Finding Perfect Matchings in Bridgeless Cubic Multigraphs Without Dynamic (2-)connectivity

Paweł Gawrychowski
University of Wrocław, Poland

Mateusz Wasylkiewicz
University of Wrocław, Poland

Abstract

Petersen's theorem, one of the earliest results in graph theory, states that every bridgeless cubic multigraph contains a perfect matching. While the original proof was neither constructive nor algorithmic, Biedl, Bose, Demaine, and Lubiw [J. Algorithms 38(1)] showed how to implement a later constructive proof by Frink in $\mathcal{O}(n \log^4 n)$ time using a fully dynamic 2-edge-connectivity structure. Then, Diks and Stańczyk [SOFSEM 2010] described a faster approach that only needs a fully dynamic connectivity structure and works in $\mathcal{O}(n \log^2 n)$ time. Both algorithms, while reasonable simple, utilize non-trivial (2-edge-)connectivity structures. We show that this is not necessary, and in fact a structure for maintaining a dynamic tree, e.g. link-cut trees, suffices to obtain a simple $\mathcal{O}(n \log n)$ time algorithm.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases perfect matching, cubic graphs, bridgeless graphs, link-cut tree

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.59

Related Version *Previous Version*: https://arxiv.org/abs/2405.03856v1

Funding *Mateusz Wasylkiewicz*: Partially supported by Polish National Science Center grant 2018/29/B/ST6/02633.

1 Introduction

Finding a maximum cardinality matching in a given graph is one of the fundamental algorithmic problems in graph theory. For bipartite graphs, it can be seen as a special case of the more general problem of finding a maximum flow, which immediately implies a polynomial-time algorithm. Already in the early 70s, Hopcroft and Karp [17] obtained a fast $\mathcal{O}(m\sqrt{n})$ time algorithm for this problem, where m denotes the number of edges and n the number of vertices. For general graphs, Edmonds [11] designed an algorithm working in $\mathcal{O}(mn^2)$ time, and in 1980 Micali and Vazirani [22] stated an $\mathcal{O}(m\sqrt{n})$ time algorithm. For dense graphs, a better complexity of $\mathcal{O}(n^\omega)$, where ω is the exponent of $n \times n$ matrix multiplication, has been achieved by Mucha and Sankowski [23]. For the case of sparse graphs, i.e. $m = \mathcal{O}(n)$, a long and successful line of research based on applying continuous techniques resulted in an $m^{1+o(1)}$ time algorithm by Chen et al. [7] for the bipartite case. However, there was no further improvement for the general case, and the $\mathcal{O}(n^{1.5})$ time algorithm obtained by applying the approach of Micali and Vazirani [22] remains unchallenged.

This naturally sparked interest in searching for natural classes of sparse graphs that admit a faster algorithm. A natural candidate is a class of graphs that always contain a perfect matching. One of the earliest results in graph theory attributed to Petersen [24], states that every bridgeless cubic graph contains a perfect matching, where cubic means that the degree of every vertex is exactly 3, while bridgeless means that it is not possible to remove a single edge to disconnect the graph. In fact, the theorem is still true for a cubic multigraph with at most two bridges, and from now on we will consider multigraphs, i.e. allow loops and parallel

32nd Annual European Symposium on Algorithms (ESA 2024).
Editors: Timothy Chan, Johannes Fischer, John Iacono, and Grzegorz Herman; Article No. 59; pp. 59:1–59:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

edges. The original proof was very complicated and non-constructive, but Frink [12] provided another approach that can be easily implemented to obtain a perfect matching in $\mathcal{O}(n^2)$ time. The high-level idea of this approach is to repeatedly apply one of the two possible reductions, in each step choosing the one that maintains the invariant that the current multigraph is bridgeless and cubic. Then, we revert the reductions one-by-one, which possibly requires finding an alternating cycle to make sure that a particular edge does not belong to the matching. Biedl, Bose, Demaine, and Lubiw [4] improved the time complexity to $\mathcal{O}(n \log^4 n)$ thanks to two insights. First, we can employ a fully dynamic 2-edge-connectivity structure of Holm, de Lichtenberg, and Thorup [15] to decide which reduction should be applied. Second, finding an alternating cycle can be avoided by requiring that a chosen edge does not belong to the matching. Diks and Stańczyk [9] further improved the complexity to $\mathcal{O}(n \log^2 n)$ time by observing that in fact a fully dynamic connectivity structure suffices if we additionally maintain a spanning tree of the current multigraph in a link-cut tree. By plugging in the fully dynamic connectivity structure by Wulff-Nilsen [29], the complexity of their algorithm can be further decreased to $\mathcal{O}(n \log^2 n / \log \log n)$. Alternatively, at the expense of allowing randomization and bit-tricks, plugging in the structure of Huang, Huang, Kopelowitz, Pettie, and Thorup [18] results in expected $\mathcal{O}(n \log n (\log \log n)^2)$ running time.

Our contribution. The algorithms of Biedl, Bose, Demaine, and Lubiw [4] and Diks and Stańczyk [9] can be seen as efficient implementations of Frink's proof [12]. While both are reasonably simple (see Section 4), they arguably hide some of their complexity in the fully dynamic (2-edge-)connectivity structure. We show that in fact this can be avoided, and present an implementation that only needs a structure for maintaining a dynamic tree, such as a link-cut tree. Inspired by the idea of Holm, de Lichtenberg, and Thorup used in the fully dynamic 2-edge-connectivity structure [15], we maintain a spanning tree T of the current multigraph G. Further, for each edge e of T we maintain its detour edge, denoted $\text{cover}(e)$, with the property that e belongs to the unique path connecting the endpoints of $\text{cover}(e)$ in T. In other words, $\text{cover}(e)$ is a witness for e not being a bridge of G. It turns out that inspecting the detour edges for all edges removed during a reduction, and checking how their endpoints are arranged in T, sufficies to determine which of the two possible reductions maintains that G is bridgeless. We apply this reduction and then we update T and the detour edges using newly added edges. This guarantees that the obtained graph is still bridgeless. The whole reduction step can be implemented to run in $\mathcal{O}(\log n)$ time using any structure for maintaining a dynamic tree. This results in a simple and self-contained algorithm that works in deterministic $\mathcal{O}(n \log n)$ time without any bit-tricks.

