Stanford University, CME 323

Edmonds' Blossom Algorithm

Amy Shoemaker and Sagar Vare
June 6, 2016

Abstract

The Blossom Algorithm is a means of finding the maximum matching in a graph, through the Blossom contraction process. This polynomial time algorithm is used in several applications including the assignment problem, the marriage problem, and the Hamiltonian cycle and path problems (i.e., Traveling Salesman Problem). In this report we present theoretical analysis of sequential, parallel and distributed versions of the algorithm. We analyze each one's computational complexity, communication cost, and memory usages. We also analyze the scenarios in which each of the algorithms (sequential, parallel, and distributed) work the best. Finally, we show the results of our experiments with the sequential and parallel implementations. The code for the sequential and parallel implementations can be found here: `https://github.com/amyshoe/CME323-Project`.

1 Introduction

Edmonds' Blossom algorithm is a polynomial time algorithm for finding a maximum matching in a graph.

Definition 1.1. *In a graph G, a **matching** is a subset of edges of G such that no vertex is included more than once.*

Definition 1.2. *A **maximum matching** M of a graph G is a matching that contains the maximum possible edges from the graph. That is, for every matching M' of G, $|M| \geq |M'|$.*

Maximum matchings find application in the Assignment problem, the Graduation Problem and the Hamiltonian Cycle/Path problem, and are widely used in other algorithms, as a sub problem. In 1955, Kuhn solved the problem for maximum matching in a bipartite graph. In 1965, Jack Edmonds came up with the Blossom's Contraction Algorithm for solving the problem in polynomial time. In subsequent years, using a clever reduction of the maximum matching to the Reachability problem and using a simple modification of Depth First Search, a faster implementation of the problem was found.

In this report we discuss the original Edmonds' Blossom algorithm for the undirected and unweighted graph. We consider the analysis for the sequential, parallel, and distributed cases.

2 The Sequential Algorithm

Given a matching, the Blossom algorithm works by finding augmenting paths to increase the size of that matching by one after every iteration. It begins by identifying *exposed vertices* and building alternating paths from these vertices.

Definition 2.1. *For graph G with matching M, an **exposed vertex** v is a vertex that does not belong to M, but is in the graph G. That is $V(G \setminus M)$ are exposed vertices.*

The alternating paths built from exposed vertices are, more specifically, *augmenting paths.*

Definition 2.2. *Given a matching M, an **augmenting path** is an odd length path whose endpoints are distinct exposed vertices, and whose edges $e_1, \ldots, e_{2k+1}$ alternate such that $e_{2i+1} \notin M$ for all $0 \leq i \leq k$ and $e_{2j} \in M$ for all $1 \leq j \leq k$.*

An augmenting path as defined above is an augmenting path of size $2k+1$ and has k edges belonging to the matching and $k+1$ which are not part of the matching. Our goal in finding augmenting paths is that we can toggle which edges are included in order to increase the matching by one, as shown in the following theorem.

Theorem 2.3. *Given an augmenting path and a matching M, by removing the matched edges of the path from M, and by adding the unmatched edges of path to M, we increase the size of M by one.*

Proof. To prove the above claim, we must show that we are increasing M by exactly one and that in doing so we maintain M's status as a matching. We proceed via induction. As a base case, consider an augmenting path of length 1. By adding the path's one edge into the M, we increase M by one. Furthermore, both nodes are matched, so we maintain M as a matching.

Now assume that the claim holds for an augmenting path of length $2k-1$, and consider an augmenting path of length $2k+1$. Since the augmenting path has $2k+1$ edges alternating as unmatched-matched, we know the endpoints of the path are not in the matching. Leaving the two endpoints and their two neighbors, consider the internal $2k-2$ nodes, which form an augmenting path of size $2k-3$. This augmenting path leaves out two unmatched edges and two matched edges from the original length $2k+1$ augmenting path. Let M' be the matching M without the two matched edges. Then, by our inductive assumption, this means that by toggling the edges of the $2k-3$ augmenting path in M', we maintain M' as a matching and increase its size by 1. Let $\hat{M'}$ be this modified matching. With this, $|\hat{M'}| = |M| - 1$. Now, let $\hat{M}$ be $\hat{M'}$ plus edges (v_1, v_2) and (v_{2k+1}, v_{2k+2}). Then, $|\hat{M}| = |\hat{M'}| + 2 = |M| + 1$, as desired. Since $\hat{M'}$ matches all nodes in the path other than v_1, v_2, v_{2k+1}, and v_{2k+2}, and since $\hat{M}$ matches those vertices to each other, we know that all vertices in $\hat{M}$ are matched to exactly one other vertex, so $\hat{M}$ is still a matching. With this, we conclude that our claim holds for augmenting paths of all lengths. □

