X-Blossom: Massive Parallelization of Graph Maximum Matching

Dayi Fan
The Ohio State University
fan.1090@osu.edu

Rubao Lee
Freelance
lee.rubao@ieee.org

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

ABSTRACT

The blossom algorithm computes maximum matchings in graphs and has been widely applied across diverse domains, including machine learning, economic analysis, and other essential data analytics applications. As data scales and the demand for real-time processing intensifies, high-performance computing solutions have become indispensable. Over the years, substantial research efforts have been dedicated to improving the sequential blossom algorithm. However, developing an efficient parallel solution remains highly challenging due to the algorithm's intricate execution patterns, sequential recursive dependencies, dynamic data structure modifications, and inefficient path search.

By thoroughly analyzing existing solutions, we have identified critical issues and proposed a new parallel framework called X-Blossom. This framework eliminates recursion entirely, enables efficient searches for multiple disjoint paths, and employs a simple path table to trace paths, removing the need for dynamic graphs and trees. These efforts in algorithm development result in significant performance enhancement. Extensive experiments on real-world datasets show that X-Blossom outperforms all existing solutions, achieving up to 992x speedup compared to the fastest sequential baseline, and an average of 431x speedup over the state-of-the-art parallel solution using 8 cores. It also demonstrates excellent scalability, achieving an average speedup of 1.72x when threads double in scalability tests to 64 cores. To the best of our knowledge, X-Blossom is the fastest solution for this class of graph algorithms.

PVLDB Reference Format:
Dayi Fan, Rubao Lee, and Xiaodong Zhang. X-Blossom: Massive Parallelization of Graph Maximum Matching. PVLDB, 18(10): 3339-3353, 2025.
doi:10.14778/3748191.3748199

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Davis-Fan/X-Blossom.

1 INTRODUCTION

As one of the most fundamental problems in graph theory, graph maximum matching has found a wide spectrum of applications across numerous domains and emerging technologies. Specifically, in machine learning, graph maximum matching is applied to multi-object tracking [45, 65, 102] and intelligent recommendation systems [44, 74, 95] to improve data association. Moreover, in economics, graph maximum matching plays a crucial role by optimizing the allocation of resources in areas such as online markets [66, 71], ride-sharing systems [43, 96, 104], supply chain management [94, 103, 113], and financial exchange networks [12, 25, 30, 114]. With the growing popularity of social media, graph maximum matching is increasingly used in game user pairing [3, 34] and dating platforms [81, 91]. In healthcare, it significantly contributes to kidney exchange programs [82] and optimizes emergency systems [31, 69].

A *matching* in a graph is a set of edges such that no two edges share a common vertex. The *maximum matching*, also known as the maximum-cardinality matching, is a matching that includes the largest possible number of edges. An example of matching is shown in Figure 1. Essentially, finding the maximum matching in a graph can be viewed as the process of pairing as many vertices as possible, which facilitates optimal resource allocation and efficient connections, thereby boosting various real-world applications.

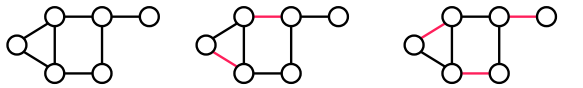


Figure 1: (a) is a graph. (b) shows a valid matching (red edges indicate the matching). (c) is the maximum matching.

Even today, solving the maximum matching for general graphs is still considered complicated [51, 98] due to the intricate computation process and the high complexity of algorithms. The first polynomial-time solution for general graphs was proposed in 1965, known as the *blossom algorithm* [32, 33], which has also become the basic and most widely used method. During matching computation, the blossom algorithm handles odd-length cycles by recursively modifying the graph structure, with a time complexity of $O(V^2E)$, where V is the number of vertices and E is the number of edges. In the following decades, extensive research efforts have focused on enhancing the efficiency of the sequential blossom algorithm by optimizing its implementation methods [14, 37–40] or tailoring it to specific graph structures [41, 48]. However, large-scale graphs in modern data-intensive applications often contain millions of vertices and edges [16, 84], turning sequential execution into an impractical choice for real-time performance. Therefore, developing an efficient and scalable parallel framework for graph maximum matching is not only crucial but also imperative.

Unfortunately, parallelizing the computation of graph maximum matching remains a highly non-trivial task. Existing research efforts [87, 93] to develop parallel blossom algorithms have achieved limited performance, due to the inherent difficulties of the problem. The goal of the blossom algorithm is to iteratively find valid paths

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748199

in the graph that can increase the matching size until it reaches its maximum. Three critical issues arise during parallel processing: (1) In the blossom algorithm, odd-length cycles in the graph may be contracted into a single vertex to recursively generate new graphs, which highlights that its operations are, by nature, sequential. This is the fundamental obstacle that hinders parallelization and has not been addressed by any existing parallel work. (2) In each search, at most one valid path can be found, so the size of the matching can increase by only one, leading to significant computational overhead. (3) The algorithm employs graphs and trees as dynamic data structures that are continuously being modified, causing irregular memory access patterns and severe data race issues.

These three systematic issues motivate us to propose an efficient and practical parallel framework that enables large-scale computation of maximum matching. By carefully examining the recursive process of odd cycle contracting, we propose a new recursion-free blossom algorithm that only records the valid path of each node in the cycle **without** the need for contraction. Unlike using a stack-based simulation to achieve a non-recursive implementation, our approach has fundamentally redefined the algorithm's logic and thus eliminates the recursion. Based on this, we further develop a new parallel framework, called X-Blossom, with an efficient lock-free synchronization mechanism that enables finding multiple valid disjoint paths concurrently, allowing the matching size to increase as much as possible in a single function call. Moreover, instead of using dynamic trees, we have implemented a path table to store the valid path for each vertex, removing the overhead of building and deconstructing trees and minimizing the time required for tracing paths. Our contributions are summarized as follows.

- We carefully analyze the intricate executions of the blossom algorithm, identify several fundamental issues in parallelization, and uncover the essence of blossom contraction. Developing an efficient parallel solution relies on effectively addressing these challenges.
- We propose a sequential recursion-free blossom algorithm that eliminates the recursive process of blossom contraction, serving as the foundation for massive parallel computation, along with the mathematical proof of its correctness.
- We further develop a parallel recursion-free blossom algorithm that enables concurrently finding multiple disjoint paths. It also incorporates an efficient lock-free synchronization mechanism and employs a path table to trace paths, eliminating the need for dynamic graphs and trees.
- Combining all efforts above, we develop X-Blossom, a massively parallel computation framework for graph maximum matching, which has been implemented on multicore processors. Extensive experiments show that X-Blossom significantly outperforms existing solutions and exhibits excellent scalability. It achieves up to 992x speedup over the fastest sequential implementation and surpasses the state-of-the-art parallel solution with an average of 431x speedup on real-world datasets by using 8 cores. In the scalability test to 64 cores, it demonstrates an average speedup of 1.72x when the number of threads doubles.

The paper is organized as follows. The basic blossom algorithm is introduced in §2. Then, we identify key issues and insights in §3. The sequential recursion-free algorithm is presented in §4. Details of the parallel recursion-free algorithm and its implementation are in §5. The experimental performance is demonstrated in §6. Finally, related works are discussed in §7, and we conclude our work in §8.

2 BACKGROUND

2.1 Matching and Augmenting Path

In a graph, a matching (denoted as M) is defined as a set of edges such that no two edges share a common vertex. The size of the matching, represented as $|M|$, is the number of edges in M, also referred to as its cardinality. The maximum matching is a matching with the maximum cardinality, which contains the largest possible number of edges. Edges in M are called matched edges, while edges not in M are unmatched edges. Similarly, a vertex is matched if it is an endpoint of one edge in M; otherwise, it is unmatched.

Figure 2(a) illustrates a graph G with a valid matching M_1. The red edges represent the edges in M_1, while the black edges are unmatched edges. An edge is denoted as (v_1, v_2) to represent an undirected edge between vertex v_1 and v_2. In this figure, $M_1 = \{(1, 2), (3, 4)\}$ and $|M_1| = 2$. Both 0 and 5 are unmatched vertices.

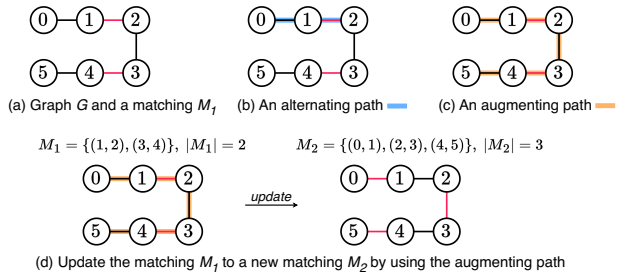


Figure 2: Matching, alternating path, and augmenting path (red edges are matched, black edges are unmatched)

An *alternating path* is a path in a graph where edges alternate between unmatched edges and matched edges. A path is denoted as $(v_1, v_2, \ldots, v_k)$, representing the path from vertex v_1 to v_k in the graph. Figure 2(b) shows an alternating path $(0, 1, 2)$, because edge $(0, 1)$ is unmatched and edge $(1, 2)$ is matched. An *augmenting path* is an alternating path that starts and ends at unmatched vertices. In Figure 2(c), the path $(0, 1, 2, 3, 4, 5)$ is an augmenting path since all edges alternate between being unmatched and matched, and the start vertex 0 and end vertex 5 are both unmatched.

Finding augmenting paths is crucial for increasing the size of the current matching. In an augmenting path, flipping all unmatched edges to be matched and all matched edges to be unmatched will generate a new matching with its size increased by one. As shown in Figure 2(d), an augmenting path for M_1 is $(0, 1, 2, 3, 4, 5)$, and if the unmatched edges $(0, 1)$, $(2, 3)$ and $(4, 5)$ are changed to be matched while the matched edges $(1, 2)$ and $(3, 4)$ are flipped to be unmatched, a new matching M_2 will be formed with the size increased to 3. According to Berge's theorem [9], a matching is maximum **if and only if** no augmenting path can be found. Thus, the main idea of maximum matching algorithms is to find augmenting paths and update the matching until no valid path exists.

2.2 Finding Augmenting Paths

Now the task of computing maximum matching has been converted into the search for augmenting paths. Finding augmenting paths in the graph relies on a pivotal tool called *alternating tree*, which provides a structured way to organize the search. An alternating tree is a tree rooted at an unmatched vertex from the graph and can expand by adding two edges each time: one unmatched edge followed by one matched edge. Thus, the path from its root to any leaf node is an alternating path and always has an even length. The forest consisting of these alternating trees is the *alternating forest*.

In Figure 3(a), the unmatched vertices are 0 and 5. Thus, two alternating trees can be constructed, rooted at 0 and 5, respectively. Since edge $(0, 1)$ is unmatched and edge $(1, 2)$ is matched, the first tree expands by adding these two edges after the root 0. Similarly, the second tree adds the unmatched edge $(5, 4)$ and the matched edge $(4, 3)$ after its root 5, as shown in Figure 3(b). For a node[1] v in the alternating tree, if the length of the path from the root to v is even, it is classified as an *even node*; otherwise, it is an *odd node*. The root node is an even node since the path length to itself is 0.