Applications. Petersen's theorem can be generalized to cubic multigraphs with at most two bridges. Finding a perfect matching in such a multigraph can be easily reduced to finding a perfect matching in a bridgeless cubic multigraph in linear time [4]. Hence, our algorithm can be used for finding a perfect matching in a cubic multigraph with at most two bridges in $\mathcal{O}(n \log n)$ time. We note that generalizing our algorithm to arbitrary cubic graphs can be difficult, as finding a perfect matching in a general graph can be reduced in linear time to finding a perfect matching in a cubic graph [3].

The complement of a perfect matching of a cubic graph is a 2-factor. A 2-factor can be used to approximate the graphic TSP problem, where we have to find a shortest tour visiting all vertices of an undirected graph. Several approximation algorithms for the graphic TSP problem in cubic graphs were presented [2, 5, 6, 8, 10, 14, 27]. Recently, Wigal, Yoo and Yu [28] presented a 5/4-approximation algorithm for the graphic TSP problem in cubic graphs which works in $\mathcal{O}(n^2)$ time. We can obtain a faster algorithm for this problem in bridgeless cubic graphs using the well-known technique of *subtour patching* [21] by merging the cycles of a 2-factor of the input graph into a single tour. However, its approximation ratio is 5/3, since every cycle of the computed 2-factor has length at least three. We can improve the approximation ratio to 3/2 by computing a 2-factor with no cycles of length three. Finding such a 2-factor can be reduced in $\mathcal{O}(n)$ time to finding an ordinary 2-factor in bridgeless cubic multigraphs by first contracting the cycles of length three of the input graph as described, for example, by Kobayashi [19, Lemma 3]. Thus, we obtain a 3/2-approximation algorithm working in $\mathcal{O}(n \log n)$ time.

Another application of finding a perfect matching in a bridgeless cubic graph is finding a P_4-decomposition in a bridgeless cubic graph, which consists in partitioning the set of edges of the input graph into a collection of paths of length exactly three. Kotzig [20] presented a simple construction that, given a perfect matching M of a cubic graph, finds its P_4-decomposition in linear time by directing every cycle of the complement of M. Therefore, using our algorithm, such a decomposition can be found in $\mathcal{O}(n \log n)$ time in a bridgeless cubic graph.

2 Preliminaries

We refer to a multiset of size exactly two as a *multipair*. A *multigraph* is an ordered pair $G = (V, E)$ where V is any finite set and E is any multiset of multipairs of elements from V. We refer to V as *vertices* of G and to E as *edges* of G. We refer to an edge $\{v, v\}$ for some $v \in V$ as *loop*. If there is exactly one copy of an edge $\{v, w\}$ in E, we refer to such an edge as *single*. We refer to an unordered pair of two different copies of $\{v, w\}$ in E as *double edge*. We define the *degree* of a vertex $v \in V$ as the total number of copies of v in all edges of G. Note that a loop $\{v, v\} \in E$ counts twice towards the degree of v. We assume that all considered graphs and multigraphs are connected. We say that G is *cubic* if the degree of every vertex of G is equal to three. Given an edge e of G, we denote by $G \setminus e$ the multigraph obtained from G by removing exactly one copy of e. Given a vertex v of G, we denote by $G \setminus v$ the multigraph obtained from G by removing vertex v together with all its incident edges. Given an edge $e = \{v, w\}$, for some $v, w \in V$, we denote by $G \cup e$ the multigraph obtained from G by adding one copy of e. An edge e of G is called a *bridge* if its removal disconnects G. We say that G is *bridgeless* if no edge of G is a bridge. Notice that a bridgeless cubic multigraph cannot contain any loops.

A multigraph $H = (V', E')$ is said to be a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A subgraph H of G is said to be *spanning* if $V = V'$. A spanning subgraph T of G is said to be a *spanning tree* of G if T is a tree. We denote the set of all vertices of a subgraph H of G by $V(H)$ and the multiset of all edges of H by $E(H)$. A *path* is a finite sequence of vertices and edges $P = (v_0, e_1, v_1, e_2, v_2, \ldots, v_{\ell-1}, e_\ell, v_\ell)$ of G, for some nonnegative integer ℓ, where the vertices $v_0, v_1, \ldots, v_\ell \in V$ are pairwise distinct, $e_1, e_2 \ldots, e_\ell \in E$, and $e_i = \{v_{i-1}, v_i\}$ for every $i \in \{1, \ldots, \ell\}$. We refer to ℓ as the *length* of P. We say that P *connects* vertices v_0 and v_ℓ. A *cycle* is defined similarly, except that v_0 and v_ℓ should be equal. We often identify a path or a cycle in G with the subgraph of G consisting of all its vertices and edges. Given a spanning tree T of G, we say that $e \in E(G) \setminus E(T)$ *covers* $f \in E(T)$ (in T) if $f \in E(P)$, where P is the path in T connecting the endpoints of e.

A subset $M \subseteq E$ is said to be a *matching* of G if the degree of every vertex in the subgraph of $H = (V, M)$ is at most one. A matching M of G is said to be *perfect* if the degree of every vertex in the subgraph $H = (V, M)$ is equal to one. The *perfect matching* *problem* consists in finding a perfect matching of a given multigraph, if it exists. Given an edge e, we say that it is *matched* (with respect to M) if it belongs to M. Otherwise, we say that it is *unmatched* (with respect to M). An *(M-)alternating cycle* is a cycle of G whose edges alternately belong and do not belong to M. An *application* of an M-alternating cycle A to M is an operation that removes all matched edges of A from M and adds all unmatched edges of A to M.

3 Link-cut trees

We need a structure for maintaining a forest $\mathcal{F}$ of vertex-disjoint unrooted trees, each of whose edges has a real-valued cost. We use link-cut trees of Sleator and Tarjan [25] which store a forest of vertex-disjoint *rooted* trees. Therefore, we root every tree of $\mathcal{F}$ at an arbitrary vertex. Link-cut trees support the following operations (among others) in $\mathcal{O}(\log n)$ time each, where n is the total number of vertices:

root($\mathbf{vertex}\, v$): return the root of the tree containing v.

cost($\mathbf{vertex}\, v$): returns the cost of the edge from v to its parent. We assume that v is not a root.

mincost($\mathbf{vertex}\, v$): returns the vertex w closest to root(v) such that the edge from w to its parent has minimum cost on the path connecting v and root(v). We assume that v is not a root.

update($\mathbf{vertex}\, v, \mathbf{real}\, x$): add x to the cost of every edge on the path connecting v and root(v).

link($\mathbf{vertex}\, u, v, \mathbf{real}\, x$): combine the trees containing u and v by adding an edge (u, v) with cost x, making v the parent of u. We assume that u and v are in different trees, and u is a root.

cut($\mathbf{vertex}\, v$): delete the edge from v to its parent. We assume that v is not a root.

evert($\mathbf{vertex}\, v$): modify the tree by making v the root.

