Utrecht University

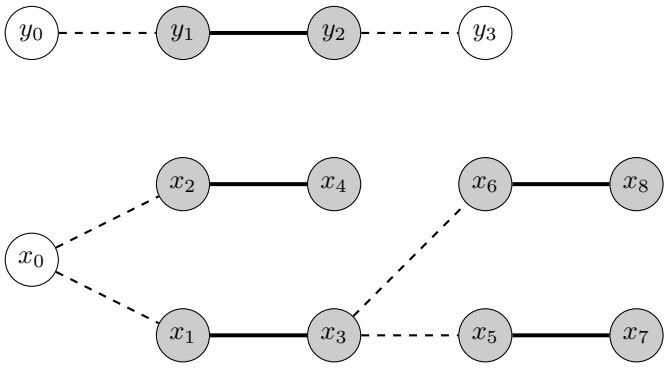
Faculty of Science

Pruning of alternating-path trees for bipartite graph matching

Bachelor Thesis

J. Heuseveldt

Mathematics



Supervisor:

Prof. dr. R.H. Bisseling

June 11, 2019

Abstract

To compute a maximum matching in a bipartite graph we can use algorithms based on augmenting paths. If a search for an augmenting path is unsuccessful, the information gained by the search is often discarded. However, it is possible to use this information to speed up future searches. In this thesis we will formally prove that it is possible to prune exhausted parts of the graph, for both single-source (SS) and multi-source (MS) breadth-first-search (BFS) algorithms. Our implementation gives a speedup for the SS-BFS algorithm. Due to additional overhead, the MS-BFS algorithm is more often slowed down by the pruning rather than sped up.

Contents

List of Algorithms

1 Introduction

Before being able to understand what matching algorithms do, we need to understand the problem that it solves and the terminology used to describe it.

A **bipartite graph** $G = (V, E)$ is a set of **vertices** V and a set of **edges** E. Each edge connects two vertices and two edges are **connected** if they share a vertex. The set of vertices V is split into two sets of vertices L and R, such that $L \cup R = V, L \cap R = \emptyset$ and each edge e can be written as $\{v_1, v_2\}$ such that $v_1 \in L$ and $v_2 \in R$. The **degree** of a vertex is defined as the number of edges that have that vertex as one of their endpoints.

For such graphs, we define a **matching** to be a set $M \subset E$ such that no two edges are connected. A **maximal matching** is a matching such that for each edge $e \in E \setminus M$, there exists an $m \in M$ such that e and m are connected. It is therefore impossible to find a matching that strictly contains a maximal matching. A **maximum matching** is a (maximal) matching such that no (maximal) matching with greater cardinality exists. We call a vertex matched or unmatched if it is the endpoint of respectively any or no edge in M. Similarly, an edge is called (un)matched if it is (not) in M. **Flipping** an edge e means taking it out of M if $e \in M$, and adding it to M otherwise.

In a graph, we can make paths. A **path** is a sequence of edges $e_1, \ldots, e_n$ such that each edge is connected to the next one, and vertex-disjoint elsewhere. A path can also be represented by a sequence of vertices $x_1, \ldots, x_{n+1}$ such that $e_i = \{x_i, x_{i+1}\}$ for all $i \in \{1, \ldots, n\}$. A **cycle** is a path of at least 3 edges where also the first and last edge are connected. Note that in bipartite graphs, cycles are always of even length. An M**-alternating path** is a path where every other edge is matched, with respect to a matching M. Similarly, we define an M**-alternating cycle** to be a cycle where every other edge is matched, with respect to M. An M**-augmenting path** is an M-alternating path of odd length, starting and ending in an unmatched vertex. An augmenting path thus consists of more unmatched than matched edges, so flipping all edges of an augmenting path results in a new matching of strictly higher cardinality. According to Berge's theorem [2], a matching is maximum if no augmenting path exists. I will show a proof of Berge's theorem in the next section.