Keeping in mind Theorem 2.3, we introduce the wrapper function of the Blossom algorithm:

Algorithm 1 Blossom Algorithm: Find Maximum Matching

```
1: procedure SEQ_FIND_MAXIMUM_MATCHING(G, M)
2:     P = SEQ_FIND_AUG_PATH(G, M)
3:     if P == [ ] then
4:         return M
5:     else
6:         Add alternating edges of P to M
7:         return SEQ_FIND_MAXIMUM_MATCHING(G, M)
```

With Theorem 2.4 below, we are guaranteed that when the above algorithm terminates, we will have found a maximum matching.

Theorem 2.4 (Theorem of Correctness)**.** *Given a graph G and a matching M, M is a maximum matching if and only if there is no augmenting path in G.*

Proof. Given a graph G and a maximum matching M, it is clear we cannot find an augmenting path because an augmenting path increases the size of the current matching by one, which would give us a matching larger than the maximum matching: a contradiction.

Next, we prove that if there is no augmenting path, then we have a maximum matching, or alternately, if M is not the maximum matching then we can find an augmenting path. If M is not the maximum matching, consider M' such that $|M'| > M$. We consider the graph $M \cup M'$ and note that each vertex in this graph must have at most degree two (one from an edge in each of the matchings) and at least degree one (it must belong to one of the matchings). Consider the connected components of this graph. The connected components must be paths or cycles, because the maximum degree of a node is two. We note that there are only three possibilities for each of the connected components:

1. A cycle of even length with alternating edges.
2. A path of even length, with alternating edges
3. A path of odd length with alternating edges, with starting and ending edges from the same matching

An odd length cycle is not possible, because then we would have two edges from the same matching incident on the same node. Note that a length-two cycle is possible, with one edge belonging to each of the two matchings.

Because we have $|M'| > M$, the number of nodes belonging to M' must be strictly larger, and only one connected component possibility makes this happen: Case 3, an odd length path which starts and ends with an edge from M'. Therefore, at least one such odd length path must exist, since we have a strict inequality. This odd length path is in fact an augmenting path, because the alternate edges are in M, with the starting and ending

nodes not in M. Hence, when M is not a maximum matching, we can find an augmenting path. With this, we conclude that M is a maximum matching if and only if we there is no augmenting path in G. □

Next, we consider the procedure SEQ_FIND_AUG_PATH for finding these paths:

Algorithm 2 Sequential Blossom Algorithm: Find Augmenting Path

```
1:  procedure SEQ_FIND_AUG_PATH(G, M)
2:      F = empty forest
3:      nodes_to_check ← exposed vertices in G
4:      for v in nodes_to_check do
5:          Add v as single-node tree to F
6:          node_to_root(v) = v
7:      in G, mark all matched edges (all edges in M)
8:      for v in forest_nodes do
9:          while there exists an unmarked edge e = (v, w) do
10:             if w ∉ F then                          ▷ Vertex w must be in M
11:                 SEQ_ADD_TO_FOREST(M, F, v, w)
12:             else
13:                 if dist(w, node_to_root(w)) % 2 == 0 then
14:                     if node_to_root(v) ≠ node_to_root(w) then
15:                         P = SEQ_RETURN_AUG_PATH(F, v, w, node_to_root)
16:                     else
17:                         P = SEQ_BLOSSOM_RECURSION(G, M, F, v, w)
18:                     return P
19:                 else
20:                     # Do nothing
21:             mark edge e
22:     return empty path
```

In Algorithm 2, we consider the exposed vertices $v \in G \setminus M$. These vertices become roots of trees within a forest. The algorithm loops through those vertices, adding pairs of unmatched-matched edges to the corresponding tree in order to build alternating paths. We loop through all vertices that are of even distance from the root, so as we add pairs of unmatched-matched edges, we always have alternating paths in the tree.