As mentioned in the original paper, instead of real-valued costs we can in fact work with an arbitrary (but fixed) semigroup. In particular, we can use the semigroup $G = (E, \oplus)$, where $x \oplus y = x$ for every x, y. This allows us to maintain a forest of vertex-disjoint unrooted trees, each of whose edges e has its associated label cover(e), under the following operations:

connected(u, v): check if u and v belong to the same tree.

remove(u, v): remove an edge $\{u, v\}$ from the forest. We assume that the edge belongs to some tree.

add(u, v, x): add an edge $\{u, v\}$ to the forest, and set its label to be x. We assume that u and v are in different trees.

cover(u, v): return the label of the edge $\{u, v\}$. We assume that the edge belongs to some tree.

update(u, v, x): set the label of every edge on the path connecting u and v to be x. We assume that u and v belong to the same tree.

It is straightforward to implement these operations in $\mathcal{O}(\log n)$ time each by maintaining a link-cut tree, except that we use the semigroup G instead of real-valued costs.

- connected(u, v) checks if root(u) = root(v).
- remove(u, v) first calls evert(v), and then cut(u).
- add(u, v, x) proceeds by calling evert(u), and then link(u, v, x).
- cover(u, v) first calls evert(v), and then returns cost(u).
- update(u, v, x) is implemented by calling evert(v), and then update(u, x).

By maintaining another link-cut tree with real-valued costs we can also support checking if the paths connecting u with v and u' with v' share a common edge in $\mathcal{O}(\log n)$ time (assuming that u, v, u', v' all belong to the same tree). The cost of each edge is initially 0. To implement a query, we first call evert(u) and update($v, -1$). This has the effect of setting the cost of every edge on the path connecting u with v to -1. Then, we call evert(u') and check if mincost(v') returns -1, which happens if and only if the path connecting u' and v' shares a common edge with the path connecting u and v. Finally, we call evert(u) again, and then update($v, 1$) to restore the costs.

We note that any other structure for maintaining dynamic trees, e.g. top trees, could be used here in place of link-cut trees.

4 Outline of previous algorithms

In this section we present the previous algorithms for the perfect matching problem in bridgeless cubic multigraphs.

4.1 $\mathcal{O}(n^2)$ time algorithm based on Frink's proof

Frink's proof of Petersen's theorem can be easily turned into an algorithm. It uses the following theorem:

▶ **Theorem 1** (Frink). *Let G be any bridgeless cubic multigraph and $\{v, w\}$ any single edge of G. Let $\{a, v\}$ and $\{b, v\}$ be other edges of G incident to v. Let $\{c, w\}$ and $\{d, w\}$ be other edges of G incident to w. Define multigraphs $H_1 = ((G \setminus v) \setminus w) \cup \{a, c\} \cup \{b, d\}$ and $H_2 = ((G \setminus v) \setminus w) \cup \{a, d\} \cup \{b, c\}$ (see Figure 1). Then both H_1 and H_2 are cubic and at least one of them is bridgeless.*

We call the operation of producing H_1 (resp. H_2) from G a *straight* (resp. *crossing*) *reduction (of type I)* on $\{v, w\}$. We refer to both straight and crossing reductions as *reductions (of type I)*. We do not provide the proof of the above theorem, but stress that it will follow from the analysis of our algorithm, making the result self-contained.

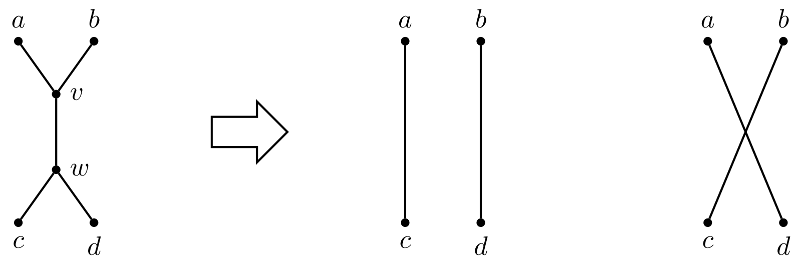

Figure 1 Straight and crossing reduction of type I on single edge $\{v, w\}$.

The idea of the algorithm based on Theorem 1 is to repeatedly perform the reduction on any single edge of the input multigraph G_0 to produce a sequence of multigraphs G_0, G_1, ..., G_k which are all cubic and bridgeless. It is easy to observe that every bridgeless cubic multigraph with more than two vertices has a single edge. Hence, we can assume that $k = n/2 - 1$ and $|V(G_k)| = 2$. To build a perfect matching of the input multigraph G_0, we can find any perfect matching of G_k and revert the reductions in a reverse order to find perfect matchings of G_{k-1}, G_{k-2}, ..., G_0. To this end, we need an auxiliary lemma.

▶ **Lemma 2** ([4]). *Let G be a bridgeless cubic multigraph, M a perfect matching of G, and e an edge of G. Then G has an M-alternating cycle that contains e that can be found in $\mathcal{O}(n)$ time.*

Then, every reduction can be reverted using the following theorem.