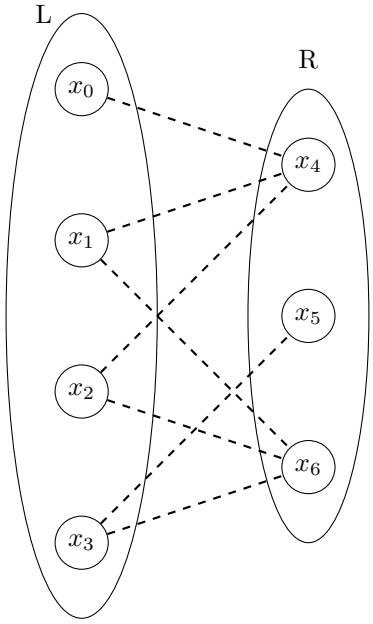


Figure 1: A bipartite graph

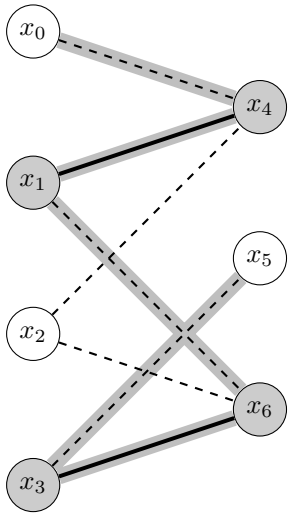


Figure 2: A maximal matching and an augmenting path

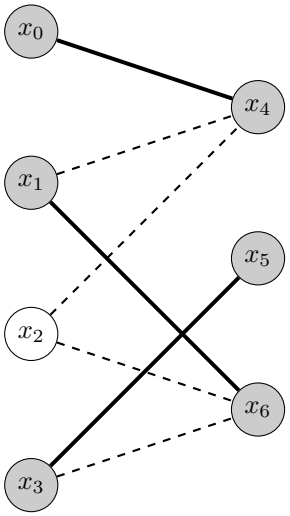


Figure 3: A maximum matching obtained by flipping the augmenting path in figure 2

2 Finding maximum matchings

There are multiple algorithms for finding maximum matchings in bipartite graphs. They can roughly be divided into several classes, with the main classes being based on augmenting paths, auctions or push-relabel. In this thesis, we will only look at algorithms based on augmenting paths. These algorithms usually try to find an augmenting path, add it to the matching and repeat.

Algorithm 1 Maximum matching algorithm based on augmenting paths

```
1: M ← ∅
2: p ← any augmenting path
3: while p exists do
4:     flip all edges in p
5:     p ← any augmenting path
```

Although it is easy to see that this leads to a maximal matching, we still need to prove that this indeed leads to a maximum matching.

Lemma 2.1. *Flipping all edges of an alternating path or cycle strictly increases the size of a matching if and only if it is an augmenting path.*

Proof. Suppose we have a graph G, a matching M and an M-augmenting path P. We need to prove that the new matching $M' = M \oplus P = (M \setminus P) \cup (P \setminus M) = (M \cup P) \setminus (M \cap P)$ is a valid matching of strictly greater cardinality than M.

Since P is an M-augmenting path, we know that it is an alternating path starting and ending in an unmatched vertex. Suppose we have a vertex x. If $x \notin P$, x is still matched at most once, like it was with respect to M. If x is an endpoint of P, then x was unmatched in M, and is matched exactly once in M'. Otherwise, x lies somewhere in the middle of P, which means that it was matched in M, and is matched to another vertex in M'. In all cases, x is matched at most once with respect to M', which means M' is a valid matching.

As P is an M-augmenting path, P consists of $2n + 1$ edges, with $n \in \mathbb{Z}_{\geq 0}$. Of these $2n + 1$ edges, n are matched in M and the other $n + 1$ are matched in M', hence $|M'| = |M| + 1$.