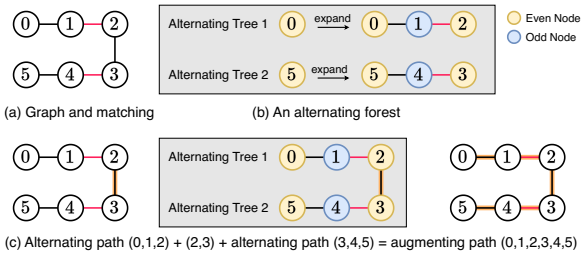


Figure 3: Finding augmenting paths by alternating trees

An augmenting path can be found if there exists an unmatched edge connecting two even nodes from different alternating trees. In an alternating tree, the root is an unmatched vertex, and the path from the root to an even node alternates between unmatched and matched edges with even length. Connecting two such paths with an unmatched edge forms an alternating path whose endpoints are both unmatched vertices, that is, an augmenting path. For example, in Figure 3(c), edge $(2, 3)$ is an unmatched edge which links the even node 2 from the first tree and the even node 3 from the second tree. The path from the root to node 2 in the first tree is $(0, 1, 2)$. Adding the unmatched edge $(2, 3)$ to this path results in $(0, 1, 2, 3)$. The path from the root to node 3 in the second tree is $(5, 4, 3)$. Appending its reverse to $(0, 1, 2, 3)$ produces the augmenting path $(0, 1, 2, 3, 4, 5)$.

2.3 Blossom Algorithm

Constructing alternating trees is an effective method for searching augmenting paths for bipartite graphs [5, 7, 27, 57]. In general graphs, the presence of odd-length cycles complicates this approach [80]. The blossom algorithm [33] addresses this challenge by introducing a mechanism to handle these odd-length cycles through recursive contraction. For a given graph G_0 and matching M_0, to find an augmenting path, the main idea is also to build alternating trees. An augmenting path is identified when an unmatched edge connects two even nodes from different alternating trees. However, if an unmatched edge connects two even nodes within the **same** tree, it forms an odd-length alternating cycle, named as a "*blossom*". The algorithm then contracts this cycle in G_0 (denoted as blossom B_0) into a single vertex, creating a new graph G_1. If another blossom B_1 is found in G_1, it will also be contracted, further modifying G_1 into a new graph G_2, continuing this recursive process. Once an augmenting path P_2 is identified in G_2, the blossom B_1 will be "lifted", restoring P_2 in G_2 to the corresponding augmenting path P_1 in G_1. Similarly, the blossom B_0 will be lifted, transforming P_1 in G_1 into the final augmenting path P_0 in the original graph G_0. The details are shown in Algorithm 1.

Algorithm 1 The Blossom Algorithm

Input: A graph G
Output: A maximum matching M

$M \leftarrow$ empty set
while True **do** ▷ continue searching for augmenting paths until none remain
  $P \leftarrow$ FIND-AUGMENTING-PATH(G, M)
  if P is empty **then**
    break ▷ no augmenting path found, so the matching is maximum
  else
    update M with the augmenting path P ▷ update the matching
  end if
end while

procedure FIND-AUGMENTING-PATH(G, M) ▷ the search procedure
  $F \leftarrow$ empty forest
  $N \leftarrow$ empty set, $N_{\text{next}} \leftarrow$ empty set ▷ store nodes to be checked when tree expands
  for each unmatched vertex u **do**
    add u as single-node tree to F and add u to N ▷ building alternating trees
  end for
  while $N \neq \emptyset$ **do**
    for each node v in N **do** ▷ every v in N is an even node actually
      for each unmatched edge $e = (v, w)$ in G **do**
        if $w \notin F$ **then** ▷ expand the tree
          $x \leftarrow$ the vertex matched with w in M
          add edges (v, w) and (w, x) to the tree of v and add x to N_{next}
        else
          if w is an even node **then**
            if root(v) $\neq$ root(w) **then** ▷ find an augmenting path
              $P \leftarrow$ path(root(v), ..., v) + (v, w) + path(w, ..., root(w))
              return P
            else ▷ find a blossom
              $B \leftarrow$ blossom formed by e and edges on the path (v, ..., w) in tree
              $G', M' \leftarrow$ contract G and M according to B
              $P' \leftarrow$ FIND-AUGMENTING-PATH(G', M')
              $P \leftarrow P'$ lifted with blossom B
              return P
            end if
          end if
        end if
      end for
    end for
    $N \leftarrow N_{\text{next}}$ ▷ nodes are updated after all nodes in N have been checked
  end while
  return empty path
end procedure

In the search procedure, the set N stores the nodes that need to be checked in each iteration. Initially, all roots are included in the set. As the tree expands, the newly introduced even node x is inserted into the set N_{next}, which will be checked in the next iteration. Each unmatched edge incident to the nodes in N is examined to determine whether it can expand the tree, construct an augmenting path, or form a blossom cycle. The blossom algorithm terminates when no augmenting path can be found.

Figure 4 is an example that illustrates the execution pattern of the blossom algorithm. Initially, the matching set is empty, and every vertex in G_0 is unmatched. When the search procedure runs, it constructs six single-node alternating trees in the forest, where each root is an even node. Then, it examines all unmatched edges related to these even nodes. The unmatched edge $(0, 1)$ connects

[1] We use *node* for trees and *vertex* for graphs to maintain a clear distinction.

node 0 and node 1 from two different trees, forming an augmenting path: $(0, 1)$. Updating the matching by using this path will flip edge $(0, 1)$ to be matched, resulting $M_0 = \{(0, 1)\}$. Following this, the search procedure runs again on G_0 and only four alternating trees are built this time (since vertices 0 and 1 are matched). Similarly, the augmenting path $(2, 3)$ is found and the matching is updated to $M_0 = \{(0, 1), (2, 3)\}$, as presented in the first two rows in Figure 4.

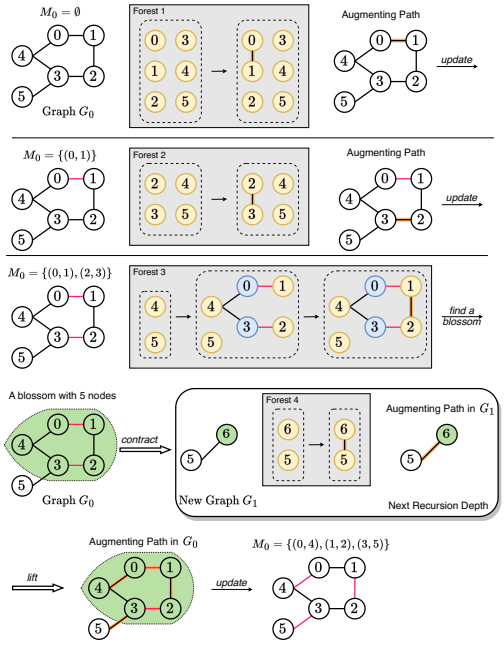


Figure 4: The execution pattern of blossom algorithm

Next, the search procedure is applied again to G_0, as shown in the third row of Figure 4. It builds two trees and the tree rooted at node 4 then expands two branches: one by adding $(4, 0)$ and $(0, 1)$, and another by adding $(4, 3)$ and $(3, 2)$. Two new even nodes, 1 and 2, are introduced, and an unmatched edge $(1, 2)$ connects these two nodes within a single tree, indicating that a blossom is formed. The green circle displays this odd-length alternating cycle: $(4, 0, 1, 2, 3, 4)$. The procedure then contracts this blossom cycle in G_0 into a new vertex 6, generating a new graph G_1. All edges inside the blossom will be deleted and all edges connected to the blossom cycle will be incident to the new vertex. From now on, the computation will proceed to the next recursion depth. For this new graph G_1, a simple augmenting path $(5, 6)$ is found. However, this augmenting path is only valid for G_1. To restore an augmenting path in G_0, the blossom is lifted upon exiting the recursion and the augmenting path $(5, 6)$ is unfolded as path $(5, 3, 2, 1, 0, 4)$. Then, the matching is updated to $M_0 = \{(5, 3), (2, 1), (0, 4)\}$, which is finally a maximum matching because no augmenting paths remain.

3 CHALLENGES AND INSIGHTS

3.1 Challenges in Parallel Processing

As data scales exponentially in modern applications, the transition from sequential to massively parallel processing has become crucial for solving maximum matching problems. However, the complex execution patterns of the blossom algorithm pose formidable challenges to parallelization, making it a highly non-trivial task both in algorithm and implementation.

Challenge #1: The recursive process of blossom contraction involves sequential modifications to the graph. The graph and augmenting path at the current recursion level rely on the structure of the blossom from the previous level, resulting in intricate data dependencies. Using stack-based simulation to replace recursion cannot resolve such inherent dependencies, which remain a fundamental structural issue in parallel computing. All existing parallel efforts [87, 93] focus on conducting concurrent searches for the augmenting path within each recursion level but have not addressed this critical challenge, thus failing to leverage parallelism effectively.

Challenge #2: Only one augmenting path can be returned by the search procedure. As a result, the matching size increases by only one for each search. For a graph with n vertices, if all vertices can be matched, the search must be repeated $\frac{n}{2}$ times sequentially because each path changes the state of the current matching. This makes it a time-intensive and computationally expensive approach for finding the maximum matching in large-scale graphs with millions of vertices.

Challenge #3: Dynamic data structures—graphs and trees—are continuously being updated and expanded during the execution of the blossom algorithm. This leads to serious data race problems when multiple threads attempt to modify the same tree or graph simultaneously. It also involves irregular memory access patterns, which degrade parallel performance when tracing paths in trees. Moreover, constructing and deconstructing these dynamic graphs and trees introduces additional computational costs.

These three inherent challenges are systematic and persistent in the blossom algorithm. To develop an efficient and scalable parallel framework, these issues must be carefully addressed. Although difficult, opportunities are often hidden within challenges if we solve the problem with the first principle thinking.

3.2 Blossom Contraction

Insight 1: *The essence of blossom contraction is to examine all unmatched edges associated with each node in the blossom cycle to determine whether an augmenting path exists.*

In the blossom algorithm, if an unmatched edge connects two even nodes within the same alternating tree, a blossom cycle forms. The entire cycle in the graph is then contracted into a single new vertex. During contraction, all unmatched edges incident to nodes within this cycle are redirected to this new vertex. At the next recursion level, these unmatched edges are further checked to verify if they can form an augmenting path. It is worth noting that the new vertex is also an even node and remains in the same alternating tree that the blossom nodes belonged to before contraction. The blossom contraction allows more unmatched edges to be examined, facilitating the discovery of a valid augmenting path.

In Figure 4, the blossom cycle in G_0 has been contracted to a new vertex 6 in G_1. The algorithm then checks the unmatched edge $(5, 6)$ in G_1 and identifies an augmenting path. However, the unmatched edge $(5, 6)$ in G_1 actually corresponds to the unmatched edge $(5, 3)$ in G_0. In Forest 3, this unmatched edge $(5, 3)$ would not be checked since node 3 is an odd node. Through blossom contraction, the

unmatched edge $(5, 3)$ has been assigned to the new even node 6 as $(5, 6)$, thereby enabling the detection of a new augmenting path.

3.3 Blossom Lifting

Insight 2: *The essence of blossom lifting is to trace a valid path from the unmatched-edge-connected node to the base, ensuring that the restored path remains an augmenting path.*

In a blossom cycle, the vertex connected to two unmatched in-cycle edges is called the *base*, and each blossom has exactly one base. When an augmenting path is found in the contracted graph, the recursion returns, and the algorithm lifts the blossom cycle to reconstruct the path in the original graph. The lifting process involves selecting a valid alternating path within the blossom cycle, from the unmatched-edge-connected-node to the base, ensuring that when this path is appended, the reconstructed path is a valid augmenting path in the original graph.