▶ **Lemma 3** ([4]). *Let G be any bridgeless cubic multigraph and G' be a multigraph obtained by performing a reduction on a single edge of G. Given a perfect matching M' of G', we can find a perfect matching of G in $\mathcal{O}(n)$ time.*

Proof. Without loss of generality, consider a straight reduction. We use the notation from the statement of Theorem 1. We construct a perfect matching M of G. We start off with the empty set. We add every edge of M' which belongs to G to M. Hence, it remains to add $\{v, w\}$ or some edges incident to $\{v, w\}$ to M. We consider the following three cases (see Figure 2).

a) If both $\{a, c\}$ and $\{b, d\}$ do not belong to M', we add $\{v, w\}$ to M.
b) If either $\{a, c\}$ or $\{b, d\}$ belongs to M', say $\{a, c\}$, we add $\{a, v\}$ and $\{c, w\}$ to M.
c) If both $\{a, c\}$ and $\{b, d\}$ belong to M', we find and apply an M'-alternating cycle of G' which contains $\{b, d\}$ to M' using Lemma 2. Then we get either case a) or b). ◀

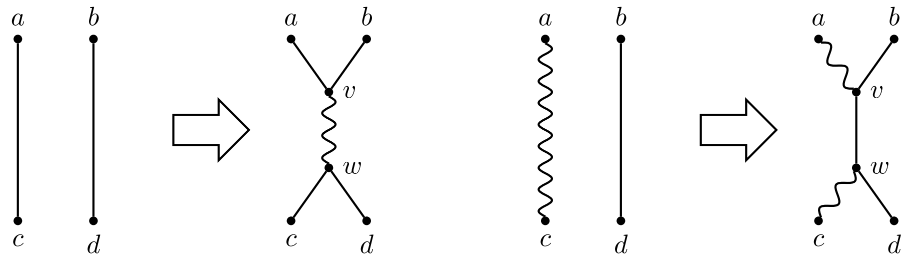

Figure 2 Reverting reduction of type I. The matched edges are marked by wavy lines.

Thus, we can find a perfect matching in a bridgeless cubic multigraph in $\mathcal{O}(n^2)$ time, since we can check if a multigraph is bridgeless and find an alternating cycle in linear time.

4.2 $\mathcal{O}(n \log^4 n)$ time algorithm with fully dynamic 2-edge-connectivity

Notice that there are two bottlenecks in the algorithm presented in Subsection 4.1: checking if a multigraph is bridgeless, and finding an alternating cycle. To check if a multigraph is bridgeless, we can use a fully dynamic 2-edge-connectivity structure. Such a structure maintains a multigraph G under the following operations:

- add an edge to G,
- remove an edge from G,
- check if given two vertices of G are in the same 2-edge-connected component of G.

To remove the first bottleneck, Biedl, Bose, Demaine, and Lubiw [4] used the fully dynamic 2-edge-connectivity structure given by Holm, de Lichtenberg and Thorup [15] with $\mathcal{O}(\log^4 n)$ amortized time per operation, thus obtaining an $\mathcal{O}(n \log^4 n)$ time algorithm. They checked, after every reduction, if the vertices a, b, c and d are still in the same 2-edge-connected component. We note that plugging in a faster (and later) structure of Holm, Rotenberg and Thorup [16] improves the time complexity to $\mathcal{O}(n(\log n)^2(\log \log n)^2)$.

In order to remove the second bottleneck, the idea of Biedl, Bose, Demaine, and Lubiw [4] was to forbid the case where both edges of some $E(G_i) \setminus E(G_{i-1})$ belong to the found perfect matching. To this end, we choose any edge e_0 of the input multigraph G_0, and search for a perfect matching which does not contain e_0. Notice that Lemma 2 implies that such perfect matching always exists. Then, we perform a reduction on a single edge incident to e_0, and we define e_1 as an edge of $E(G_1) \setminus E(G_0)$ such that $e_0 \cap e_1 \neq \emptyset$, that is, e_0 and e_1 are incident to the same vertex. Note that we cannot perform the reduction if all edges of G_0 incident to e_0 are double edges, and in such a case we use an alternative reduction defined later. Next, we recursively find a perfect matching in G_1 which does not contain e_1. Again, we perform a reduction on a single edge incident to e_1, and so on. Recall that G_k consists of exactly two vertices, so it is trivial to find a perfect matching which does not contain e_k. Again, we revert all reductions to construct a perfect matching of the input multigraph G_0. However, now we can use the assumption that the perfect matching of G_i does not contain $e_i \in E(G_i) \setminus E(G_{i-1})$. Therefore, we can construct a perfect matching of G_{i-1} in constant time since we do not have to apply an alternating cycle to get rid of the case c) from the proof of Lemma 3. This optimization gives us the desired $\mathcal{O}(n \log^4 n)$ running time.

If all edges of G_i incident to e_i are double edges, we perform a *reduction of type II* on any edge incident to e_i instead of reduction of type I as follows. Consider a double edge $e = \{\{v, w\}, \{v, w\}\}$ of G_i. Let $\{a, v\}$ be a single edge incident to v and $\{b, w\}$ a single edge incident to w. The reduction removes both copies of $\{v, w\}$ and all their incident edges, adds an edge $\{a, b\}$ to the multigraph and defines $e_{i+1} = \{a, b\}$ (see Figure 3). When reverting this reduction, we have a guarantee that $\{a, b\}$ does not belong to a perfect matching, hence we can add any copy of $\{v, w\}$ to it.

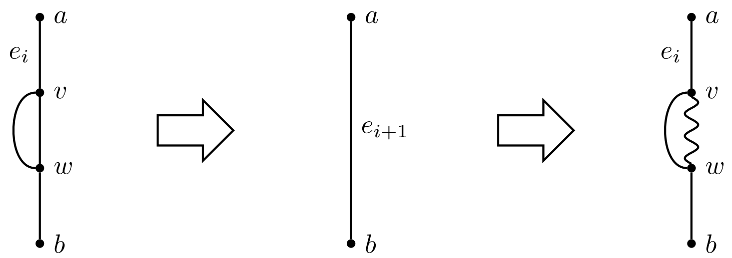

Figure 3 Reduction of type II (and its reverting). The matched edges are marked by wavy lines.