Now suppose we have an M-alternating path or cycle P, such that $M' = M \oplus P$ is a valid matching and $|M'| > |M|$. For the cardinality of M to increase by flipping P, P must contain more unmatched edges than matched edges. Because P is an alternating path or cycle, this can only be the case if $|P \setminus M| = |P \cap M| + 1$. Therefore $|P|$ is odd and P can not be an alternating cycle. It follows that P must be a path and both the first and last edge must be unmatched with respect to M. For M' to be a valid matching, the first and last vertex of P must also be unmatched with respect to M. It follows that P is an M-augmenting path. □

Lemma 2.2. *The symmetric difference between two matchings only consists of alternating paths and cycles with respect to either of those matchings.*

Proof. Suppose we have matchings M_1 and M_2. Denote the symmetric difference, $M_1 \oplus M_2 = (M_1 \cup M_2) \setminus (M_2 \cap M_1)$, by $\mathbb{P}$. Since M_1 and M_2 are both valid matchings, the degree of each vertex is at most 1 in M_1 and M_2. It follows that the degree of a vertex in $\mathbb{P}$ is at most 2. Therefore, $\mathbb{P}$ only consists of paths and cycles.

Suppose we have a vertex x that is contained in a path or cycle, but is not an endpoint of a path. Each vertex is incident to at most one edge in a matching. It follows that x must be the endpoint of one edge in M_1 and one edge in M_2. Therefore all paths and cycles in $\mathbb{P}$ are alternating paths and cycles in a graph with either M_1 or M_2. □

Theorem 2.3 (Berge's Theorem [2])**.** *Algorithm 1 results in a maximum matching.*

Proof. Suppose we have a matching M_1 generated by algorithm 1 that is not a maximum matching. This means there are no augmenting paths and there is a maximum matching M_m such that $|M_m| > |M_1|$.

Let $\mathbb{P}$ be the symmetric difference $M_m \oplus M_1 = (M_m \setminus M_1) \cup (M_1 \setminus M_m)$. By flipping all edges in $\mathbb{P}$, we can take M_1 to M_m. By lemma 2.2, $\mathbb{P}$ consists of only alternating paths and cycles. For the cardinality to increase by flipping all paths and cycles (and thus all edges) in $\mathbb{P}$, there must be at least one path that is an augmenting path (lemma 2.1), contradicting that M_1 has no augmenting path. We conclude that M_1 must be a maximum matching. □

Within the class of augmenting path-based algorithms, we can distinguish two kinds of algorithms: single-source (SS) algorithms and multi-source (MS) algorithms. SS algorithms search for an augmenting path from a single unmatched vertex. Most MS algorithms search from all unmatched vertices at one side of the graph instead and usually find multiple augmenting paths in the same round. This makes it easier to run MS algorithms in parallel, but it is often harder to prove useful properties about MS algorithms.

3 Single-source alternating-path trees

What remains is how to efficiently find augmenting paths. Before jumping to MS algorithms, I will first prove a useful theorem for SS algorithms.

Algorithm 2 SS algorithm for finding maximum matchings in bipartite graphs

```
1: function SS-MATCH(V = L ∪ R, E)
2:     M ← ∅
3:     for all y ∈ V do visited[y] ← false
4:     for all x ∈ L do
5:         if x unmatched then
6:             P ← SS-SEARCH(V, E, x, M)
7:             for all y ∈ V visited in the last search do visited[y] ← false
8:             M ← M ⊕ P
9: function SS-SEARCH(V, E, x_0, M)
10:    Build an M-alternating-path tree T(x_0) using breadth first search (BFS) rooted at x_0, ignoring
       vertices y for which visited[y] is true. For every traversed vertex y, visited[y] is set to true. Return once
       an augmenting path has been found. Otherwise return ∅.
```

Theorem 3.1 ([1])**.** *If, in algorithm 2, we find no augmenting path in line 6, we can prune the entire searched tree, ignoring all of its vertices in all future searches, and skip line 7.*