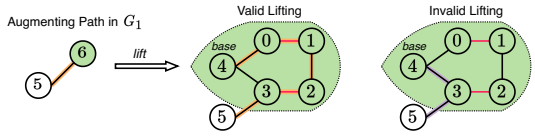


Figure 5: The lifting process of blossom

Figure 5 illustrates the lifting process of the blossom cycle from Figure 4. The base of the blossom cycle is 4 because it is the only vertex with two unmatched in-cycle edges $(4, 0)$ and $(4, 3)$. In G_1, the unmatched edge connected to the contracted vertex 6 is $(5, 6)$, which corresponds to the edge $(5, 3)$ before contraction. The lifting process then traces a valid path from node 3 to the base 4. Since the blossom is an odd-length cycle, there are two potential paths from node 3 to the base: path $(3, 2, 1, 0, 4)$ and path $(3, 4)$. To ensure the restored path remains an augmenting path, the path $(3, 2, 1, 0, 4)$ should be selected and appended after edge $(5, 3)$, generating a valid augmenting path $(5, 3, 2, 1, 0, 4)$. If path $(3, 4)$ is used instead, the resulting path $(5, 3, 4)$ would not qualify as an augmenting path.

We believe that it is only by understanding the underlying principles of contraction and lifting that we can find the most efficient way to parallelize the blossom algorithm. Unlike existing solutions, this paper aims to develop an efficient and scalable parallel framework that fundamentally addresses all three challenges.

4 RECURSION-FREE BLOSSOM ALGORITHM

In this section, we propose a sequential recursion-free blossom algorithm which eliminates the recursive process of contracting and lifting blossom cycles to first address **Challenge #1**. Before presenting our algorithm, we introduce and prove several lemmas that serve as its theoretical foundation.

4.1 Key Lemmas

Lemma 1. *If there exists an augmenting path P in graph G, then P must include an unmatched edge (v, w) connecting two even nodes, v and w, from two distinct alternating trees.*

Proof. Let $P = (x, \ldots, v, w, \ldots, y)$ be an augmenting path in G, where x and y are unmatched vertices. By definition, an augmenting path is an alternating path whose start and end vertices are both unmatched. Therefore, P must begin and end with unmatched edges, and the number of edges in P must be odd. Figure 6 illustrates such a path P. Let (v, w) be an arbitrary unmatched edge within P.

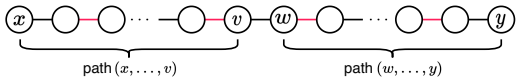


Figure 6: An augmenting path P in graph G

Each time the blossom algorithm constructs the forest, every unmatched vertex in G is treated as the root of a separate alternating tree. Therefore, the start vertex x and the end vertex y must belong to two distinct trees. The augmenting path P consists of three parts: the prefix $(x, \ldots, v)$, the unmatched edge (v, w), and the suffix $(w, \ldots, y)$. The subpath $(x, \ldots, v)$ is an alternating path that begins with an unmatched edge and ends with a matched edge, having even length. Therefore, the vertex v must be an even node in the alternating tree of x. Similarly, the reverse of $(w, \ldots, y)$ is also such an alternating path with even length, and the vertex w must be an even node in the alternating tree of y. Thus, Lemma 1 is proved. □

Lemma 2. *When a blossom is formed in an alternating tree, all odd nodes within the blossom cycle may be treated as even nodes.*

Proof. Let T_x be an alternating tree rooted at x and suppose that an arbitrary unmatched edge (v, w) connects two even nodes v and w in T_x. An odd-length blossom cycle is thus formed. Let b denote the base of the blossom, and let the odd nodes along the branch from b to v be labeled as $k_1, k_2, \ldots, k_n$ and the odd nodes along the branch from b to w be labeled as $l_1, l_2, \ldots, l_m$, as illustrated in Figure 7. Before the edge (v, w) is found, they are odd nodes because the alternating path from x to each of them is of odd length.

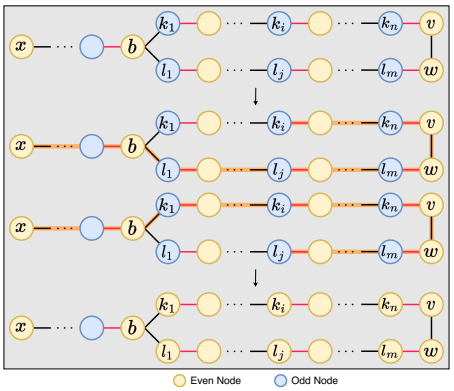


Figure 7: A blossom formed in a single alternating tree T_x

Consider a node k_i. After the blossom is formed, the path from x to k_i can consist of three parts: the subpath $(x, \ldots, b, l_1, \ldots, l_m, w)$, the unmatched edge (w, v), and the subpath $(v, k_n, \ldots, k_i)$. Since w is an even node, $(x, \ldots, b, l_1, \ldots, l_m, w)$ is an even-length alternating path that ends with a matched edge. Appending the unmatched edge (w, v) turns it into an odd-length path that ends with the unmatched edge. The remaining subpath $(v, k_n, \ldots, k_i)$ is an odd-length path that begins with a matched edge. Thus, the complete

path $(x, \ldots, b, l_1, \ldots, l_m, w, v, k_n, \ldots, k_i)$ is an even-length alternating path, which is valid for building an augmenting path if k_i is connected to another even node from a different tree. As a result of the blossom cycle, the odd node k_i can be treated as an even node.

Similarly, for an odd node l_j, the alternating path from x to l_j can be $(x, \ldots, b, k_1, \ldots, k_n, v, w, l_m, \ldots, l_j)$, which is also an even-length path and is valid for generating an augmenting path. Consequently, l_j can also be treated as an even node. Thus, Lemma 2 is proved.

□

4.2 Sequential Recursion-Free Algorithm

The two lemmas above reveal that, for a node in the alternating tree, only the even-length path from the root to the node itself is crucial for finding the augmenting path. This even-length path begins at an unmatched vertex (the root) and alternates between unmatched edges and matched edges. Two such paths connected by one unmatched edge can generate an augmenting path. Therefore, all even nodes in the alternating tree should be checked to verify if there exists an unmatched edge that can form an augmenting path.

Based on these findings, we propose a new sequential recursion-free blossom algorithm (Algorithm 2) that omits the entire recursive process of blossom contraction and lifting. When a blossom cycle is found in an alternating tree, **instead of** contracting the cycle and lifting it later, the new algorithm converts all odd nodes within the cycle to even nodes, stores their even-length paths from the root to themselves, and adds them to the check set — **without** modifying the graph and trees. If an unmatched edge connects one of these even nodes to another even node from a different tree, the even-length path will be read and utilized to build the valid augmenting path. By doing this, the graph no longer requires transformation and the recursion is completely removed, which resolves the essential data dependencies within the blossom algorithm, thereby addressing the key obstacle to enable parallel computation.

Algorithm 2 The Sequential Recursion-Free Blossom Algorithm

```
Input: A graph G
Output: A maximum matching M
M ← empty set
while True do                                   ▷ continue searching for augmenting paths until none remain
    P ← Recursion-Free-Find-Augmenting-Path(G, M)
    if P is empty then
        break                                   ▷ no augmenting path found, so the matching is maximum
    else
        update M with the augmenting path P     ▷ update the matching
    end if
end while

procedure Recursion-Free-Find-Augmenting-Path(G, M)   ▷ the search procedure
    F ← empty forest
    N ← empty set, N_next ← empty set
    for each unmatched vertex u do
        add u as single-node tree to F and add u to N
    end for
    while N ≠ ∅ do
        for each node v in N do
            for each unmatched edge e = (v, w) in G do
                if w is an even node ∧ root(v) ≠ root(w) then   ▷ find an augmenting path
                    P_1 ← Even-Path(v, tree_v, path_table)
                    P_2 ← Even-Path(w, tree_w, path_table)
                    P ← P_1 + (v, w) + the reverse of P_2
                    return P
                end if
                if w ∉ F then                   ▷ expand the tree
                    x ← the vertex matched with w in M
                    add edges (v, w) and (w, x) to tree_v and add x to N_next
                end if
                if w is an even node ∧ root(v) = root(w) then   ▷ find a blossom
                    B ← blossom formed by e and edges on the path (v, ..., w) in tree_v
                    for each node i in B do
                        if i is an odd node then
                            path_table[i] ← the even-length path from root(v) to i
                            set i as an even node and add i to N_next
                        end if
                    end for
                end if
            end for
        end for
        N ← N_next
    end while
    return empty path
end procedure

procedure Even-Path(i, tree_i, path_table)
    if path_table[i] is empty then              ▷ check if a valid path can be found in path table
        return path(root(i), ..., i) in tree_i
    else
        return path_table[i]
    end if
end procedure
```

4.3 Correctness and Complexity

Theorem 1. *For any graph, the sequential recursion-free blossom algorithm correctly finds a maximum matching.*

Proof. When the search procedure in the sequential recursion-free algorithm executes, each unmatched vertex is initially regarded as the root of an alternating tree, designated as an even node, and inserted into a check set. The search procedure then examines all unmatched edges incident to nodes in the check set for three independent cases: forming an augmenting path, expanding the alternating tree, or creating a blossom cycle.

When the tree expands, all newly introduced even nodes are added to the check set. **By Lemma 2**, if a blossom forms, all odd nodes within the blossom cycle are converted to even nodes and inserted into the check set. Thus, the check set includes all vertices that have an even-length alternating path to an unmatched vertex and examines all unmatched edges incident to them.

By Lemma 1, an augmenting path must be formed by an unmatched edge that connects two even nodes from different alternating trees. If no such unmatched edge is found among all even nodes, then no augmenting path exists in the graph, indicating that the current matching is already a maximum matching. Hence, the sequential recursion-free blossom algorithm correctly finds a maximum matching.

□

Theorem 2. *The time complexity of the sequential recursion-free blossom algorithm is $O(V^2E)$, where V is the number of vertices and E is the number of edges.*

Proof. Since the size of the maximum matching is at most $\frac{V}{2}$, the search procedure will be called at most $O(V)$ times. In each invocation, the main while loop iterates over all unmatched edges incident to even nodes, with at most $O(E)$ edges to examine. For each unmatched edge, there are three possible cases: if an augmenting path is found, tracing the path in the tree costs $O(V)$ time; if the tree expands, adding two new edges requires constant time $O(1)$; if a blossom cycle is formed, converting all odd nodes within the cycle to even nodes and recording their even-length paths takes $O(V)$ time. Thus, the time complexity of the sequential recursion-free blossom algorithm is: $O(V) \times O(E) \times (O(V) + O(1) + O(V)) = O(V^2E)$.

□

While the time complexity remains unchanged, this sequential recursion-free algorithm fundamentally resolves the inherent data dependencies caused by sequential modifications of the graph. Furthermore, it transforms the task of finding an augmenting path in dynamic graphs into computation over a static graph, providing significant benefits for parallel computing.

5 PARALLEL FRAMEWORK: X-BLOSSOM

Building on the sequential recursion-free algorithm proposed above, we develop a parallel recursion-free blossom algorithm that is both efficient and scalable to tackle **Challenge #2**. In addition, we discard the use of alternating trees in the implementation to overcome **Challenge #3**, as detailed in the final subsection.

5.1 Efficient Parallel Strategy

Challenge #2 arises from the inefficient search for augmenting paths in the basic blossom algorithm. Each time the search procedure runs, it returns at most one augmenting path, which means that the size of matching can increase by only one. For large graphs with millions of vertices, the procedure may need to be called sequentially millions of times. If this structural issue is not properly addressed, merely parallelizing the edge-checking stage will fail to achieve satisfactory performance.