Algorithm 3 Sequential Blossom Algorithm: Add to Forest

```
1:  procedure SEQ_ADD_TO_FOREST(M, F, v, w)
2:      x ← vertex adjacent to w in M
3:      add edges (v, w), (w, x) to tree(v) in F
4:      add vertex x to nodes_to_check
5:      node_to_root(w) = node_to_root(v)
6:      node_to_root(x) = node_to_root(v)
```

When two of these alternating paths are connected by an unmatched edge, the algorithm has found an augmenting path. That is, when the algorithm come across an edge (v, w) that connects two trees, it takes the alternating path from the the root of v to v, then adds the new edge from (v, w), then adds the alternating path from w to the root of w. The algorithm then returns this augmenting path.

Note that if w is in the forest, in a different tree than v, but is an odd distance from the root, then the above strategy wouldn't work. The path from w to root(w) would then start with an unmatched edge, but the previous edge (v, w) would also be unmatched, thereby ensuring the path is not alternating. Thus, if w is of odd distance from the root, we do nothing and break from the loop.

Algorithm 4 Sequential Blossom Algorithm: Return Aug Path

```
1: procedure SEQ_RETURN_AUG_PATH(F, v, w, node_to_root)
2:     root_v = node_to_root(v)
3:     root_w = node_to_root(w)
4:     P1 ← SHORTEST_PATH(F,root_v,v)
5:     P2 ← SHORTEST_PATH(F, w,root_w)
6:     return P1 + P2
```

The most interesting case is when the algorithm encounters an edge such that adding it to the tree would create a cycle. This cycle is called a *blossom*, which is an alternating cycle of odd length. Before providing the algorithm for handling blossoms, we first show that when the blossom case arrives, there is indeed a blossom found.

Lemma 2.5. *The Algorithm correctly detects blossoms.*

Proof. If the Algorithm 2 reaches line 19, then we know vertex v in the list of even distance forest nodes, and adjacent vertex w is also in F, is in the same tree as v, and is an even distance from the root. Since vertices v and w are both even distances from the root of their tree, we know that there is an even-length alternating path between them. This is ensured by the fact that edges are added to the forest, it is always done in unmatched-matched pairs. Thus, adding unmarked edge (v, w) closes the even-length path, creating an odd-length cycle. In the cycle, vertex v will be incident to two edges that are not in the matching, whereas all other vertices in the cycle will be incident to one matched edge and one unmatched edge.

Note here that if w was of odd distance from the root of the tree, then the cycle would have three consecutive unmatched edges, meaning it would not be a blossom. However, since we skip over all w that are odd distance from the root, we can be ensured we have a blossom. □

And now, the blossom contraction-recursion process:

Algorithm 5 Sequential Blossom Algorithm: Blossom Recursion

```
1: procedure SEQ_BLOSSOM_RECURSION(G, M, F, v, w)
2:     Form blossom: B = SHORTEST_PATH(F, v, w) + [v]
3:     G' = G with all blossom nodes contracted into w
4:     M' = M with all blossom nodes contracted into w
5:     P' = FIND_AUG_PATH(G', M')
6:     if w ∈ P' then
7:         P = P' lifted with blossom B
8:         return P
9:     else
10:        return P'
```

As a bit of additional terminology before explaining the *lifting* process, consider the following definition.

Definition 2.6. *Blossom vertex v, which is the only blossom vertex incident to two unmatched edges, is the* ***base*** *of the blossom.*

After contracting the blossom into a single node, w, the algorithm proceeds via recursion. When an augmented path is found, the recursion is unraveled. If the augmented path that is returned does not contain the blossom node w, we simply return the augmented path as-is. If, however, the augmented path contains w, then we can lift the blossom to expand the path. In order to do this, we must consider several different cases. The node w could be an endpoint of augmented path P', or it could be somewhere in the middle. We refer to the edges incident to w as the *stems*. In order to lift the blossom, we must consider which blossom nodes the stems were incident to in the original graph. Consider the following claim.