The `SS-search` function in algorithm 2 builds an M-alternating-path tree, given a matching M. Let x_0 be the root of the M-alternating-path tree generated in `SS-search` and $T(x)$ the subtree of $T(x_0)$ rooted at x. We divide $T(x_0)$ in layers. A layer n contains all vertices with distance n to x_0 in $T(x_0)$, as can be seen in figure 5. We denote the layer of a vertex x by L_x. Since the tree consists of alternating paths increasing in layer number and the root is unmatched, for $i \in \mathbb{Z}_{\geq 0}$, all edges between layer $2i$ and $2i+1$ are unmatched and all edges between layer $2i+1$ and $2i+2$ are matched. Because we have a bipartite graph, all vertices at even levels belong to L and all vertices at odd levels belong to R.

Lemma 3.2. *Let P be an augmenting path. If no augmenting path is found during the construction of $T(x_0)$, then P and $T(x_0)$ are vertex disjoint.*

Proof. Since x_0 is unmatched, any augmenting path that contains x_0 has x_0 as one of the endpoints. Because we did not find an augmenting path during the construction of $T(x_0)$, there exists no augmenting path starting at x_0. It follows that P has both endpoints, which must be unmatched, outside of $T(x_0)$.

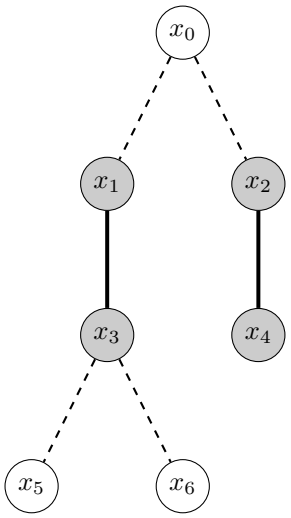


Figure 4: An alternating-path tree $T(x_0)$ with two augmenting paths

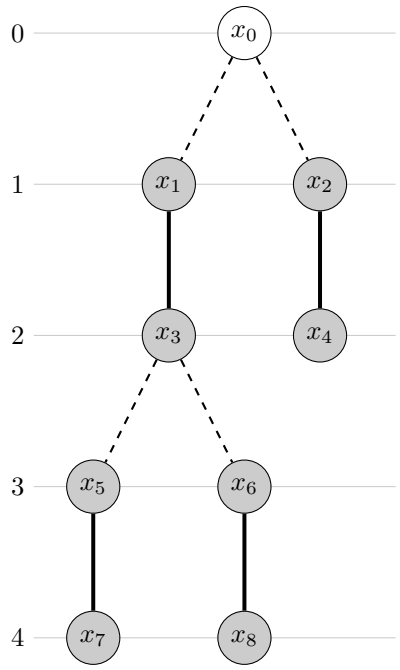


Figure 5: A layered alternating-path tree without augmenting paths

Let u be the first vertex of P contained in $T(x_0)$ and v the last vertex of P in $T(x_0)$. By construction of $T(x_0)$, both u and v are incident to a matched edge within $T(x_0)$. The edges in P preceding u and following v must therefore be unmatched. Since there's an odd number of edges in P between u and v, either $u \in L$ or $v \in L$.

Suppose $u \in L$, then L_u is even and the alternating path P_2 from u to x_0 through $T(x_0)$ starts with a matched edge. Let P_1 be the part of P until u, starting and ending with an unmatched edge. Consider the path $P' = P_1 \cup P_2$. It is an alternating path starting in the same vertex as P and ending in x_0, which are both unmatched. It follows that P' is an augmenting path with x_0 as an endpoint, contradicting that there is no augmenting path with x_0 as an endpoint.

The case $v \in L$ is analogous to the case $u \in L$, as can be seen by reversing P.