To resolve this difficulty, we propose a new parallel strategy that concurrently identifies and returns multiple disjoint augmenting paths in a single procedure call. During the update process, unmatched edges in an augmenting path are flipped to matched, and all matched edges are flipped to unmatched. Since disjoint paths do not share any common vertex or edge, they can be processed simultaneously to increase the matching size while keeping the matching valid. Augmenting paths constructed from distinct pairs of trees must be disjoint. Hence, our parallel strategy is to find as many augmenting paths as possible by checking all unmatched edges of even nodes in the forest in parallel, which dramatically accelerates the process of finding the maximum matching.

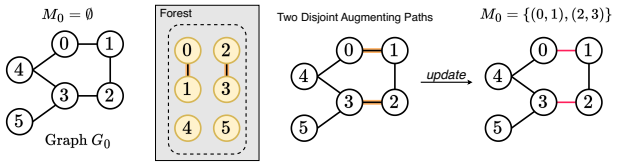


Figure 8: Find multiple disjoint augmenting paths in parallel

Figure 8 shows the same graph G_0 and matching M_0 as the first row in Figure 4. Initially, the matching is empty, so the forest builds six single-node trees rooted at unmatched vertices. All unmatched edges incident to these even nodes can be checked in parallel, and two disjoint paths are thus identified simultaneously: path $(0, 1)$ and path $(2, 3)$. By using these two paths to increase the matching, the size of M_0 grows directly by 2. To obtain the same matching, in Figure 4, the search procedure in the basic blossom algorithm must be called twice, and the forest and trees are rebuilt each time. Here, these two paths are found in parallel and the forest is built only once, which significantly improves the computational efficiency.

5.2 Parallel Recursion-Free Algorithm

Incorporating the strategy of computing disjoint augmenting paths in parallel with the sequential recursion-free algorithm, we propose a new parallel recursion-free blossom algorithm, as presented in Algorithm 3. In the sequential algorithm, each unmatched edge related to even nodes in alternating trees is processed individually through three stages: finding an augmenting path, expanding the tree, and detecting a blossom cycle. Since these three stages are distinct and independent, we can reformulate the iteration loops. For even nodes in the current check set, all unmatched edges incident to them are first examined in parallel to find multiple disjoint augmenting paths. If any valid paths are found, the function returns immediately. Next, these edges are checked concurrently for the second case, and all newly introduced even nodes are inserted into the new check set. Finally, they are verified in parallel to determine if they can form blossom cycles, which involves converting other odd nodes to even nodes, storing their even-length paths, and adding them to the new check set as well.

Algorithm 3 The Parallel Recursion-Free Blossom Algorithm

Input: A graph G
Output: A maximum matching M
$M \leftarrow$ empty set
while True **do** ▷ continue searching for augmenting paths until none remain
  $P_{\text{total}} \leftarrow$ Parallel-Recursion-Free-Find-Augmenting-Path(G, M)
  if P_{total} is empty **then**
    break ▷ no augmenting path found, so the matching is maximum
  else
    update M with all augmenting paths in P_{total} ▷ update the matching
  end if
end while

procedure Parallel-Recursion-Free-Find-Augmenting-Path(G, M) ▷ the search procedure
  $F \leftarrow$ empty forest
  $N \leftarrow$ empty set, $N_{\text{next}} \leftarrow$ empty set
  $P_{\text{total}} \leftarrow$ empty set ▷ A collection to store valid augmenting paths
  parallel for each unmatched vertex u **do**
    add u as single-node tree to F and add u to N
  end parallel for
  while $N \neq \emptyset$ **do**
    ▷ find multiple disjoint augmenting paths in parallel
    parallel for each node v in N and each unmatched edge $e = (v, w)$ in G **do**
      if w is an even node $\wedge$ root(v) $\neq$ root(w) **then**
        $P \leftarrow$ Parallel-Disjoint-Augmenting-Path(v, w, tree_v, tree_w, path_table)
        add P to P_{total}
      end if
    end parallel for
    if $P_{\text{total}} \neq \emptyset$ **then**
      return P_{total}
    end if

    ▷ expand multiple trees in parallel and add new even nodes to N_{next}
    parallel for each node v in N and each unmatched edge $e = (v, w)$ in G **do**
      if $w \notin F$ **then**
        Parallel-Expand-Tree(v, w, tree_v, path_table, M, N_{next})
      end if
    end parallel for

    ▷ find blossom cycles in parallel and add new even nodes to N_{next}
    parallel for each node v in N and each unmatched edge $e = (v, w)$ in G **do**
      if w is an even node $\wedge$ root(v) = root(w) **then**
        Parallel-Find-Blossom(v, w, tree_v, path_table, N_{next})
      end if
    end parallel for
    $N \leftarrow N_{\text{next}}$
  end while
  return empty
end procedure

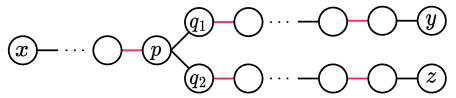
The advantages of this parallel recursion-free blossom algorithm are evident from its significant reduction in computational overhead. By finding multiple disjoint augmenting paths in parallel, it not only expedites the process of examining unmatched edges

but, more importantly, leads to a substantial increase in matching size. Moreover, since the algorithm eliminates recursion and avoids repeatedly rebuilding the forest and trees, it remarkably lowers both memory requirements and computation costs. Additionally, the parallel processing pattern for checking all unmatched edges simultaneously enables massive computations and delivers dramatic performance improvements for data-intensive applications with large-scale graphs. Therefore, this efficient and scalable parallel solution is the core of our parallel computation framework.

5.3 Lock-Free Synchronization

The parallel processing of all unmatched edges on even nodes inevitably introduces the data race problem, where multiple threads may concurrently access the same tree to expand branches, write even-length paths to the same location in the path table, or construct overlapping augmenting paths. This makes designing an effective synchronization mechanism crucial for all three checking stages.

Theorem 3. *An alternating tree in the forest can only be used to construct at most one augmenting path.*



Two augmenting paths: path $(x, \ldots, y)$ and path $(x, \ldots, z)$

Figure 9: Two augmenting paths: one constructed between T_x and T_y, and the other between T_x and T_z

Proof. Let T_x, T_y, and T_z denote three distinct alternating trees rooted at x, y, and z, respectively. Assume two augmenting paths are formed using tree T_x: one between T_x and T_y, and the other between T_x and T_z, as shown in Figure 9. Both paths must share the same endpoint x since they are constructed from the same tree T_x. Let the merging vertex of these two paths be denoted as p. If these two augmenting paths are used together to update the matching, the two unmatched edges (p, q_1) and (p, q_2) will be flipped to matched edges, which implies that p is matched with both q_1 and q_2, leading to an invalid matching. This contradiction completes the proof. □

According to Theorem 3, when the parallel recursion-free algorithm runs, each alternating tree can only be paired with other trees once. All threads should identify as many distinct pairs as possible to maximize the number of valid paths returned. Each tree can maintain its own atomic attribute to facilitate this process. When an unmatched edge connects two even nodes from different trees, a thread will attempt to use atomic Compare-And-Swap (CAS) operations to modify the attributes of these two trees and gain access to them. If either tree has already been accessed by another thread, the thread will revert any changes it made to the attributes, thereby ensuring all paths are disjoint and preserving the correctness. The procedure of finding paths in parallel is detailed in Algorithm 4.

The synchronization method above is lock-free and efficient. If one thread gains access to the first tree v, but the second tree w is being accessed by another thread (meaning that tree w is currently being used to establish another augmenting path), then the atomic

Algorithm 4 Compute Multiple Augmenting Paths in Parallel

```
1:  select_tree[] ← array of atomic elements initialized to 0      ▷ store atomic attributes for trees
2:  procedure Parallel-Disjoint-Augmenting-Path(v, w, tree_v, tree_w, path_table)
3:      ▷ atomicCAS returns the value that exists prior to the swap
4:      if atomicCAS(select_tree[tree_v], 0, 1) = 0 then
5:          if atomicCAS(select_tree[tree_w], 0, 1) = 0 then
6:              P1 ← Even-Path(v, tree_v, path_table)
7:              P2 ← Even-Path(w, tree_w, path_table)
8:              P ← P1 + (v, w) + the reverse of P2
9:              return P
10:         else
11:             atomicCAS(select_tree[tree_v], 1, 0)
12:         end if
13:     end if
14:     return empty path
15: end procedure
```

attribute of tree v will be restored to its original value. This ensures that if other threads later require tree v to build an augmenting path, it remains accessible. No thread needs to wait for others, although very few paths may be missed due to access contention. Since the search procedure is called repeatedly until no augmenting path exists, any missed path will certainly be discovered in subsequent searches. By doing so, this lock-free approach maximizes the number of disjoint paths without blocking any thread, resulting in more efficient parallel processing and improved scalability.

Data races also occur when expanding trees in parallel. If there are two nodes v_1 and v_2 from different trees that satisfy the condition to expand the same matched edge (w, x), only one of them can add it. In the parallel algorithm, we can still use a lock-free method to resolve this conflict, as depicted in the first procedure in Algorithm 5. It does not matter which tree expands the edges, as long as the edges are added to the forest. The purpose of expanding is to introduce a new even node x for later checking. Placing x in different trees only generates different even-length paths. All even-length paths are functionally equivalent for building an augmenting path, because no matter the length of the augmenting path, it can only increase the matching size by one.

Algorithm 5 Expanding Trees and Finding Blossoms in Parallel

```
1:  select_edge[] ← array of atomic elements initialized to 0      ▷ attributes for matched edges
2:  procedure Parallel-Expand-Tree(v, w, tree_v, path_table, M, N_next)
3:      x ← the vertex matched with w in M
4:      i ← Min(w, x)                    ▷ the matched edge (v, w) can be indexed by one number
5:      if atomicCAS(select_edge[i], 0, 1) = 0 then
6:          add edges (v, w) and (w, x) to tree_v and add x to N_next
7:      end if
8:  end procedure
9:
10: select_odd_node[] ← array of atomic elements initialized to 0      ▷ attributes for odd nodes
11: procedure Parallel-Find-Blossom(v, w, tree_v, path_table, N_next)
12:     B ← blossom formed by (v, w) and edges on the path (v, ..., w) in tree_v
13:     for each node k in B do
14:         if k is an odd node ∧ atomicCAS(select_odd_node[k], 0, 1) = 0 then
15:             path_table[k] ← the even-length path from root(v) to k
16:             set k as an even node and add k to N_next
17:         end if
18:     end for
19: end procedure
```

In the third stage, when checking all unmatched edges in parallel to find blossoms, an odd node k may be contained in different blossoms. In this case, the node k should be treated as a new even node, and it may have multiple valid even-length paths corresponding to different blossom structures. Only one even-length path needs to be stored in the path table, as all such paths are equivalent for finding an augmenting path. Therefore, we also design a lock-free parallel approach as presented in the second procedure in Algorithm 5.

The work complexity of the parallel recursion-free algorithm is $O(V^2E)$, which is equivalent to that of the sequential recursion-free version because it does not involve additional operations with higher time complexity. Assuming the number of processors is p and the cost of an atomic CAS operation is c, the atomic operation cost when finding augmenting paths is $V \times c$, the cost when expanding trees is $E \times c$, and the cost when finding blossoms is $V \times c$. Thus, the parallel time complexity on p processors is $O(\frac{V^2E}{p} + (2V + E) \times c)$.

5.4 Discarding Alternating Tree

Challenge #3 highlights that when the blossom algorithm runs, the dynamic data structures, graphs and trees, are frequently modified. By removing the contraction and lifting processes, we have eliminated dynamic modifications to the graph, transforming the problem into one over a static graph. However, alternating trees are still continuously being accessed and expanded during computation. Building an augmenting path requires tracing the path from the root to the even node using breadth-first search, which introduces significant overheads, particularly for large trees. Furthermore, building and deconstructing these trees incurs extra system costs and results in irregular memory access. Hence, overcoming this challenge is also essential for implementing a practical parallel framework.