Lemma 2.7. *If w is not an endpoint, one of the two stems must be incident to the base.*

Proof. First note that exactly one of the two stems is in M', since they are part of an augmented path in G'. The unmatched stem could have been incident to any of the nodes in the original graph G. The matched stem, however, can only be incident to the base. Otherwise, the vertex to which it's incident to would be part of two edges within M, which contradicts the fact that M is a matching. We conclude that the matched stem must be incident to the base. □

When two stems are present, lifting the blossom means we insert into the augmented path the portion of the blossom between the two stems. Note there are two possible ways to traverse a blossom: via an odd number of edges or via an even number of edges. Since the stems are alternating (one matched, one unmatched), we lift through the even path in the blossom. This extends the augmenting path in G' to a path in G which remains augmented.

When the blossom is an endpoint of the augmented path (i.e., has just one stem), must be unmatched, as the final edges of augmented paths are always unmatched. We thus traverse the blossom in the direction of the incident matched edge, adding edges until we reach the base (thereby again ensuring that the final edge in the lifted path is unmatched and the path is thus augmented). Note that this implies that if the unmatched edge is already at the base, we do nothing. In implementation of the algorithm, we consider in more detail which cases can and cannot appear in blossom contractions and lifts (see: `https://github.com/amyshoe/CME323-Project`).

Lemma 2.8. *Each augmenting path iteration can have $O(n)$ Blossom contraction steps.*

Proof. We note that each contraction of a blossom reduces the nodes in a graph by at least two. Hence, we can have at most $\lfloor \frac{n}{2} \rfloor$ blossom contractions, meaning each augmenting path iteration cannot have more than $O(n)$ blossom contraction steps. To address that we cannot bring this bound any lower, consider the graph shown below:

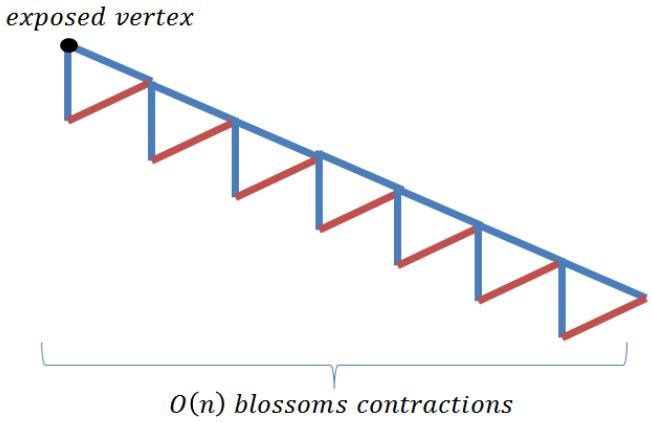

Figure 2.1: Tightness of the Analysis for Sparse Graph

Figure 2.1 will contract the first size-three blossom recurse, and the exposed vertex will move to the base of the second blossom. It will find the second blossom, recurse, etc., until it has contracted all blossoms (the whole graph). At this point, it will begin lifting. We will have to lift out of all $O(n)$ blossom recursions before we can return the augmented path to the main wrapper function. Hence, this graph, actually attains the bound of $\lfloor \frac{n}{2} \rfloor$ contractions in each iteration. With this, we conclude that $O(n)$ is a tight bound. □

Another important consideration is whether finding augmenting paths in the contracted graphs can be productively used (once lifted) to increase the matching in the pre-contraction graph. Indeed, it can! We confirm this with the theorem below.

Theorem 2.9. *If G' is the graph formed by contracting a blossom B in G, then G has an augmenting path if and only if G' has an augmenting path.*

Proof. Consider an augmenting path in G', which passes through the contracted blossom node (if it does not, then the path is identical in G and G', and thus trivially satisfies our claim). We have two cases: whether the contracted blossom node is an interior point on the path, or whether it is an endpoint.

If the blossom node is one of the endpoints of the path, we know that in the blossom (which, once lifted, will have $2k + 1$ nodes), there will be only one attaching stem. This stem not be in M, since it was a final edge in the augmenting path in G', and the final edge augmenting paths are by definition unmatched. Thus, as mentioned previously, if the stem is attached to the base of the lifted blossom, the augmenting path ends there, and we return the original augmenting path (but with the base node in G as endpoint instead of the contracted blossom node in G'). This of course remains an augmenting path in G, since we did not alter the path. If, on the other hand, the stem is incident to a non-base node, then we can extend our augmenting path. Since the stem is attached to a non-base node, one of the two neighboring edges is in M. Extend the augmenting path in this direction, adding pairs of matched-unmatched edges until you reach the base node. Since we've only added alternating edges and have again ended with an unmatched edge, this extended path is an augmenting path in G.