4.3 $\mathcal{O}(n \log^2 n)$ time algorithm with fully dynamic connectivity

Diks and Stańczyk presented a faster algorithm for the perfect matching problem in bridgeless cubic multigraphs by replacing a fully dynamic 2-edge-connectivity structure by a fully dynamic connectivity structure. Such a structure supports checking if two given vertices of the multigraph are in the same connected component, and allows adding and removing edges. They maintain a fully dynamic connectivity structure together with a spanning tree of the current multigraph. When performing a reduction of type I on a single edge e, they remove all edges incident to any endpoint of e from both the fully dynamic connectivity structure and the spanning tree. Checking which pairs of vertices adjacent to the endpoints of e are in the same connected component and how they are connected allows to check if we have to perform straight or crossing reduction of type I to obtain a bridgeless cubic multigraph as well. The spanning tree is maintained in a link-cut tree, so the total running time is dominated by the running time of the fully dynamic connectivity structure. Originally, the algorithm used either the structure of Holm, de Lichtenberg and Thorup [15], which works in $\mathcal{O}(\log^2 n)$ amortized time per operation, or the randomized variant presented by Thorup [26] which works in $\mathcal{O}(\log n(\log\log n)^3)$ expected amortized time per operation. However, now the fastest known fully dynamic connectivity structures are by Wulff-Nilsen [29] with $\mathcal{O}(\log^2 n/\log\log n)$ amortized time per operation, or the structure given by Huang, Huang, Kopelowitz, Pettie, and Thorup [18] with $\mathcal{O}(\log n(\log\log n)^2)$ amortized expected time per operation. Hence, the perfect matching problem in bridgeless cubic multigraphs can be solved in $\mathcal{O}(n\log^2 n/\log\log n)$ deterministic time, or $\mathcal{O}(n\log n(\log\log n)^2)$ expected time.

5 Outline of our algorithm

We give an outline of our algorithm below. It is based on the algorithm given by Biedl, Bose, Demaine, and Lubiw [4], but it does not need a fully dynamic 2-edge-connectivity structure. Let G_0 be the input multigraph, which is bridgeless and cubic. We proceed in iterations that construct the sequence $G_0, G_1, \ldots, G_k$ as follows.

Algorithm 1 Main algorithm.

```
e_0 ← any edge of G_0
T_0 ← any spanning tree of G_0
for e ∈ E(G_0) \ E(T_0) do
    cover_0(f) ← e for every f ∈ E(T_0) on the path in T_0 connecting both endpoints of e
end for
for i = 0 to n/2 − 2 do
    if e_i is incident to a single edge e of G_i then
        Obtain G_{i+1} by a reduction of type I on edge e of G_i
    else
        Obtain G_{i+1} by a reduction of type II on a double edge of G_i incident to e_i
    end if
    Obtain a spanning tree T_{i+1} of G_{i+1} from T_i
end for
M_k ← {e} for some e ∈ E(G_k) \ {e_k}
for i = n/2 − 2 downto 0 do
    Obtain a perfect matching M_i of G_i from M_{i+1} by reverting the corresponding reduction
end for
```

Similarly as in the algorithm given by Diks and Stańczyk, for every G_i, we construct a spanning tree T_i of G_i as well. Additionally, for every $e \in E(T_i)$ we maintain any edge from $E(G_i) \setminus E(T_i)$ which covers e in T_i, denoted $\text{cover}_i(e)$. Notice that such an edge always exists, since G_i is bridgeless. The spanning tree T_i and $\text{cover}_i(e)$, for every $e \in E(T_i)$, are maintained in a link-cut tree as described in Section 3. Moreover, we maintain an edge e_i, which will not belong to the found perfect matching M_i as in the algorithm presented in Subsection 4.2. We will construct the spanning trees $T_0, T_1, \ldots, T_k$ during the execution of the algorithm, making sure to maintain the following invariant.

▶ **Invariant 1.** *For every T_i and edge $e \in E(T_i)$, $\text{cover}_i(e)$ is an edge of $E(G_i) \setminus E(T_i)$ that covers e in T_i.*

6 Details

In this section we explain how to implement the reductions and update the maintained information during the execution of the algorithm. Moreover, we prove that Invariant 1 is maintained. In Subsection 6.4 we present the time and space complexity analysis.

6.1 Swap

We first define our atomic operation *swap* on an edge $e \in E(T_i)$. It consists in removing e from T_i, adding $e' = \text{cover}_i(e)$ to T_i, and setting $\text{cover}_i(f) = e$ for every edge f covered by e in the new T_i. We will be using swap operation as a black box. The following lemma proves that performing a swap does not spoil the maintained information.

▶ **Lemma 4.** *The swap operation maintains Invariant 1 for T_i.*

Proof. Assume that we perform a swap on an edge e and let $e' = \text{cover}_i(e)$. Let T^0 be the tree T_i before the swap and T^1 after the swap. Let P be the path in T^0 which connects both endpoints of e'. Since e' covers exactly the edges of P in T^0, $e \in E(P)$. Notice that e covers all edges of $P' = (P \setminus e) \cup e'$ in T^1, so every $\text{cover}_i(f)$, for $f \in E(P')$, is updated correctly. Consider any $f \in E(T^1) \setminus E(P')$. By construction, f belongs to $E(T^0)$, so $f' = \text{cover}_i(f)$ covers f in T^0. Therefore, there exists a path R in T^0 which connects both endpoints of f' such that $f \in E(R)$. We claim that f' covers f in T^1 as well. If $e \notin E(R)$, then R is a path in T^1 so we are done. Hence, assume that $e \in E(R)$. Notice that $f' \neq e'$ since e' covers only the edges of $E(P)$ in T^0. We construct a path R' in T^1 from R by replacing its fragment which is contained in P by going through edge e' instead of e (see Figure 4). Notice that $f \in E(R')$ since $f \notin E(P')$, so we are done. ◀

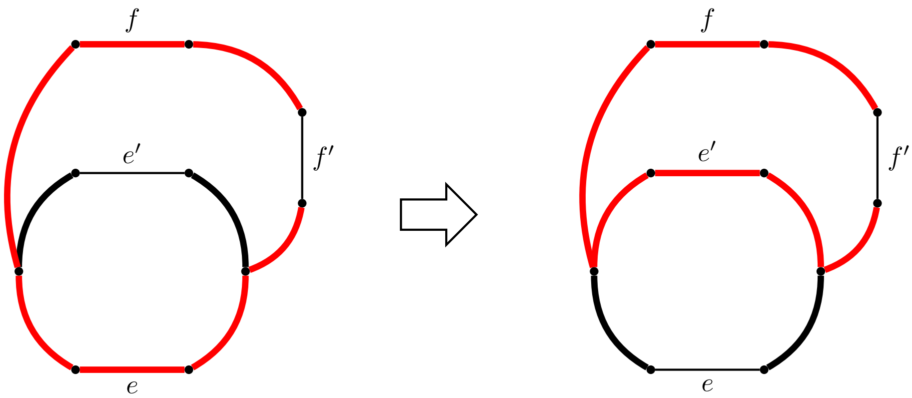

Figure 4 Proof of Lemma 4. The edges of T_i are marked by thick lines and edges of, correspondingly, R and R' are marked by red lines.

