of Petersen's Theorem. pages 59–63, 1917-1918.

[11] P. Chaudhuri. Finding maximum matching for bipartite graphs in parallel. 1994.

[12] Combinatorica. A Database of Graphs in Combinatorica Format. *http://www.cs.sunysb.edu/∼skiena/combinatorica/graphs/.*

[13] N. Corporation. CUBLAS Library. *http://developer.download.nvidia.com/-compute/cuda/2_0/docs/CUBLAS.*

[14] E. W. Dijkstra. A simple proof of Hall's Theorem. Jan. 1998.

[15] K. Diks and P. Stanczyk. Perfect Matching for Biconnected Cubic Graphs in $O(n \log^2 n)$ Time. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '10, pages 321–333, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] M. K. E. Dahlhaus and A. Lingas. A Parallel Algorithm for Maximum Matching in Planar Graphs. *TR-89-018.*

[17] J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[18] J. Egerváry. Matrix kombinatorius tulajdonságairól. *Matematikai és Fizikai Lapok*, 38:16–28, 1931.

[19] S. Even and Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 100–112, 1975.

[20] M. Fayyazi, D. Kaeli, and W. Meleis. An adjustable linear time parallel algorithm for maximum weight bipartite matching. *Inf. Process. Lett.*, 97:186–190, March 2006.

[21] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 123–133, New York, NY, USA, 1991. ACM.

[22] C. Fremuth-Paeger and D. Jungnickel. Balanced network flows. VIII. A revised theory of phase-ordered algorithms and the $O(\log(n^2/m)/\log n)$ bound for the nonbipartite cardinality matching problem. *Networks*, 41(3):137–142, 2003.

[23] O. Frink. A Proof of Petersen's Theorem. *The Annals of Mathematics*, 27:491–493, 1926.

[24] H. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching in graphs. *Journal of the ACM*, 23(2):221–234, 1976.

[25] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. ACM*, 38(4):815–853, 1991.

[26] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 315–324, New York, NY, USA, 1987. ACM.

[27] R. Greenlaw and R. Petreschi. Cubic graphs. *ACM Comput. Surv.*, 27:471–495, December 1995.

[28] K. M. H. Alt, N. Blum and M. Paul. Computing a Maximum Cardinality Matching in a Bipartite Graph in Time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37:237–240, 1991.

[29] N. J. A. Harvey. Algebraic Algorithms for Matching and Matroid Problems. *SIAM J. Comput.*, 39:679–702, July 2009.

[30] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 79–89, New York, NY, USA, 1998. ACM.

[31] R. Honsberger. Lovász' Proof of a Theorem of Tutte. Mathematical Gems II, pages 147–157, 1976.

[32] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16:372–378, June 1973.

[33] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[34] J. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1992.

[35] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 22–32, New York, NY, USA, 1985. ACM.

[36] D. König. Graphok és matrixok. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.

[37] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[38] L. Lovász. On determinants, matchings and random algorithms. In L. Budach, editor, *Fundamentals of Computation Theory*, pages 565–574. Akademie-Verlag, 1979.

[39] P. T. M. Mucha. Finding maximum matchings via Gaussian elimination, Warsaw University, Faculty of Mathematics, Informatics and Mechanics.

[40] Mathematica. Mathematica's GraphData collection.

[41] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, pages 17–27, 1980.

[42] M. Mucha and P. Sankowski. Maximum Matchings in Planar Graphs via Gaussian Elimination. In *Proceedings of the 12th Annual European Symposium on Algorithms*, pages 532–543, 2004.

[43] M. Mucha and P. Sankowski. Maximum Matchings via Gaussian Elimination. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 248–255, Washington, DC, USA, 2004. IEEE Computer Society.

[44] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, January 1987.

[45] N. S. N. Eén. An Extensible SAT-solver, http://minisat.se/.

[46] Networks. High productivity software for complex networks.

[47] U. of Florida. The University of Florida Sparse Matrix Collection.

[48] J. Petersen. Die Theorie der regulären Graphs. *Acta Mathematica*, 15:193–220, 1891.

[49] S. Rajasekaran and J. Reif. *Handbook of Parallel Computing: Models, Algorithms and Applications (Chapman & Hall/Crc Computer & Information Science Series).* 1 edition.

[50] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[51] A. Schrijver. Bipartite Edge Coloring in $O(\Delta m)$ Time. *SIAM Journal on Computing*, 28:841–846, 1999.

[52] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency.* Springer-Verlag, 2003.

[53] G. Shannon. Parallel Independent Set Algorithms for Sparse Graphs. *CSD-TR-634*, 1986.

[54] R. Sharan and A. Wigderson. A new NC algorithm for perfect matching in bipartite cubic graphs. In *Proceedings of ISTCS 96*, pages 56–65, 1996.

[55] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26:362–391, June 1983.

[56] P. Stańczyk. Parallel Maximum Matching with GPU. *Workshop on Concurrency, Specification, and Programming*, pages 562–573, 2009.

[57] R. E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[58] M. Thorup. Decremental dynamic connectivity. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 305–313, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[59] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM.

[60] W. T. Tutte. The factorization of linear graphs. *J. London Math. Soc.*, 22:107–111, 1947.

[61] W. T. Tutte. The factors of graphs. *Canadian Journal of Mathematics 4*, pages 314–328, 1952.

[62] V. Volkov and J. Demmel. Using GPUs to Accelerate the Bisection Algorithm for Finding Eigenvalues of Symmetric Tridiagonal Matrices. Technical Report UCB/EECS-2007-179, EECS Department, University of California, Berkeley, Dec 2007.

[63] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

Index