Now, for the second case when the blossom is in interior of the augmenting path, we let v_L denote the node at which the left stem is connected and v_R denote and the node at which the right stem is connected. We then see that we can always take a path from v_L to v_R of even length (even number of edges) because the total number of edges in the blossom are odd, so among the two ways to go from v_L to v_R one way must be of even length. Thus, we have the augmented path in G as: the augmented path in G' up to v_L and then the even length path from v_L to v_R and then the remaining augmented path of G'. Recall from Lemma 2.7 that one of the two stems (namely, the matched stem) must be attached to the base of the blossom. Without loss of generality, suppose the left stem is matched and v_L is the base. Thus, we are guaranteed that traveling the even path across the blossom from v_L to v_R will include precisely unmatched-matched pairs of edges. Thus, the lifted path will thus be augmenting in G. Hence, we see that for every augmenting path of G', we have a augmenting path in G.

Finally, we need to show that for every augmenting path in G, there exists an augmenting path in G'. We note that if the blossom's edges are not part of the augmenting path in G, then the augmented path in G' is trivially the same path. If there are blossom's edges included in the augmenting path, contracting the blossom will eliminate an even number edges (else, Lemma 2.7 would be contradicted). Thus, with an even number of alternating edges removed, the path in G remains an odd-length augmenting path in G'.

Hence we see that G has an augmenting path if and only if G' has an augmenting path. □

2.1 Complexity Analysis

We begin by assessing how many iterations the wrapper function (Algorithm 1) will require.

Theorem 2.10. *The Blossom Algorithm requires at most* $\lfloor \frac{n}{2} \rfloor$ *calls to the* FIND_AUG_PATH *function.*

Proof. Note, that the number of edges in the maximum matching in a graph on n nodes, can be at most $\lfloor \frac{n}{2} \rfloor$. Additionally, we know that each time the augmenting path function is called, the number of edges in the matchings is increased by one. Hence, starting from an empty matching, we can call the function at most $\lfloor \frac{n}{2} \rfloor$ times. □

At each iteration of the augmenting path, we go through at most all the nodes in the graph. And for each node, we go through the unmarked edges. There are 3 possible cases in our analysis for a given node v, and an unmarked edge $e = (v, w)$, as follows:

CASE 1: Blossom Recursion We contract the nodes within the blossom. We go through all the nodes and edges in our graph, and relabel the blossom nodes with a new id given to the blossom. Going through all nodes and edges in the graph (and the matching) brings the computational cost of this case to $O(E + V) = O(m)$. Recall from Lemma 2.8 that there are $O(n)$ recursive calls resulting from this case.

CASE 2: Add to Forest For the edge $e = (v, w)$, if $(w, x) \in M$, where x is the matched partner of w, then we add the edges (v, w) and (w, x) to the Forest. To append these edges to the forest edge list is an $O(1)$ operation. There can be at most m calls to this case, which brings the complexity to $O(m)$.

CASE 3: Return Augmenting Path After the algorithm finds an augmenting path, the algorithm terminates and returns to the wrapper function to determine whether another iteration is necessary. Thus, for each call to FIND_AUG_PATH, Case 3 can happen at most once. In a call to Case 3, we have two calls to SHORTEST_PATH. Since we are performing the shortest path on trees in the forest, we can use a simple depth first-search implementation, bringing the total complexity of Case 3 to $O(n)$.

Total Cost

Now, for the total cost, we have the sum of these above costs per iteration of the algorithm, multiplied by the number of iterations (the number of times we call SEQ_FIND_AUG_PATH).

Hence, we can write this as:

$$\text{Total Cost} = \underbrace{O(n)}_{\text{Iterations}} * \left[\underbrace{O(m)}_{\text{CASE 1}} + \left(\underbrace{O(m)}_{\text{CASE 2}} + \underbrace{O(m)}_{\text{CASE 3}} \right) * \underbrace{O(n)}_{\text{Blossom recursions}} \right]$$

$$= O(n^2 m)$$

Analysis is Tight

Next, we show a family of graphs that achieve the bound of $O(n^2 m)$, revealing that our analysis is tight.