We conclude that P and $T(x_0)$ are vertex disjoint. □

Proof of theorem 3.1. Let M_0 be the initial matching and for $k \in \mathbb{N}$, define M_k to be the matching before constructing $T(x_0)$ for the k^{th} time. Since M_0 hasn't changed before constructing $T(x_0)$ for the first time, $M_1 = M_0$. We proceed with induction on k.

Suppose $T(x_0)$ is still the same after constructing it with M_k, then, by lemma 3.2, any M_k-augmenting path is vertex disjoint with $T(x_0)$. Flipping all edges in a M_k-augmenting path to create M_{k+1} thus doesn't change any edge incident to a vertex in $T(x_0)$, which means the M_{k+1}-alternating-path tree $T(x_0)$ is the same as the M_k-alternating-path tree $T(x_0)$. By induction, there will never be an augmenting path passing through any vertex in $T(x_0)$, so we can ignore it for the rest of the algorithm. □

4 Multi-source alternating-path trees

The algorithm for MS-BFS is very similar to SS-BFS. The main difference is that an alternating-path tree is built from all unmatched vertices in L rather than one at a time. Any tree that finds an augmenting path will flip that path and stop growing, while other trees keep growing to search for augmenting paths. The resulting alternating-path trees therefore lack one property that was crucial in the proof of lemma 3.2: there may exist an augmenting path starting at the root of a tree, even if we did not find one. This is illustrated in figure 6.

Still, it is possible to prune entire trees when using a MS algorithm, but we have to be more careful.

Definition 4.1. A tree $T(x)$ *depends on* $T(y)$ if $T(x)$, during the construction, tried to search an edge to a vertex that $T(y)$ discovered earlier.

Remark 4.2. A tree $T(x)$ depends on $T(y)$ if and only if $T(x)$ would have grown bigger if $T(y)$ did not exist.

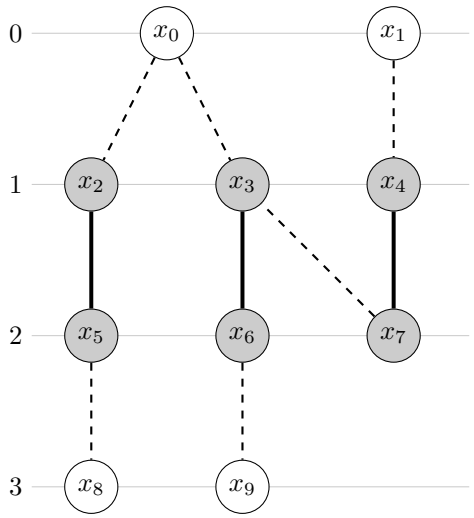


Figure 6: The augmenting path (x_0, x_2, x_5, x_8) is found this phase, but $(x_1, x_4, x_7, x_3, x_6, x_9)$ is not

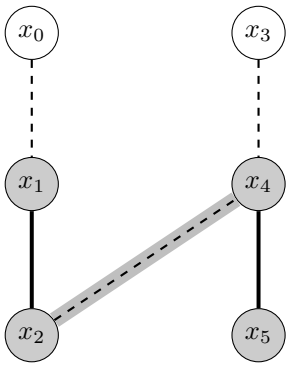


Figure 7: Due to the highlighted edge, $T(x_0)$ depends on $T(x_3)$

Theorem 4.3. *If, for an alternating-path subforest $T_I = \bigcup_{i \in I} T(x_i)$ generated by a MS algorithm, no tree in T_I depends on a tree outside T_I and no augmenting path was found during the construction of T_I, then T_I can be pruned.*

Proof. Given a bipartite graph $G = (L \cup R, E)$ and an M-alternating-path forest $T_I = \bigcup_{i \in I} T(x_i)$. Suppose no tree in T_I depends on a tree outside T_I. We will construct a new graph and prove that we can prune T_I using theorem 3.1. We add a vertex s and for each $i \in I$, we add a vertex y_i. Define $G_+ = (L_+ \cup R_+, E_+)$ with $L_+ = L \cup \{s\}, R_+ = R \cup \bigcup_{i \in I} \{y_i\}$ and $E_+ = E \cup \bigcup_{i \in I} \{\{x_i, y_i\}, \{y_i, s\}\}$. Let $M_+ = M \cup \bigcup_{i \in I} \{\{x_i, y_i\}\}$. The construction of this new graph is illustrated in figure 8.