6.2 Reductions of type I

In this subsection we present how we implement a reduction of type I on a single edge $\{v, w\}$ of G_i. We use the notation from the statement of Theorem 1. Let A_i be the set of all edges of G_i incident to edge $\{v, w\}$. Notice that a, b, c and d are not necessarily distinct, but they are different from v and w since $\{v, w\}$ is single. Moreover, $v \neq w$.

The main idea is the following. Before performing the reduction, we reduce the number of cases to consider by performing several swaps on some edges incident to v or w. Recall that, by Lemma 4, these swaps do not spoil the maintained information. Then, we perform either a straight or a crossing reduction (of type I) on $\{v, w\}$, depending on how the vertices a, b, c and d are connected in $(T_i \setminus v) \setminus w$. We obtain T_{i+1} from T_i by deleting the removed edges and adding some of the new edges. This implicitly sets $\text{cover}_{i+1}(e) = \text{cover}_i(e)$ for every edge $e \in E(T_i) \cap E(T_{i+1})$. Then, we update $\text{cover}_{i+1}(e)$ for every edge e covered with the edges of $E(G_{i+1}) \setminus E(T_{i+1})$ in T_{i+1} accordingly. Finally, we set e_{i+1} to be the new edge which is incident in G_{i+1} to one of the endpoints of e_i in G_i.

If $\{v, w\} \in E(T_i)$, we perform a swap on the edge $\{v, w\}$. Hence, we can assume that $\{v, w\} \notin E(T_i)$. Moreover, if $A_i \subseteq E(T_i)$, we can perform a swap on at least one of the edges of A_i without adding $\{v, w\}$ to T_i. Thus, we assume that either two or three edges of A_i belong to $E(T_i)$. Furthermore, we can assume that, for every $e \in A_i \cap E(T_i)$, $\text{cover}_i(e) \in A_i \cup \{\{v, w\}\}$, as otherwise we can perform a swap on such edge e.

If $|A_i \cap E(T_i)| = 3$, we can assume that $\{d, w\} \notin E(T_i)$. We consider two subcases depending on how a, b, c and d are connected in $(T_i \setminus v) \setminus w$ (see Figure 5):

- both c and d are connected to a (or both to b), or
- c and d are connected to different vertices a and b.

Notice that the first subcase cannot happen: if both c and d are connected to a then, since $\text{cover}_i(\{b, v\}) \notin A_i \cup \{\{v, w\}\}$, we could have performed a swap on $\{b, v\}$. Hence, we are left with the second subcase. Assume, without loss of generality, that c is connected to a and d to b in $(T_i \setminus v) \setminus w$. Then we perform a crossing reduction on $\{v, w\}$. Moreover, we add exactly one of the added edges to T_{i+1}.

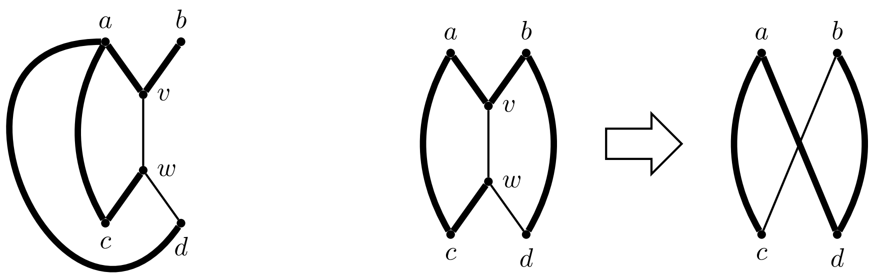

Figure 5 The case when $|A_i \cap E(T_i)| = 3$. The edges of T_i and T_{i+1} are marked by thick lines.

If $|A_i \cap E(T_i)| = 2$, assume that $\{a, v\}$ and $\{c, w\}$ belong to $E(T_i)$. Since $(T_i \setminus v) \setminus w$ is still connected in that case, we have the three following subcases depending on which pairs of the vertices a, b, c and d are connected in $(T_i \setminus v) \setminus w$ first (see Figure 6). Formally, we partition $\{a, b, c, d\}$ into two pairs $\{x, y\}$ and $\{x', y'\}$ such that the paths connecting x with y and x' with y' in $(T_i \setminus v) \setminus w$ are edge-disjoint (but not necessarily vertex-disjoint). Such edge-disjoint paths always exist, for example it is straightforward to verify that paths with the smallest total length are edge-disjoint. Then, we say that x is connected to y and x' to y'.

- If a is connected to b and c to d, we perform an arbitrary reduction of type I on $\{v, w\}$.
- If a is connected to c and b to d, we perform the crossing reduction on $\{v, w\}$.
- If a is connected to d and b to c, we perform the straight reduction on $\{v, w\}$.

In all of these subcases we add no new edges to T_{i+1}. Notice that the subcases may overlap.

After performing every reduction of type I, we update the maintained information as follows. For every edge $e \in E(G_{i+1}) \setminus E(G_i)$ which does not belong to T_{i+1}, we set $\text{cover}_{i+1}(f) = e$ for every edge f on the path in T_{i+1} connecting both endpoints of e.

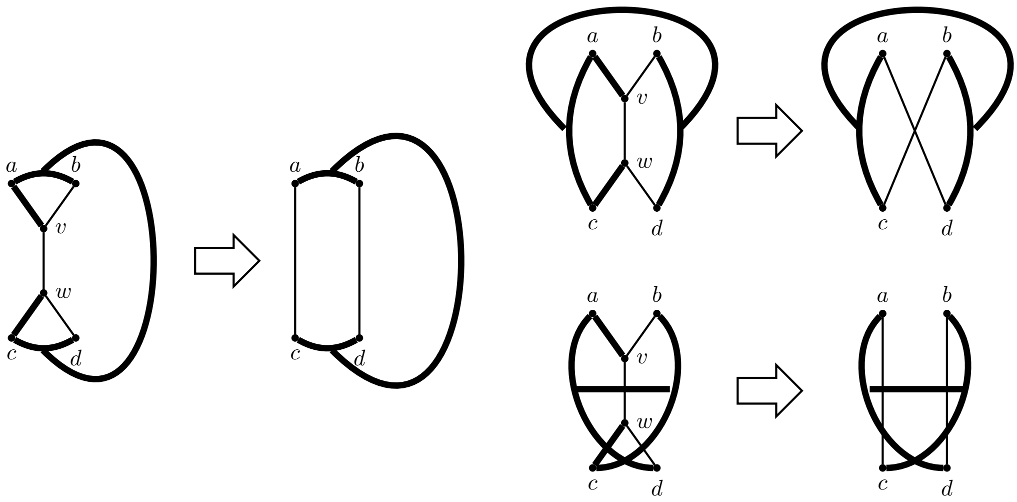

Figure 6 The case when $|A_i \cap E(T_i)| = 2$. The edges of T_i and T_{i+1} are marked by thick lines.