Inspired by the path table used to record even-length paths for odd nodes within the blossom, we develop an efficient implementation method that removes the necessity of constructing alternating trees. **Only** even-length alternating paths that start from the unmatched vertex are valid and useful for finding augmenting paths. Hence, instead of building alternating trees, our path table can record the valid even-length alternating path for every even node in the forest, regardless of whether it belongs to a blossom or not. To make it more memory-efficient, for each newly expanded even node, the path table only needs to record the path from its preceding even node to itself. And for each odd node within a blossom, the path table records the even-length path from the base to itself. By doing this, we can easily trace the even-length alternating path for any even node by simply looking up the path table, eliminating the need to establish trees and reducing the tracing cost of constructing an augmenting path.

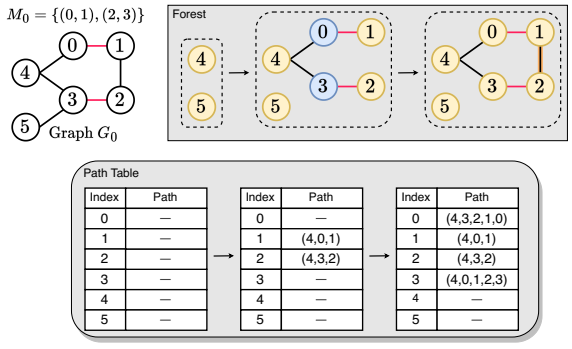


Figure 10: Only use path table to trace path

Figure 10 illustrates the same example as shown in the third row of Figure 4. In the basic blossom algorithm, two alternating trees rooted at the unmatched vertices 4 and 5 are constructed. Both 4 and 5 are even nodes without any preceding even node, so their paths do not need to be recorded. Next, the tree rooted at node 4 expands two branches, generating two new even nodes 1 and 2. The path from 4 to 1 is $(4, 0, 1)$, and the path from 4 to 2 is $(4, 3, 2)$. Therefore, these two paths are stored in the path table, indicating the even-length paths from the preceding even node to the current even node. Subsequently, the unmatched edge $(1, 2)$ is found, forming a blossom cycle. As a result, two odd nodes 0 and 3 are regarded as even nodes. Their even-length paths starting from the base are $(4, 3, 2, 1, 0)$ and $(4, 0, 1, 2, 3)$. Since node 3 becomes an even node and the unmatched edge $(3, 5)$ connects two even nodes from different trees, an augmenting path is identified. The path for node 5 in the table is empty, indicating that it is a root. The path for node 3 in the table is $(4, 0, 1, 2, 3)$, and node 4 is also a root. Appending the edge $(3, 5)$ to the path $(4, 0, 1, 2, 3)$ generates the augmenting path $(4, 0, 1, 2, 3, 5)$.

6 PERFORMANCE EVALUATION

6.1 Evaluation Platform Settings

6.1.1 Baselines. We have deployed the blossom algorithm from the widely recognized open-source C++ Boost Graph Library [17] as a sequential baseline. Additionally, we have utilized the implementation provided by the LEMON library [28], a well-known high-performance C++ library with highly tuned optimizations specialized for the blossom algorithm, as another strong and credible sequential baseline. To fairly compare the parallel performance, we have also deployed the most recently published parallel blossom implementation called Par-EB [87]. We have implemented our X-Blossom framework on multi-core platforms and evaluated the performance of all these implementations across various datasets.

6.1.2 Experimental Environment. All implementations are evaluated on a server with dual AMD EPYC 7643 processors (96 cores in total) and 256 GB of RAM. The latest versions of the Boost Graph Library (1.87.0) and the LEMON library (1.3.1) are used. All code is compiled using $O3$ optimizations, and the gcc version is 9.4.0.

6.1.3 Datasets. We have evaluated all implementations using both real-world datasets and synthetic datasets. A diverse collection of real-world graph datasets from the SNAP database [61] have been summarized in Table 1, which includes four main categories: social graphs (Twitter, Google, Twitch, YouTube, LiveJournal), economic networks (Amazon, CryptoTrans), communication graphs (HiggsNet, Wikipedia, StackOverflow), and reference networks (Hyperlink, Patent). These datasets span various domains and exhibit a wide range of vertex and edge sizes, providing a comprehensive and representative evaluation of performance under different scenarios.

Table 1: Statistics of real-world graphs

Graph	$\|V\|$	$\|E\|$	Description
Twitter [62]	81,306	1,342,296	Social network of Twitter users
Google [62]	107,614	12,238,285	Social network of Google users
Twitch [83]	168,114	6,797,557	Social network of Twitch gamers
Amazon [58]	334,863	925,872	Product co-purchasing network on Amazon
HiggsNet [26]	456,626	12,508,413	Communications graph of the Higgs boson
CryptoTrans [88]	781,896	2,674,043	Transactions network of six cryptocurrencies
YouTube [111]	1,134,890	2,987,624	Social network of YouTube users
Hyperlink [55]	1,791,489	25,444,207	Web graph of hyperlinks
Wikipedia [59]	2,394,385	4,659,565	Communications graph on Wikipedia
StackOverflow [61]	2,601,977	28,183,518	Communications graph on Stack Overflow
Patent [60]	3,774,768	16,518,947	Citation graph of U.S. patents
LiveJournal [61]	4,847,571	42,851,237	Social network of LiveJournal users

For synthetic datasets, we generate both large Erdős–Rényi random graphs [76, 79] and d-regular random graphs [15]. Erdős–Rényi random graphs are graphs where edges are assigned randomly with specific edge densities. The d-regular random graphs are graphs whose edges are assigned randomly to ensure each vertex is connected to exactly d other vertices. We also generate random graphs with gamma-distributed degrees to model more irregular graphs.

6.1.4 Evaluation Method. All graphs are represented in Compressed Sparse Row (CSR) format [24]. Each data points has been measured and averaged over 20 runs. For random graphs in synthetic datasets, we randomly generate 10 graphs for each specific category and measure the average time for computing the maximum matching.

6.2 Overall Performance

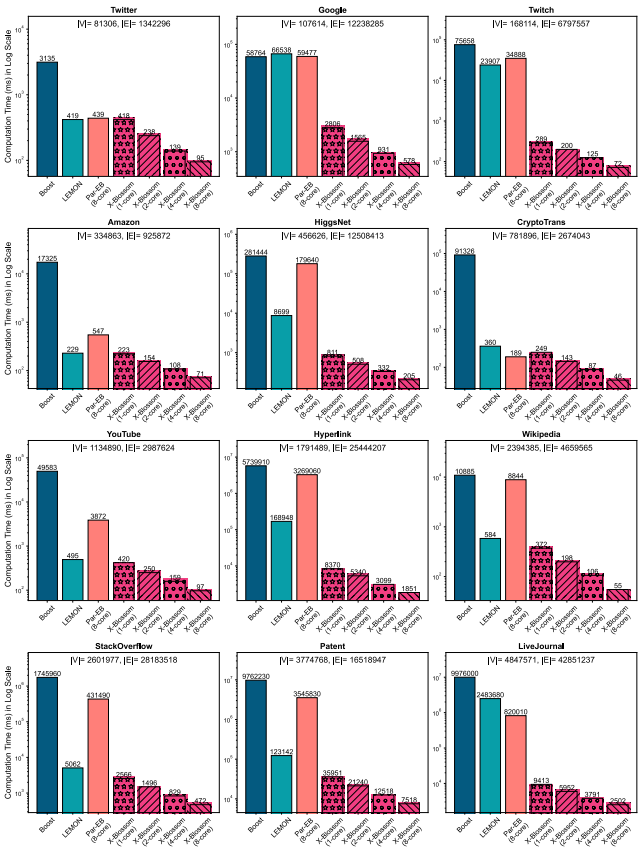


Figure 11: Computation time for maximum matching (in log scale) on real-world datasets

Figure 11 presents the computation time for all implementations on real-world datasets in **log** scale. Both Boost and LEMON serve as sequential benchmarks. The state-of-the-art parallel baseline Par-EB is tested using 8-core, and X-Blossom is evaluated from 1 core (sequential) to 8 cores. X-Blossom has dramatically outperformed all these baselines. Compared to the fastest sequential implementation LEMON, 8-core X-Blossom achieves a speedup of up to 992x. Moreover, it surpasses the state-of-the-art Par-EB with an average speedup of 431x. Notably, even 1-core X-Blossom exceeds 8-core Par-EB and all other sequential benchmarks in most cases. This outstanding performance is attributed to our strategy of identifying multiple disjoint augmenting paths in a single function call, which significantly reduces the computational overhead of repeated calls and tree reconstruction. Additionally, X-Blossom achieves up to 6.79x speedup as the number of cores increases from 1 to 8.

Par-EB [87] is a parallel implementation of blossom algorithm that only searches augmenting paths from checking nodes in parallel and does not address the fundamental issue of recursive data dependencies. Thus, in many cases, its 8-core performance is even much slower than the highly optimized sequential baseline LEMON.

6.3 Random Graphs with Specific Density

To evaluate the performance over synthetic datasets, we generate Erdős–Rényi random graphs with four specific densities from sparse to dense, and their vertex counts range from 400,000 to 1,200,000. Figure 12 presents the results on these random graphs for all implementations except Boost, which requires several hours even for the smallest sparse graph.

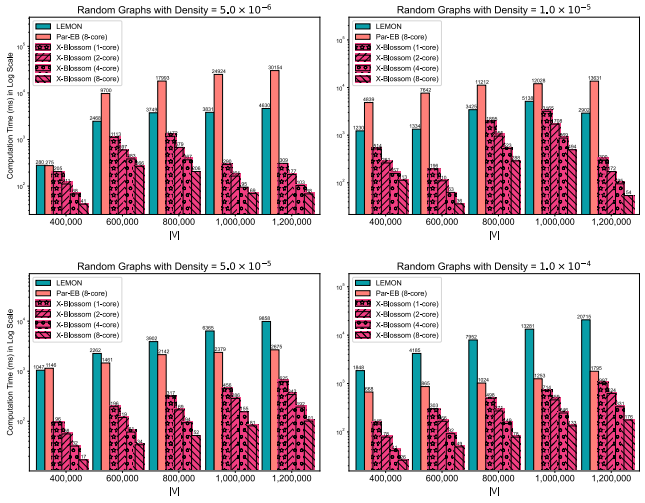


Figure 12: Computation time (in log scale) on random graphs of specific density and $|V|$ ranging from 400,000 to 1,200,000

As the density and vertex count increase, the speedup of 8-core X-Blossom over LEMON grows, reaching up to 117x for the largest and densest graph. This aligns with our analysis because X-Blossom can check all unmatched edges from all even nodes in parallel and return batches of disjoint valid paths concurrently. In sparse graphs, X-Blossom achieves a speedup of up to 442x compared to Par-EB, with both tested using 8 cores. In summary, X-Blossom demonstrates superior performance on both sparse and dense graphs, which highlights its efficiency and adaptability across diverse scenarios.

6.4 Random Graphs with Specific Degree

We also generate synthetic datasets of d-regular random graphs with four specific degrees ranging from 4 to 16, and vertex counts varying from 1,000,000 to 4,000,000. The evaluation results are demonstrated in Figure 13. With the increase in degree, each vertex in the graph has more edges that need to be checked, allowing the parallel mechanism in X-Blossom to further accelerate the process

of finding augmenting paths. Compared to LEMON, the speedup of 8-core X-Blossom increases progressively with higher degrees, achieving up to 56x. Meanwhile, the computation time for 8-core Par-EB is significantly longer than that of 1-core X-Blossom, due to its inefficient parallel computation and recursion procedures. Although both LEMON and Par-EB require recursive contraction and lifting of blossom cycles, LEMON's highly optimized implementation contributes to its performance. Overall, X-Blossom outperforms Par-EB with an average speedup of 232x by using 8 cores.