We construct a new M_+-alternating-path tree $T_+(s)$ using the `SS-search` function in algorithm 2. Suppose we find an M_+-augmenting path P starting at s during the construction of $T_+(s)$. By construction of G_+, this path intersects T_I. Let x be the last vertex of P that intersects T_I and let $i \in I$ such that $x \in T(x_i)$.

If x is unmatched (with respect to M_+), we would have found an M-augmenting path during the construction of T_I, which contradicts that this was not the case.

Assume x is matched, and let y be the next vertex of P, which is outside of T_I. If $\{x, y\}$ is matched, y would have been searched during the construction of T_I, which is not the case. The edge $\{x, y\}$ is therefore unmatched.

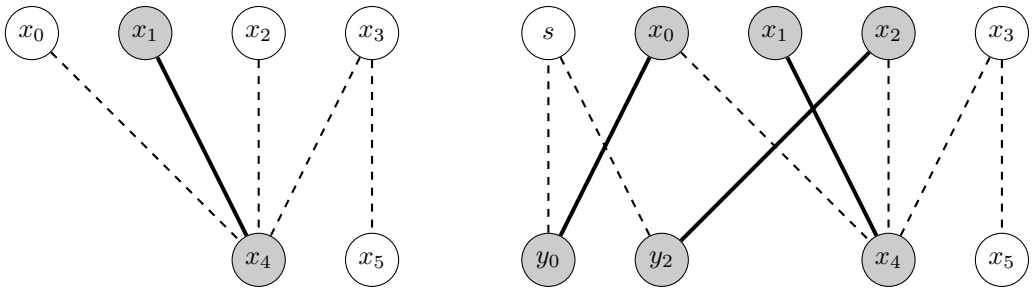


Figure 8: Construction of G_+ with $I = \{0, 2\}$.

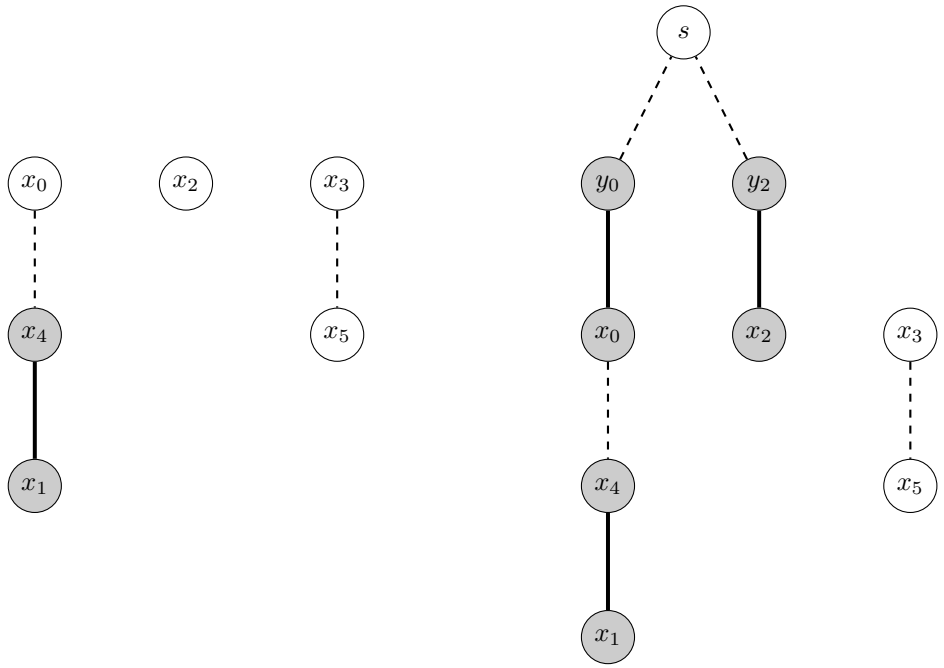


Figure 9: The alternating-path forests of G (left) and G_+ (right) from figure 8. Note that $T(x_3)$ and $T(x_2)$ depend on $T(x_0)$, but $T(x_0)$ and $T(x_2)$ do not depend on $T(x_3)$.