▶ **Lemma 5.** *Performing a reduction of type I maintains Invariant 1.*

Proof. Let S_i and S_{i+1} be the subgraphs of, respectively, T_i and T_{i+1} consisting of all edges which lie on some path in, respectively, T_i and T_{i+1} connecting some of the vertices v, w, a, b, c and d. It is easy to check that every edge of $E(S_{i+1})$ is covered by some edge added to G_{i+1} which does not belong to T_{i+1}. Hence, $\text{cover}_{i+1}(e)$ for every edge $e \in E(S_{i+1})$ is correct.

It is left to prove that $\text{cover}_{i+1}(e)$ is correct for every edge $e \in E(T_{i+1}) \setminus E(S_{i+1})$. By construction, $e \in E(T_i) \setminus E(S_i)$. Notice that $\text{cover}_{i+1}(e) = \text{cover}_i(e)$. We claim that $f = \text{cover}_i(e)$ covers e in T_{i+1}. First, we notice that f belongs to G_{i+1}. This follows from an easy observation that every edge of $(A_i \cup \{\{v, w\}\}) \setminus E(T_i)$ covers only some edges of $E(S_i)$ in T_i, so it cannot cover e in T_i. Consider a path P in T_i which connects both endpoint of f. From definition of f, $e \in E(P)$. If P does not contain any edges of $E(S_i)$, then P is a path in T_{i+1} as well, so f covers e in T_{i+1}. Otherwise, we construct a path P' from P by replacing its fragment which is contained in S_i by a corresponding path in S_{i+1} (see Figure 7). Since $e \notin E(S_i)$, $e \in E(P')$. Hence, f covers e in T_{i+1}. ◀

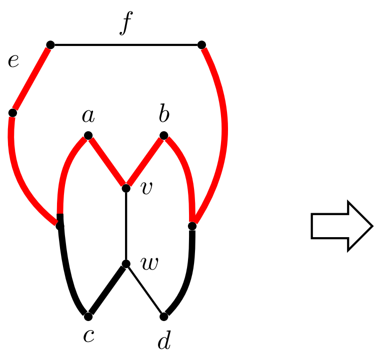

Figure 7 The proof of Lemma 5. The edges of T_i and T_{i+1} are marked by thick lines. The edges of P and P' are marked by red lines.

6.3 Reductions of type II

Now we consider the reduction of type II on a double edge incident to e_i of G_i. Recall that e_i is incident to two different double edges. Therefore, at least one of them, say $\{f_1, f_2\}$, contains some edge of $E(T_i)$ as otherwise T_i would be disconnected. We perform the reduction of type II on a double edge $\{f_1, f_2\}$ where $f_1 = \{v, w\} = f_2$. Let $a \neq w$ be a neighbor of v, so $e_i = \{a, v\}$, and $b \neq v$ be a neighbor of w. Assume that $f_1 \in E(T_i)$. Of course, $f_2 \notin E(T_i)$ then. First, we remove from G_i edges e_i, f_1, f_2 and $\{b, w\}$. If any of these edges is in T_i it is not included in T_{i+1}. Then, we create an edge $\{a, b\}$. We consider the two following subcases (see Figure 8).

- If both $\{a, v\}$ and $\{b, w\}$ belong to $E(T_i)$, we add $\{a, b\}$ to T_{i+1} and set $\text{cover}_{i+1}(\{a, b\}) = \text{cover}_i(\{a, v\})$.
- If only one of $\{a, v\}$ or $\{b, w\}$ belongs to $E(T_i)$, say $\{a, v\}$, we identify $\{a, b\}$ with $\{b, w\}$. This guarantees that $\text{cover}_{i+1}(e)$ is correct for every edge e on the path in T_{i+1} connecting a and b.

It is easy to check that Invariant 1 is maintained after a reduction of type II.

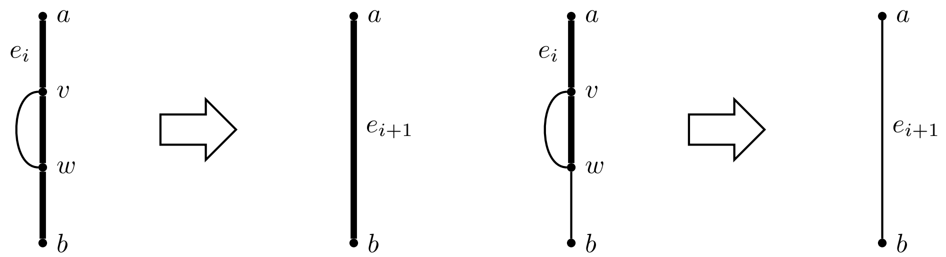

Figure 8 Performing a reduction of type II. The edges of T_i and T_{i+1} are marked by thick lines.

6.4 Complexity analysis

The algorithm performs less than n iterations of the for loop. Throughout the execution of the algorithm, we maintain the spanning tree T_i in a link-cut tree. Additionally, we maintain the edges incident to any vertex of G_i on a doubly-linked list. Each edge maintains a single bit denoting whether it belongs to T_i. Because the degree of every vertex of G_i is constant, this allows us to find a single edge incident to e_i, or choose a double edge $\{f_1, f_2\}$ such that $f_1 \in E(T_i)$, in constant time.