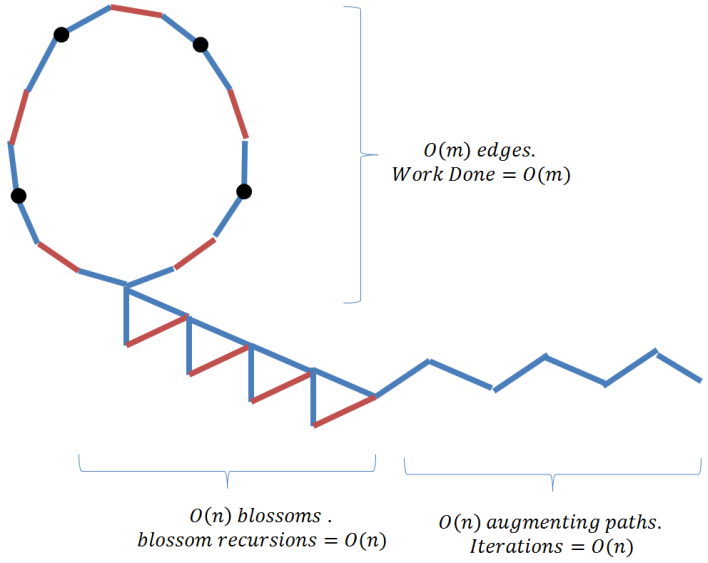

Figure 2.2: Tightness of the Analysis for Sparse Graph

In the above sparse graph (where $O(m) = O(n)$), the edges in matching are shown in red, and the unmatched edges are shown with blue. We have $O(m)$ edges in the balloon structure shown, with $O(n)$ nodes and $O(n)$ exposed vertices shown in black.Then we have $O(n)$ blossoms of size three, connected to one of the nodes, as shown. Finally we have a tail which is also of length $O(n)$. The balloon is constructed in a special way, where we have an exposed vertex which is connected to two nodes, both of which are matched, as shown. So, at the beginning of the algorithm we construct the forest with these exposed vertices, along with the exposed vertices in the tail. If the algorithm goes through all exposed vertices in the balloon, it will add edges to the forest, but will not find any augmenting paths yet. When it reaches the final exposed vertex at the base of the balloon, which is also part of a blossom, the algorithm will contract the blossom into

the same node and start over. We do this same thing in the contracted graph, G', i.e. do $O(m)$ work on the edges and then again contract the blossom at the end. We do this $O(n)$ times because we have $O(n)$ blossoms. Finally the algorithm will arrive at the tail and return a length-1 augmenting path. Because we have $O(n)$ edges in the tail too, we can have $O(n)$ iterations (calls to the FIND_AUG_PATH function). Note that after finding the augmenting paths one-by-one through the tail, the algorithm will need to go back through the new vertices added to the forest from the balloon and it will take additional iterations to find the augmenting paths in the balloon. To find the augmenting paths in the tail along with the augmenting paths in the balloon will still take a total of $O(n)$ iterations.

Thus, in total, we have a $O(n)$ iterations, with $O(n)$ blossom recursions per call, and $O(m)$ work per recursion. Hence in total we have $O(n^2m)$, which makes the analysis tight for this graph.

Note the graph in Figure 2.2 is sparse. To show that our analysis is tight for dense graphs too, we consider a slight modification of the above graph as follows:

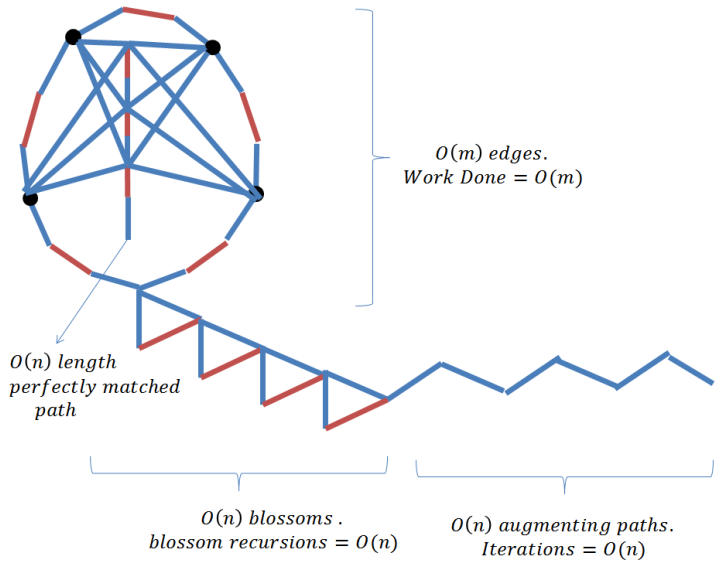

Figure 2.3: Tightness of the Analysis for Dense Graph

The above graph is very similar to the sparse case, with the only difference being that at the center of the balloon we have a perfectly matched set of nodes as shown. The alternating nodes of this path are connected to the exposed vertices on the balloon. Because we have $O(n)$ exposed vertices and $O(n)$ nodes in the central path, we must have $O(n^2)$ edges, and hence we get that most of the edges are present between the connections in the central path and the balloon's edge vertices. Thus, this graph is indeed dense, with $O(m) = O(n^2)$.

Now, we consider the steps that the algorithm would do. First it would find the $O(n)$ exposed vertices of the balloon. The first exposed vertex would add all nodes of the central path to its forest, which would require $O(n)$ work. Now, the second exposed vertex of the balloon, would add the 2 edges that are adjacent to it (on the balloon's rim) to its forest. Then it would consider the $O(n)$ edges that are going from it to the central path. It would check and see that all these nodes are already present in the forest (tree of the first exposed vertex), and hence it would reject Case 1. But then it would see that all the nodes adjacent to it are present at an odd length from the root in the forest, and hence it would even reject Case 2 and Case 3. It would have to check each of the nodes, but for each node it would do nothing. With this it would require $O(n)$ work of checking each of the edges, but would add nothing to the forest. The remaining $O(n)$ nodes would do the same, and with this they would check a total of $O(n^2) = O(m)$ edges. Finally, at the end of this, the last exposed vertex to be checked would be base of the balloon, where the blossom is attached. The algorithm would then contract the blossom, and restart the algorithm. The same procedure will repeat and another $O(n)$ blossoms that are shown would be contracted and finally, we would get an augmenting path. As shown in the tail, we have $O(n)$ augmenting paths, so there would be $O(n)$ iterations of this whole process. As with the sparse graph, the algorithm would require some additional final iterations (still within the $O(n)$ bound) to find the augmenting paths within the balloon portion of the graph. Thus, the algorithm would take $O(n^2m)$ time to find a maximum matching in this graph.

Hence, we see that the analysis is tight for both the sparse and the dense case with a family of graphs that attains the bound.

3 The Parallel Algorithm

Of the two sub-algorithms within Edmonds' Blossom Algorithm (FIND_MAXIMUM_MATCHING and FIND_AUG_PATH), we can only work to parallelize the latter. The wrapper algorithm (Algorithm 6 below) requires integrating each result before recursing, which creates dependencies that are too sequential to parallelize. As such, our parallel algorithm will have to go through the $\lfloor \frac{n}{2} \rfloor$ iterations sequentially, and its complexity will thus always have a factor $O(n)$ in it.

Algorithm 6 Parallel Blossom Algorithm: Find Maximum Matching

```
1: procedure PAR_FIND_MAXIMUM_MATCHING(G, M)
2:     P = PAR_FIND_AUG_PATH(G, M)
3:     if P == [ ] then
4:         return M
5:     else
6:         Add alternating edges of P to M in parallel
7:         return PAR_FIND_MAXIMUM_MATCHING(G, M)
```

Notice, however, that for each augmenting path that we find (sequentially), we can add the alternating edges to the matching in parallel.

The algorithm for finding an augmenting path (Algorithm 8 on the next page) also has an inherently sequential loop: the outermost for loop to go through the vertices in forest's `nodes_to_check`. With each loop, the result changes drastically depending on the current sate of the tree (which itself depends on the order that you visit the nodes in). While any sequential order will guarantee an augmenting path, doing all nodes in parallel would yield a faulty return. The remainder of the algorithm, however, can be done in parallel. That is, we can loop through each of the unmarked edges incident to v in parallel. Notice that this algorithm, like the sequential version, finds which of the three main cases an edge (v, w) falls into, and then calls the appropriate procedure. However, here there are some nuances to how we examine the cases in order to allow for all edges' work to be done parallel.