Since $\{x, y\}$ is unmatched and P starts at s with $L_s = 0$, L_x must be even. For y not to be in T_I, y must have been searched by another another tree T outside T_I. Therefore $T(x_i)$ depends on T, which contradicts that no tree in T_I depends on a tree outside T_I. We conclude that there exists no M_+-augmenting path starting from s.

If there is an $i \in I$ for which there exists an M-augmenting path starting at x_i, we can construct an M_+-augmenting path starting from s by adding y_i and s to the x_i-side of the path. Therefore there exists no M-augmenting path starting from x_i, for any $i \in I$. By using theorem 3.1 $|I|$ times, we conclude that T_I may be pruned from the search space. □

An alternative approach to prevent information leaking away between searches is tree-grafting, proposed by Azad et al[1]. Instead of trying to discard a tree T from an unsuccessful search, tree-grafting remembers the dependencies of T. Once a tree that T depends on yields an augmenting path, the search from T is continued.

5 Experimental results

To test whether pruning makes a differece, I have implemented both SS-BFS and MS-BFS without pruning and their respective variants with pruning. To speed up the computations, I initialized the matching with a maximal matching using a greedy algorithm. As a reference, I also tested the built-in Hopcroft-Karp algorithm of the networkx python package[3]. For each of these algorithms, I computed the average time over 5 consecutive runs.

For this test, I prepared 3 random bipartite graphs, each consisting of 10007 vertices in L and 10007 vertices in R, and some graphs built from matrices of the SuiteSparse Matrix Collection[4]. The results are in table 1.

Graph type or name	Number of vertices (in L and R)	Number of edges	Cardinality of maximum matching	SS-BFS	MS-BFS	built-in
nopoly	10774	70842	10774	0.403 **0.381**	0.334 **0.334** 0.334	0.093
Random 1	10007	10097	5456	0.844 **0.825**	0.322 **0.309** 0.350	0.159
qpband	20000	45000	20000	1.363 **1.347**	11.456 **11.444** 11.503	0.203
cyl6	13681	714241	13681	2.884 **2.853**	2.566 **2.559** 2.531	0.384
Random 2	10007	76005	10001	4.147 **3.747**	1.906 **2.347** 2.341	0.384
Random 3	10007	1001751	10007	5.500 **5.425**	4.206 **4.484** 4.431	0.456

Table 1: Speed comparison of different algorithms. For each algorithm, the first value is without pruning, the second with pruning (in bold), and the third value (if available) is the algorithm without pruning, but with all calculations to make pruning possible in place.

For SS-BFS, the speedup gained by pruning is small, but consistent. Even if the pruning does not help, the algorithm is not slowed down. Where the pruning does help greatly, it gives an improvement of about 10%.

For MS-BFS, it should be impossible for the third value to be lower than any of the other two. However, this is the case for multiple test cases. This is most likely caused by variance during measurement. We will consider those values to be equal in these cases. There are only two tested graphs where MS-BFS with pruning is faster than MS-BFS without pruning, with a maximal speedup of 4%. On the other tested graphs, the speed is roughly equal or up to 23% slower.

One result that stands out is MS-BFS on the graph built from the 'qpband' matrix. The algorithm spends most of the time in the phase where all augmenting paths are found, but we do not yet know what causes this slowdown.