It is straightforward to verify that a swap operation can be implemented with a constant number of operations on the link-cut tree. To implement a reduction of type I, we first need a constant number of swap operations. Then, we need to distinguish between $|A_i \cap E(T_i)| = 3$ and $|A_i \cap E(T_i)| = 2$, which is easy by inspecting the bits maintained by the edges in A_i. In the latter case, we need to find a partition of $\{a, b, c, d\}$ into two pairs $\{x, y\}$ and $\{x', y'\}$ such that the corresponding paths in $(T_i \setminus v) \setminus w$ are edge-disjoint. To this end, we can check all 3 possibilities, and for each of them test if the corresponding paths in T_i are edge-disjoint in $\mathcal{O}(\log n)$ time. Finally, after deciding whether we should apply a crossing or a straight reduction, we construct G_{i+1} from G_i by removing vertices v and w and their incident edges (and possibly from T_i), and adding the appropriate two edges, and (in the case $|A_i \cap E(T_i)| = 3$) add one of them to T_{i+1}. Then, we update the cover values. Overall, this takes $\mathcal{O}(\log n)$ time.

To implement a reduction of type II, we obtain G_{i+1} from G_i by removing vertices v and w and their incident edges (and possibly from T_i), and adding edge $\{a, b\}$. In the first subcase, we update the cover value of the new edge. In the second subcase, we need to implicitly update the cover value of every edge e such that $\text{cover}_i(e) = \{b, w\}$ to the new edge. To this end, we think that each edge $e = \{u, v\}$ is an object that stores the endpoints u and v. Then, $\text{cover}_i(e)$ returns a pointer to the corresponding object. When creating a new edge, we create a new object. However, in the second subcase we reuse the object corresponding to the edge $\{b, w\}$, and modify its endpoints.

To reverse the reductions, we maintain the current matching M_i. Each edge stores a single bit denoting whether it belongs to M_i. Then, reverting a reduction of type I takes only constant time by inspecting one of the new edges and checking if it belongs to M_i. Depending on the case, we appropriately update M_i. Reverting a reduction of type II is even simpler, as we always add one copy of the double edge to M_i, and possibly need to restore the object corresponding to the edge $\{b, w\}$. For both types, we remove the new edges and add back the removed vertices and edges.

The overall time complexity is $\mathcal{O}(n \log n)$, and the algorithm uses $\mathcal{O}(n)$ space.

7 Conclusions

We presented a simple algorithm for the perfect matching problem in bridgeless cubic multigraphs, which works in $\mathcal{O}(n \log n)$ deterministic time. As opposed to the previous algorithms, it does not use any complex fully dynamic (2-edge-)connectivity structure. The natural open question is to further improve the time complexity.

Another open problem is to apply a similar approach to the *unique perfect matching problem* in sparse graphs. It consists in checking if a given graph admits exactly one perfect matching, and finding it if so. The fastest known deterministic algorithm for this problem was given by Gabow, Kaplan and Tarjan [13], and takes $\mathcal{O}(n(\log n)^2(\log \log n)^2)$ time when using the fastest fully dynamic 2-edge-connectivity structure given by Holm, Rotenberg and Thorup [16]. Note that the unique perfect matching problem can be solved in optimal linear time in dense graphs by using the decremental dynamic 2-edge-connectivity structure given by Aamand et al. [1].

References

1. Anders Aamand, Adam Karczmarz, Jakub Łącki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, pages 6:1–6:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
2. Nishita Agarwal, Naveen Garg, and Swati Gupta. A 4/3-approximation for TSP on cubic 3-edge-connected graphs. *Operations Research Letters*, 46(4):393–396, 2018.
3. Therese Biedl. Linear reductions of maximum matching. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 825–826, USA, 2001. Society for Industrial and Applied Mathematics.
4. Therese C. Biedl, Prosenjit Bose, Erik D. Demaine, and Anna Lubiw. Efficient algorithms for Petersen's matching theorem. *Journal of Algorithms*, 38(1):110–134, 2001.
5. Sylvia Boyd, René Sitters, Suzanne van der Ster, and Leen Stougie. The traveling salesman problem on cubic and subcubic graphs. *Mathematical Programming*, 144, July 2011.
6. Barbora Candráková and Robert Lukot'ka. Cubic TSP - a 1.3-approximation. *SIAM J. Discret. Math.*, 32:2094–2114, June 2015.
7. Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *FOCS*, pages 612–623. IEEE, 2022.

8 Jose Correa, Omar Larré, and José Soto. TSP tours in cubic graphs: Beyond 4/3. *SIAM Journal on Discrete Mathematics*, 29:915–939, October 2015.

9 Krzysztof Diks and Piotr Stańczyk. Perfect matching for biconnected cubic graphs in $O(n \log^2 n)$ time. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, pages 321–333, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

10 Zdenek Dvorák, Daniel Král, and Bojan Mohar. Graphic TSP in cubic graphs. In *Symposium on Theoretical Aspects of Computer Science*, 2017.

11 Jack Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17, 1965.

12 Orrin Frink. A proof of Petersen's theorem. *Annals of Mathematics*, 27(4):491–493, 1926.

13 Harold N. Gabow, Haim Kaplan, and Robert E. Tarjan. Unique maximum matching algorithms. *Journal of Algorithms*, 40(2):159–183, 2001.

14 David Gamarnik, Moshe Lewenstein, and Maxim Sviridenko. An improved upper bound for the TSP in cubic 3-edge-connected graphs. *Operations Research Letters*, 33:467–474, September 2005.

15 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

16 Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 35–52, 2018.

17 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

18 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *TheoretiCS*, 2023.

19 Yusuke Kobayashi. A simple algorithm for finding a maximum triangle-free 2-matching in subcubic graphs. *Discrete Optimization*, 7(4):197–202, 2010.

20 Anton Kotzig. Z teorie konečných pravidelných grafov tretieho a štvrtého stupňa. *Časopis pro pěstování matematiky*, 082(1):76–92, 1957.

21 E. L. Lawler, Jan Karel Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley-Interscience, 1985.

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

23 Marcin Mucha and Piotr Sankowski. Maximum matchings via Gaussian elimination. In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 248–255, November 2004.

24 Julius Petersen. Die Theorie der regulären graphs. *Acta Mathematica*, 15:193–220, 1891.

25 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

26 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Symposium on the Theory of Computing*, 2000.

27 Anke van Zuylen. Improved approximations for cubic bipartite and cubic TSP. *Mathematical Programming*, 172:399–413, 2015.

28 Michael C. Wigal, Youngho Yoo, and Xingxing Yu. Approximating TSP walks in subcubic graphs. *J. Comb. Theory, Ser. B*, 158:70–104, 2021.

29 Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*, 2012.