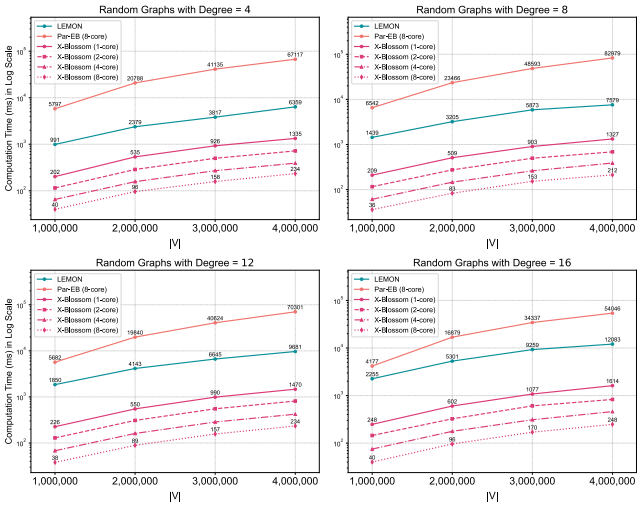


Figure 13: Computation time (in log scale) on random graphs of specific degree and $|V|$ ranging from 1,000,000 to 4,000,000

6.5 Graphs with Gamma-Distributed Degrees

To examine the effectiveness of X-Blossom on more irregular graph structures, we additionally generate random graphs whose vertex degrees follow a gamma distribution. The vertex counts range from 500,000 to 2,000,000, and two distinct gamma distributions are employed: the first with a shape parameter of 4 and a scale parameter of 1, and the second with a shape parameter of 2 and a scale parameter of 2, as presented in Figure 14.

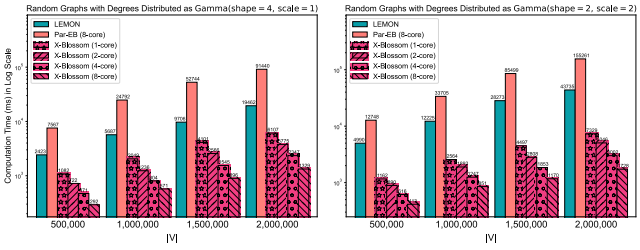


Figure 14: Computation time (in log scale) on random graphs with degrees following two distinct gamma distributions

When the vertex count increases, 8-core X-Blossom achieves up to 25x and 90x speedup compared to LEMON and 8-core Par-EB, respectively. The second gamma distribution has greater variation in vertex degrees—some nodes have extremely high degrees while others have low degrees—whereas the first distribution is more concentrated. This structural irregularity considerably degrades the performance of LEMON and Par-EB, resulting in increased processing time. In contrast, X-Blossom remains scalable and demonstrates strong robustness to irregular structures.

6.6 Memory Usage

Section 5.2 reveals that X-Blossom eliminates recursion and thereby avoids the memory overhead associated with recursive call stacks and tree rebuilding, which contributes to more efficient memory usage. In Figure 15, we compare the peak memory consumption of X-Blossom with that of other baselines on random graphs with two specific densities. The vertex sizes range from 400,000 to 1,200,000.

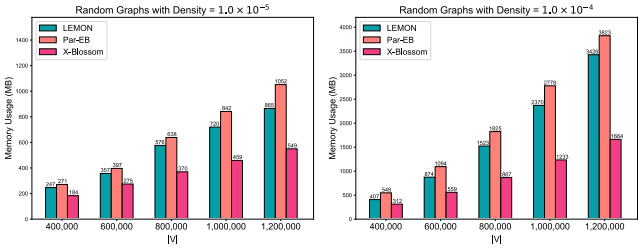


Figure 15: Memory usage comparison on random graphs of specific density and $|V|$ ranging from 400,000 to 1,200,000

The results indicate that X-Blossom consumes significantly less memory than LEMON and Par-EB, with reductions of up to 2.1x and 2.3x, respectively. Furthermore, its memory savings become increasingly pronounced as the vertex count grows and the graph density rises. Both LEMON and Par-EB involve deep recursive executions when processing large graphs, which impose a substantial memory burden. The memory efficiency of X-Blossom enhances its practicality in data-intensive applications.

6.7 Effectiveness of Path Table

As mentioned in Section 5.4, X-Blossom employs a path table to trace even-length alternating paths for all even nodes. We have compared the performance of 8-core X-Blossom with the path table to its performance using alternating trees on random graphs with a specific density. Figure 16 demonstrates that the 8-core X-Blossom with the path table outperforms the version using alternating trees by an average speedup of 3.3x. When finding augmenting paths, only even-length alternating paths need to be considered. A simple path table can thereby be effectively utilized to trace paths, achieving the same functionality as alternating trees. This optimization enhance the parallel performance and scalability of X-Blossom.

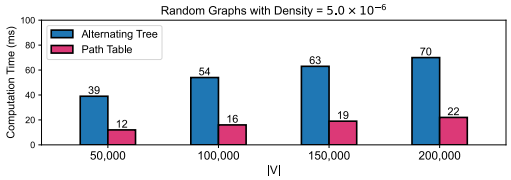


Figure 16: Performance comparison of 8-core X-Blossom: path table vs. alternating tree

6.8 Scalability Evaluation

To further evaluate the scalability of X-Blossom, we have tested its performance by applying an increasing number of cores from 1 to 64. Figure 17 presents the performance over four large random graphs with vertex counts ranging from 10,000,000 to 40,000,000 and a degree of 4. Both x-axis and y-axis are in log scale.

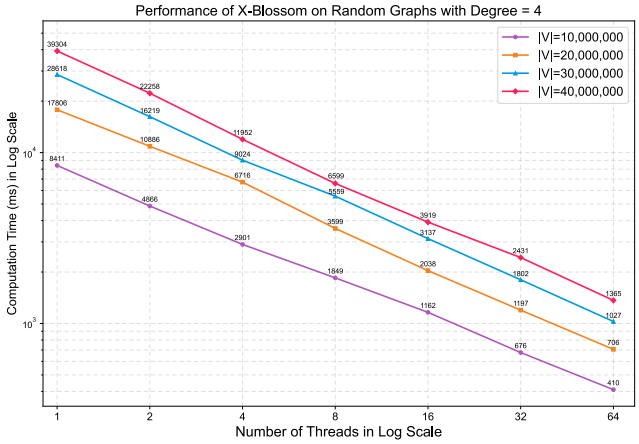


Figure 17: Computation time (in log scale) of X-Blossom with an increasing number of threads (in log scale)

When the number of threads doubles, the average speedup of X-Blossom is 1.72x, demonstrating the excellent scalability of our parallel framework. The four straight lines in the log-log plot intuitively illustrate this property. The speedup for 96 cores is consistent with the scalability line, but it has not been presented since 96 cores do not follow the doubling pattern of the number of threads in log scale. The notable scalability of X-Blossom highlights its critical role in computing the maximum matching for large graphs.

7 RELATED WORK

Matching is one of the most central problems in graph theory and finds extensive applications in various fields, including databases [53, 68, 108]. The blossom algorithm, proposed in [33], is the first polynomial-time algorithm to solve the maximum matching problem in general graphs. After that, dedicated research efforts [8, 10, 14, 35, 37–42, 48, 51, 54, 72, 92] have focused on improving the complexity of time and optimizing its implementations. Although some optimized algorithms [13, 86, 99] have achieved improved worst-case complexity, the basic algorithm remains the most widely used choice in common cases and practical applications [46, 47, 56, 109].

The research literature on parallelizing the blossom algorithm is scarce, due to its intricate execution patterns and inherently sequential nature. In 2016, Shoemaker et al. [93] made an attempt and proposed a solution that parallelizes the procedure of checking the edges for individual nodes. However, it does not address the key issue of recursive data dependencies, and all nodes are still processed sequentially. The most recent work is from [87], named Par-EB, which is a benchmark in the evaluation.

Parallel solutions not based on the blossom algorithm have also been explored. Azad et al. [6] presented a parallel approach utilizing a multi-source breadth-first search technique, but it is applicable only to bipartite graphs. Mulmuley et al. [73] proposed a randomized parallel solution for general graphs by reducing the problem to matrix inversion; however, it requires $O(V^{3.5}E)$ processors and relies on randomization. More recently, Abu-Khzam et al. [2] introduced a fixed-parameter parallel algorithm where the matching size must be provided as input, making it infeasible for large-scale graphs with unknown matching sizes. Another work by Israeli and Itai [50] presented a randomized parallel algorithm that operates in multiple phases to solve the maximal matching problem, which is computationally simpler than finding the maximum matching, as it does not involve careful handling of alternating trees or blossoms.

Graph algorithms are a critical research area with significant practical value [22, 23, 36, 63, 75, 85, 97, 101, 110]. However, parallelizing these algorithms remains challenging due to the irregular nature of graph data processing [20, 70, 105, 112]. Many prior research efforts have focused on addressing synchronization issues [67, 89, 100] and load imbalance [1, 18, 90]. Additionally, many studies have explored techniques to improve data locality [21, 49, 78] and memory usage [4, 106]. Other matching problems, such as subgraph matching [19, 52, 64, 77, 107] in graph databases and maximal matching [11, 29], have been well-studied and are distinct from the maximum matching problem addressed in this paper.

8 CONCLUSION

We have developed X-Blossom, a massively parallel computation framework for solving maximum matching in general graphs, with its implementation on multi-core processors. We have identified three fundamental issues in the blossom algorithm that block its parallel processing: recursive sequential data dependencies, inefficient path search, and dynamic data structure modifications. These challenges have persistently hindered the development of an efficient parallel solution since the basic algorithm was developed in 1965. By eliminating the recursive process of contracting and lifting blossom cycles, we have designed a sequential recursion-free algorithm that creates the critical opportunity for parallel computing. Building on this foundation, we propose an efficient parallel search strategy and develop a scalable parallel recursion-free algorithm. The lock-free synchronization mechanism further enhances parallel efficiency, and our optimization of the path table eliminates the use of alternating trees, leading to improved performance. Extensive evaluation results demonstrate that X-Blossom significantly outperforms all existing blossom solutions. We believe that X-Blossom, along with its open-source software, systematically addresses the structural issues of parallel processing for solving graph maximum matching problems. Our work paves the way for future research on developing a GPU-based implementation of the blossom algorithm and applying it as a subroutine to solve other essential matching problems, such as minimum-weight perfect matching.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. The work is supported in part by the U.S. National Science Foundation under grants MRI-2018627, CCF-2005884, CCF-2210753, CCF-2312507, and OAC-2310510.

REFERENCES

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 716–727. https://doi.org/10.1109/SC.2016.60

[2] Faisal N. Abu-Khzam, Shouwei Li, Christine Markarian, Friedhelm Meyer auf der Heide, and Pavel Podlipyan. 2019. Efficient parallel algorithms for parameterized problems. *Theor. Comput. Sci.* 786, C (Sept. 2019), 2–12. https://doi.org/10.1016/j.tcs.2018.11.006

[3] Navid Aghdaie, John Kolen, Mohamed Marwan Mattar, Mohsen Sardari, Su Xue, and Kazi Atif-Uz Zaman. 2018. Multiplayer video game matchmaking optimization. US Patent 9,993,735.