6 Conclusion

For single-source algorithms, pruning exhausted search trees is quite easy and gives a speedup of up to 10% on the tested graphs. The multi-source case however, requires much more overhead for the pruning to work, and only gives a relatively small speedup. If the pruning can be implemented much more efficiently, it may be worth it. As far as we know, pruning for MS algorithms (theorem 4.3) is a new contribution.

The tree-grafting technique proposed by Azad et al.[1] also addresses the problem of information leaking away between searches. In contrast to pruning, grafting does make the algorithm run an order of magnitude faster than the standard MS-BFS algorithm.

References

[1] Ariful Azad, Aydın Buluç, and Alex Pothen. "Computing maximum cardinality matchings in parallel on bipartite graphs via tree-grafting". In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 44–59.

[2] Claude Berge. "Two theorems in graph theory". In: *Proceedings of the National Academy of Sciences of the United States of America* 43.9 (1957), p. 842.

[3] NetworkX Developers. *Bipartite - NetworkX 2.3 documentation*. URL: `https://networkxX.github.io/documentation/stable/reference/algorithms/bipartite.html#module-networkx.algorithms.bipartite.matching` (visited on 05/28/2019).

[4] *SuiteSparse Matrix Collection*. URL: `https://sparse.tamu.edu/` (visited on 05/30/2019).

Appendix: Code for SS-BFS with and without pruning

```
import networkx as nx
from matching import Matching
from collections import deque


# The Matching superclass defines a function for flipping edges, as well as
# the functions init_matching and greedy_match used below
class SsBfsPrune(Matching):

    def __init__(self, graph: nx.Graph, prune: bool):
        Matching.__init__(self, graph, None)

        # Indicates whether unsuccessful trees will be pruned
        self.prune = prune

    def algorithm(self):
        # Set all 'match' (matched) and 'vis' (visited) flags to False
        self.init_matching()
        # Greedily find a maximal matching
        self.greedy_match()

        for i in self.G.nodes:
            self.visited = [i]
            # Run a BFS from the new root node i
            if self.bfs(i) or not self.prune:
                # If the search was successful or pruning is disabled,
                # reset all visited flags of the last search
                for j in self.visited:
                    self.G.nodes[j]['vis'] = False

        # Obtain all matched edges from the graph
        return {a: b for (a, b, m) in self.G.edges.data('match') if m}

    def bfs(self, start):
        # Only run the BFS if the starting node is unmatched
        if self.G.nodes[start]['match']:
            return False

        # Initialise the queue with the start node and an artificial 'end
        # of layer indicator'
        q = deque([start, -1])
        even_layer = True
        self.G.nodes[start]['vis'] = True
        prev = [None] * self.G.number_of_nodes()

        while len(q) > 1:
            n = q.popleft()

            if n == -1:
                # End of current layer, initialise new layer
                even_layer = not even_layer
                q.append(-1)
                continue

            for n2 in self.G[n]:
                # Ignore nodes we have already visited
                if self.G.nodes[n2]['vis']:
```

```
                continue

            # Make sure the tree we build is an alternating-path tree, so
            # ignore matched edges when on an even layer, and unmatched
            # edges when on an odd layer
            if self.G.edges[n, n2]['match'] == even_layer:
                continue

            # Mark n as the parent node of n2
            prev[n2] = n

            # If the node is unmatched, we have found an augmenting path
            if not self.G.nodes[n2]['match']:
                last = n2

                # Flip the entire path from the current node to the root
                # node
                while not last == start:
                    self.flip(last, prev[last])
                    last = prev[last]

                # Only mark the first and last node on the path as visited,
                # the nodes in the middle of the path are already marked
                # as matched
                self.G.nodes[n2]['match'] = True
                self.G.nodes[start]['match'] = True
                return True

            # Visiting node n2 only yielded an alternating path, so add it
            # to the search queue
            q.append(n2)
            self.G.nodes[n2]['vis'] = True
            self.visited.append(n2)
    return False
```