[4] Naheed Anjum Arafat, Arijit Khan, Arpit Kumar Rai, and Bishwamittra Ghosh. 2023. Neighborhood-Based Hypergraph Core Decomposition. *Proc. VLDB Endow.* 16, 9 (May 2023), 2061–2074. https://doi.org/10.14778/3598581.3598582

[5] Ariful Azad, Aydın Buluç, and Alex Pothen. 2017. Computing Maximum Cardinality Matchings in Parallel on Bipartite Graphs via Tree-Grafting. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 44–59. https://doi.org/10.1109/TPDS.2016.2546258

[6] Ariful Azad, Aydın Buluç, and Alex Pothen. 2017. Computing Maximum Cardinality Matchings in Parallel on Bipartite Graphs via Tree-Grafting. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 44–59. https://doi.org/10.1109/TPDS.2016.2546258

[7] Ariful Azad, Mahantesh Halappanavar, Sivasankaran Rajamanickam, Erik G. Boman, Arif Khan, and Alex Pothen. 2012. Multithreaded Algorithms for Maximum Matching in Bipartite Graphs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 860–872. https://doi.org/10.1109/IPDPS.2012.82

[8] Michel L Balinski. 1967. Labelling to obtain a maximum matching. In *Combinatorial Mathematics and Its Applications (Proceedings Conference Chapel Hill, North Carolina*. 585–602.

[9] Claude Berge. 1957. Two Theorems in Graph Theory. *Proceedings of the National Academy of Sciences* 43, 9 (1957), 842–844. https://doi.org/10.1073/pnas.43.9.842

[10] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2015. Deterministic fully dynamic data structures for vertex cover and matching. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Diego, California) *(SODA '15)*. Society for Industrial and Applied Mathematics, USA, 785–804.

[11] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 308–317. https://doi.org/10.1145/2312005.2312058

[12] Francis Bloch, Bhaskar Dutta, and Mihai Manea. 2019. Efficient partnership formation in networks. *Theoretical Economics* 14, 3 (2019), 779–811. https://doi.org/10.3982/TE3453

[13] Norbert Blum. 1990. A new approach to maximum matching in general graphs. In *Automata, Languages and Programming*, Michael S. Paterson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 586–597.

[14] Norbert Blum. 2015. Maximum Matching in General Graphs Without Explicit Consideration of Blossoms Revisited. arXiv:1509.04927 [cs.DS] https://arxiv.org/abs/1509.04927

[15] Béla Bollobás. 2001. Random Graphs. Cambridge University Press, Chapter 2.4. Random Regular Graphs.

[16] Angela Bonifati, M. Tamer Özsu, Yuanyuan Tian, Hannes Voigt, Wenyuan Yu, and Wenjie Zhang. 2024. The Future of Graph Analytics. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 544–545. https://doi.org/10.1145/3626246.3658369

[17] Boost.org. 2024. The Boost Graph Library. https://www.boost.org/doc/libs/1_87_0/libs/graph/doc/maximum_matching.html. Accessed: 2025-01-18.

[18] Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. 2024. PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware. 2, 3, Article 161 (May 2024), 25 pages. https://doi.org/10.1145/3654964

[19] Hongtai Cao, Qihao Wang, Xiaodong Li, Matin Najafi, Kevin Chen–Chuan Chang, and Reynold Cheng. 2024. Large Subgraph Matching: A Comprehensive and Efficient Approach for Heterogeneous Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2972–2985. https://doi.org/10.1109/ICDE60146.2024.00231

[20] Matteo Ceccarello, Carlo Fantozzi, Andrea Pietracaprina, Geppino Pucci, and Fabio Vandin. 2017. Clustering uncertain graphs. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 472–484. https://doi.org/10.1145/3186728.3164143

[21] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: an efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (April 2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[22] Xin Chen, Jieming Shi, You Peng, Wenqing Lin, Sibo Wang, and Wenjie Zhang. 2024. Minimum Strongly Connected Subgraph Collection in Dynamic Graphs. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1324–1336. https://doi.org/10.14778/3648160.3648173

[23] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. 2020. Finding k-shortest paths with limited overlap. *The VLDB Journal* 29, 5 (2020), 1023–1047. https://doi.org/10.1007/s00778-020-00604-x

[24] Wikipedia contributors. 2025. Sparse matrix. https://en.wikipedia.org/wiki/Sparse_matrix. Accessed: 2025-01-19.

[25] Margarida Corominas-Bosch. 2004. Bargaining in a network of buyers and sellers. *Journal of Economic Theory* 115, 1 (2004), 35–77. https://doi.org/10.1016/S0022-0531(03)00110-8

[26] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. 2013. The Anatomy of a Scientific Rumor. *Scientific Reports* 3, 1 (2013), 2980. https://doi.org/10.1038/srep02980

[27] Mehmet Deveci, Kamer Kaya, Bora Ucar, and Umit V. Catalyurek. 2013. GPU accelerated maximum cardinality matching algorithms for bipartite graphs. arXiv:1303.1379 [cs.DC] https://arxiv.org/abs/1303.1379

[28] Balázs Dezső, Alpár Jüttner, and Péter Kovács. 2011. LEMON – an Open Source C++ Graph Template Library. *Electronic Notes in Theoretical Computer Science* 264, 5 (2011), 23–45. https://doi.org/10.1016/j.entcs.2011.06.003 Proceedings of the Second Workshop on Generative Technologies (WGT) 2010.

[29] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (April 2021), 70 pages. https://doi.org/10.1145/3434393

[30] Loc Do, Hady W. Lauw, and Ke Wang. 2015. Mining revenue-maximizing bundling configuration. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 593–604. https://doi.org/10.14778/2735479.2735491

[31] Gabriel Cristian Dragomir-Loga and Marius Pop. 2021. Blossom Algorithm Analysis in a Medical Emergency Management System. In *2021 International Conference on e-Health and Bioengineering (EHB)*. 1–4. https://doi.org/10.1109/EHB52898.2021.9657586

[32] Jack Edmonds. 1965. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics* (1965), 125. https://api.semanticscholar.org/CorpusID:15379135

[33] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17 (1965), 449–467.

[34] Yuval Emek, Shay Kutten, and Roger Wattenhofer. 2016. Online matching: haste makes waste!. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing* (Cambridge, MA, USA) *(STOC '16)*. Association for Computing Machinery, New York, NY, USA, 333–344. https://doi.org/10.1145/2897518.2897557

[35] S. Even and O. Kariv. 1975. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. 100–112. https://doi.org/10.1109/SFCS.1975.5

[36] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2019. Effective and Efficient Community Search Over Large Directed Graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2019), 2093–2107. https://doi.org/10.1109/TKDE.2018.2872982

[37] Harold N. Gabow. 1976. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *J. ACM* 23, 2 (April 1976), 221–234. https://doi.org/10.1145/321941.321942

[38] Harold N. Gabow. 1990. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, USA) *(SODA '90)*. Society for Industrial and Applied Mathematics, USA, 434–443.

[39] Harold N. Gabow, Zvi Galil, and Thomas H. Spencer. 1989. Efficient implementation of graph algorithms using contraction. *J. ACM* 36, 3 (July 1989), 540–572. https://doi.org/10.1145/65950.65954

[40] Harold N. Gabow and Robert E. Tarjan. 1991. Faster Scaling Algorithms for General Graph Matching Problems. *J. ACM* 38, 4 (Oct. 1991), 815–853. https://doi.org/10.1145/115234.115366

[41] Zvi Galil. 1986. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.* 18, 1 (March 1986), 23–38. https://doi.org/10.1145/6462.6502

[42] Zvi Galil, Silvio Micali, and Harold Gabow. 1986. An $O(EVlogV)$ Algorithm for Finding a Maximal Weighted Matching in General Graphs. *SIAM J. Comput.* 15, 1 (1986), 120–130. https://doi.org/10.1137/0215009 arXiv:https://doi.org/10.1137/0215009

[43] Zengyang Gong, Yuxiang Zeng, and Lei Chen. 2024. Real-Time Insertion Operator for Shared Mobility on Time-Dependent Road Networks. *Proc. VLDB Endow.* 17, 7 (March 2024), 1669–1682. https://doi.org/10.14778/3654621.3654633

[44] Jiabao Han and Hongzhi Wang. 2021. Graph matching based reasoner: A symbolic approach to question answering. *Engineering Applications of Artificial Intelligence* 105 (2021), 104425. https://doi.org/10.1016/j.engappai.2021.104425

[45] Jiawei He, Zehao Huang, Naiyan Wang, and Zhaoxiang Zhang. 2021. Learnable Graph Matching: Incorporating Graph Partitioning With Deep Feature Learning

for Multiple Object Tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 5299–5309.

[46] Oscar Higgott. 2022. PyMatching: A Python Package for Decoding Quantum Codes with Minimum-Weight Perfect Matching. *ACM Transactions on Quantum Computing* 3, 3, Article 16 (June 2022), 16 pages. https://doi.org/10.1145/3505637

[47] Oscar Higgott and Craig Gidney. 2023. Sparse Blossom: correcting a million errors per core second with minimum-weight matching. arXiv:2303.15933 [quant-ph] https://arxiv.org/abs/2303.15933

[48] John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 2, 4 (1973), 225–231. https://doi.org/10.1137/0202019

[49] Lin Hu, Yinnian Lin, Lei Zou, and M. Tamer Özsu. 2024. A graph pattern mining framework for large graphs on GPU. *The VLDB Journal* 34, 1 (2024), 6. https://doi.org/10.1007/s00778-024-00883-8

[50] Amos Israeli and A. Itai. 1986. A fast and simple randomized parallel algorithm for maximal matching. *Inform. Process. Lett.* 22, 2 (1986), 77–80. https://doi.org/10.1016/0020-0190(86)90144-4

[51] Taisuke Izumi, Naoki Kitamura, and Yutaro Yamaguchi. 2024. A Nearly Linear-Time Distributed Algorithm for Exact Maximum Matching. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*. SIAM, 4062–4082. https://doi.org/10.1137/1.9781611977912.141

[52] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *Proc. VLDB Endow.* 16, 7 (March 2023), 1699–1711. https://doi.org/10.14778/3587136.3587144

[53] Marek Karpinski and Wojciech Rytter. 1998. *Fast Parallel Algorithms for Graph Matching Problems*. Oxford University Press. https://doi.org/10.1093/oso/9780198501626.001.0001

[54] Naoki Kitamura and Taisuke Izumi. 2022. A subquadratic-time distributed algorithm for exact maximum matching. *IEICE Transactions on Information and Systems* 105, 3 (2022), 634–645.

[55] Christine Klymko, David Gleich, and Tamara G. Kolda. 2014. Using Triangles to Improve Community Detection in Directed Networks. arXiv:1404.5874 [cs.SI] https://arxiv.org/abs/1404.5874

[56] Vladimir Kolmogorov. 2009. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation* 1, 1 (2009), 43–67. https://doi.org/10.1007/s12532-009-0002-8

[57] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[58] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM Trans. Web* 1, 1 (May 2007), 5–es. https://doi.org/10.1145/1232722.1232727

[59] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 641–650. https://doi.org/10.1145/1772690.1772756

[60] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (Chicago, Illinois, USA) *(KDD '05)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/1081870.1081893

[61] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. Accessed: 2025-01-19.

[62] Jure Leskovec and Julian Mcauley. 2012. Learning to Discover Social Circles in Ego Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/7a614fd06c325499f1680b9896beedeb-Paper.pdf

[63] Jia Li, Wenyue Zhao, Nikos Ntarmos, Yang Cao, and Peter Buneman. 2023. MITra: A Framework for Multi-Instance Graph Traversal. *Proc. VLDB Endow.* 16, 10 (June 2023), 2551–2564. https://doi.org/10.14778/3603581.3603594

[64] Siyu Li, Zhiwei Zhang, Meihui Zhang, Ye Yuan, and Guoren Wang. 2024. Authenticated Subgraph Matching in Hybrid-Storage Blockchains. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 1986–1998. https://doi.org/10.1109/ICDE60146.2024.00159

[65] Wuyang Li, Xinyu Liu, and Yixuan Yuan. 2022. SIGMA: Semantic-Complete Graph Matching for Domain Adaptive Object Detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 5291–5300.

[66] Yuchen Li, Dongxiang Zhang, and Kian-Lee Tan. 2015. Real-time targeted influence maximization for online advertisements. *Proc. VLDB Endow.* 8, 10 (June 2015), 1070–1081. https://doi.org/10.14778/2794367.2794376

[67] Qing Liu, Xuankun Liao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2023. Distributed (,)-Core Decomposition over Bipartite Graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 909–921. https://doi.org/10.1109/ICDE55515.2023.00075

[68] Cheng Long, Raymond Chi-Wing Wong, Philip S. Yu, and Minhao Jiang. 2013. On optimal worst-case matching. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 845–856. https://doi.org/10.1145/2463676.2465321

[69] Bo Lu, Robert Greevy, Xiaochun Xu, and Curt Beck. 2011. Optimal Nonbipartite Matching and Its Statistical Applications. *The American Statistician* 65, 1 (2011), 21–30. https://doi.org/10.1198/tast.2011.08294

[70] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2024. A Survey of Distributed Graph Algorithms on Massive Graphs. *ACM Comput. Surv.* 57, 2, Article 27 (Oct. 2024), 39 pages. https://doi.org/10.1145/3694966

[71] Panayotis Mertikopoulos, Heinrich H. Nax, and Bary S.R. Pradelski. 2024. Quick or cheap? Breaking points in dynamic markets. *Journal of Mathematical Economics* 112 (2024), 102987. https://doi.org/10.1016/j.jmateco.2024.102987

[72] Silvio Micali and Vijay V. Vazirani. 1980. An $O(\sqrt{V}E)$ Algorithm for Finding Maximum Matching in General Graphs. In *21st Annual Symposium on Foundations of Computer Science (FOCS), Syracuse, New York, USA, 13-15 October 1980*. IEEE Computer Society, 17–27. https://doi.org/10.1109/SFCS.1980.12

[73] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. 1987. Matching is as easy as matrix inversion. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 345–354. https://doi.org/10.1145/28395.383347

[74] Farid M. Naini, Jayakrishnan Unnikrishnan, Patrick Thiran, and Martin Vetterli. 2016. Where You Are Is Who You Are: User Identification by Matching Statistics. *Trans. Info. For. Sec.* 11, 2 (Feb. 2016), 358–372. https://doi.org/10.1109/TIFS.2015.2498131

[75] Lutz Oettershagen, Honglian Wang, and Aristides Gionis. 2024. Finding Densest Subgraphs with Edge-Color Constraints *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 936–947. https://doi.org/10.1145/3589334.3645647

[76] Erdős P. and Rényi A. 1959. On random graphs I. *Publicationes Mathematicae* 6, 290-297 (1959), 18.

[77] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1099–1114. https://doi.org/10.1145/3318464.3389702

[78] Peng Peng, Shengyi Ji, Zhen Tian, Hongbo Jiang, Weiguo Zheng, and Xuecang Zhang. 2023. Locality Sensitive Hashing for Optimizing Subgraph Query Processing in Parallel Computing Systems. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 1885–1896. https://doi.org/10.1145/3580305.3599419

[79] Chengzhi Piao, Weiguo Zheng, Yu Rong, and Hong Cheng. 2020. Maximizing the reduction ability for near-maximum independent set computation. *Proc. VLDB Endow.* 13, 12 (July 2020), 2466–2478. https://doi.org/10.14778/3407790.3407838

[80] James S. Plank. 2017. Edmonds' General Matching Algorithm (The Blossom Algorithm). https://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/notes/Edmonds/. Accessed: 2025-01-28.

[81] Ignacio Rios and Alfredo Torrico. 2024. Platform Design in Curated Dating Markets. arXiv:2308.02584 [math.OC] https://arxiv.org/abs/2308.02584

[82] Alvin E. Roth, Tayfun Sönmez, and M. Utku Ünver. 2005. Pairwise kidney exchange. *Journal of Economic Theory* 125, 2 (2005), 151–188. https://doi.org/10.1016/j.jet.2005.04.004

[83] Benedek Rozemberczki and Rik Sarkar. 2021. Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings. arXiv:2101.03091 [cs.SI] https://arxiv.org/abs/2101.03091

[84] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 420–431. https://doi.org/10.1145/3186728.3164139

[85] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (Aug. 2021), 62–71. https://doi.org/10.1145/3434642

[86] Gregory Schwing, Daniel Grosu, and Loren Schwiebert. 2024. Parallel Maximum Cardinality Matching for General Graphs on GPUs. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 880–889.

https://doi.org/10.1109/IPDPSW63119.2024.00157

[87] Gregory Schwing, Daniel Grosu, and Loren Schwiebert. 2024. Shared-Memory Parallel Edmonds Blossom Algorithm for Maximum Cardinality Matching in General Graphs. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 530–539. https://doi.org/10.1109/IPDPSW63119.2024.00107

[88] Kiarash Shamsi, Friedhelm Victor, Murat Kantarcioglu, Yulia Gel, and Cuneyt G Akcora. 2022. Chartalist: Labeled Graph Datasets for UTXO and Account-based Blockchains. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., 34926–34939.

[89] Yingxia Shao, Lei Chen, and Bin Cui. 2014. Efficient cohesive subgraphs detection in parallel. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/2588555.2593665

[90] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 625–636. https://doi.org/10.1145/2588555.2588557

[91] Suraj Shetiya, Ian P. Swift, Abolfazl Asudeh, and Gautam Das. 2024. Shapley Values for Explanation in Two-sided Matching Applications. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*. OpenProceedings.org, 584–596. https://doi.org/10.48786/EDBT.2024.50

[92] Baruch Shieber and Shlomo Moran. 1986. Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: maximum matchings. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada) *(PODC '86)*. Association for Computing Machinery, New York, NY, USA, 282–292. https://doi.org/10.1145/10590.10615

[93] Amy Shoemaker and Sagar Vare. 2016. Edmonds' blossom algorithm. https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/shoemaker_vare.pdf. Accessed: 2025-01-03.

[94] Lawrence V Snyder and Zuo-Jun Max Shen. 2019. The Traveling Salesman Problem. In *Fundamentals of Supply Chain Theory*. John Wiley Sons, Ltd, Chapter 10, 403–461. https://doi.org/10.1002/9781119584445.ch10

[95] Yixin Su, Rui Zhang, Sarah M. Erfani, and Junhao Gan. 2021. Neural Graph Matching based Collaborative Filtering. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) *(SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 849–858. https://doi.org/10.1145/3404835.3462833

[96] Amirmahdi Tafreshian and Neda Masoud. 2020. Using subsidies to stabilize peer-to-peer ridesharing markets with role assignment. *Transportation Research Part C: Emerging Technologies* 120 (2020), 102770. https://doi.org/10.1016/j.trc.2020.102770

[97] Nguyen Thanh Tam, Matthias Weidlich, Bolong Zheng, Hongzhi Yin, Nguyen Quoc Viet Hung, and Bela Stantic. 2019. From anomaly detection to rumour detection using data streams of social platforms. *Proc. VLDB Endow.* 12, 9 (May 2019), 1016–1029. https://doi.org/10.14778/3329772.3329778

[98] Vijay V. Vazirani. 2013. A Simplification of the MV Matching Algorithm and its Proof. arXiv:1210.4594 [cs.DS] https://arxiv.org/abs/1210.4594

[99] Vijay V. Vazirani. 2020. A Proof of the MV Matching Algorithm. arXiv:2012.03582 [cs.DS] https://arxiv.org/abs/2012.03582

[100] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3293883.3295733

[101] Libin Wang and Raymond Chi-Wing Wong. 2024. PCSP: Efficiently Answering Label-Constrained Shortest Path Queries in Road Networks. *Proc. VLDB Endow.* 17, 11 (July 2024), 3082–3094. https://doi.org/10.14778/3681954.3681985

[102] Yongxin Wang, Kris Kitani, and Xinshuo Weng. 2021. Joint Object Detection and Multi-Object Tracking with Graph Neural Networks. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 13708–13715. https://doi.org/10.1109/ICRA48506.2021.9561110

[103] Yansheng Wang, Yongxin Tong, Cheng Long, Pan Xu, Ke Xu, and Weifeng Lv. 2019. Adaptive Dynamic Bipartite Graph Matching: A Reinforcement Learning Approach. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1478–1489. https://doi.org/10.1109/ICDE.2019.00133

[104] Yuandong Wang, Hongzhi Yin, Lian Wu, Tong Chen, and Chunyang Liu. 2023. Secure Your Ride: Real-Time Matching Success Rate Prediction for Passenger-Driver Pairs. *IEEE Transactions on Knowledge and Data Engineering* 35, 3 (2023), 3059–3071. https://doi.org/10.1109/TKDE.2021.3112739

[105] Frank Wanye, Vitaliy Gleyzer, and Wu-chun Feng. 2019. Fast Stochastic Block Partitioning via Sampling. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2019.8916542

[106] Manuel Widmoser, Daniel Kocher, and Nikolaus Augsten. 2024. Scalable Distributed Inverted List Indexes in Disaggregated Memory. *Proc. ACM Manag. Data* 2, 3, Article 171 (May 2024), 27 pages. https://doi.org/10.1145/3654974

[107] Leonard Wörteler, Moritz Renftle, Theodoros Chondrogiannis, and Michael Grossniklaus. 2022. Cardinality Estimation using Label Probability Propagation for Subgraph Matching in Property Graph Databases. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:247802252

[108] Michael M. Wu and Michael C. Loui. 1990. An efficient distributed algorithm for maximum matching in general graphs. *Algorithmica* 5, 1 (1990), 383–406. https://doi.org/10.1007/BF01840395

[109] Yue Wu and Lin Zhong. 2023. Fusion Blossom: Fast MWPM Decoders for QEC . In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 928–938. https://doi.org/10.1109/QCE57702.2023.00107

[110] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. 2024. Efficient and effective algorithms for densest subgraph discovery and maintenance. *The VLDB Journal* 33, 5 (2024), 1427–1452. https://doi.org/10.1007/s00778-024-00855-y

[111] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213. https://doi.org/10.1007/s10115-013-0693-z

[112] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. 2020. Distributed subgraph counting: a general approach. *Proc. VLDB Endow.* 13, 12 (July 2020), 2493–2507. https://doi.org/10.14778/3407790.3407840

[113] Kaiqi Zhao, Gao Cong, Jin-Yao Chin, and Rong Wen. 2019. Exploring market competition over topics in spatio-temporal document collections. *The VLDB Journal* 28, 1 (2019), 123–145. https://doi.org/10.1007/s00778-018-0522-9

[114] Yingli Zhou, Yixiang Fang, Chenhao Ma, Tianci Hou, and Xin Huang. 2024. Efficient Maximal Motif-Clique Enumeration over Large Heterogeneous Information Networks. *Proc. VLDB Endow.* 17, 11 (July 2024), 2946–2959. https://doi.org/10.14778/3681954.3681975