An interesting difference between this parallel algorithm and the sequential version is that PAR_ADD_TO_FOREST does not modify the forest. Observe Algorithm 7,

Algorithm 7 Sequential Blossom Algorithm: Add to Forest

```
1: procedure PAR_ADD_TO_FOREST(M, w,temp)
2:     x ← vertex adjacent to w in M
3:     temp[w] = x
```

and how it fits into Algorithm 8 below.

Algorithm 8 Parallel Blossom Algorithm: Find Augmenting Path

```
1: procedure PAR_FIND_AUG_PATH(G, M)
2:     F = empty forest
3:     nodes_to_check ← exposed vertices in G
4:     parallel for v in nodes_to_check do
5:         Add v as single-node tree to F
6:         node_to_root(v) = v
7:     end parallel for
8:     in G, parallel mark all matched edges (all edges in M)
9:     for v in forest_nodes do
10:        temp ← empty vector of size |N(v)|
11:        parallel for unmarked edge e = (v, w) in N(v) do
12:            if w ∉ F then                          ▷ Vertex w must be in M
13:                PAR_ADD_TO_FOREST(M, w, temp)
14:            else
15:                if dist(w, node_to_root(w)) % 2 == 0 then
16:                    if node_to_root(v) ≠ node_to_root(w) then
17:                        P = PAR_RETURN_AUG_PATH(F, v, w, node_to_root)
18:                    else
19:                        P = PAR_BLOSSOM_RECURSION(G, M, F, v, w)
20:                    return P
21:                else
22:                    # Do nothing
23:            mark edge e
24:        end parallel for
25:        parallel for i in range(N(v).size()) do
26:            if i == temp[temp[i]] then
27:                P = PAR_BLOSSOM_RECURSION(G, M, F, v, i)
28:                return P
29:            else if temp[i] ≠ Null then
30:                F[node_to_tree(v)].edges().append((v, i))
31:                F[node_to_tree(v)].edges().append((i, temp[i]))
32:                nodes_to_check.append(temp[i])
33:        end parallel for
34:    return empty path
```

By postponing adding the edges to the forest, we allow the other two cases (when an augmenting path is returned and when a blossom is found so we recurse) to take precedence. Since upon both return and recursion we throw out the forest and start over in the next iteration, we needn't add the edges to the forest until we are certain that for our given v there will be no returns or blossom recursions. Note that with this, each call to PAR_ADD_TO_FOREST is $O(1)$.

Of course, if no augmenting paths or blossoms are found within our main parallel for loop (lines 11-24 in Algorithm 8), we need to do some post-processing to add the new edges to the forest and to check if doing so leads to any cycles (blossoms). This job is made much easier by the fact that only length-three blossoms can appear at this stage, as shown in the following theorem.

Theorem 3.1. *Suppose we process the incident edges of node v in parallel, and we find that for each unmarked edge (v, w), adding to the forest the edge (v, w) alone yields neither an augmenting path nor a blossom. Then, after the adding to the forest all edge pairs $(v, w), (w, x)$ which had $w \notin F$ and $(w, x) \in M$, the only possible cycles that appear are length-three blossoms.*

Proof. Since these edges when considered individually did not yield a blossom, we need only worry about the interaction of these added edges. We note that all the edges added are of the form $(v, w_i), (w_i, x_i)$, i.e. they all have the node v in common. Secondly, note that each time we add the pair of edges as above, we know that each of the nodes $w$and x are not in the Forest, otherwise it would have created a blossom. Thus, the only way to form a blossom would be to add $(v, w), (w, x)$ when considering edge (v, w), *and* to add $(v, x), (x, w)$ when considering edge (v, x). This would create a blossom of length 3.

Suppose toward contradiction that adding these edges yielded a blossom of size five or greater. Since we are adding only unmatched-matched pairs of edges to v, it must be the case that v is the base of the blossom. Then, if $(v, w_i), (w_i, x_i)$ and $(v, w_j), (w_j, x_j)$ were added, it would have to be the case that a blossom is formed through there being an augmenting path between nodes x_i and x_j (where a path of length $2k - 1$ corresponds to a blossom of length $2k + 3$). However, for there to be a path between x_i and x_j in the tree, the two nodes must have already been present, which contradicts our original assumption. With this, we see that blossoms of size five or larger are not possible when the edges are added in parallel. Therefore, after adding to the forest, to identify all blossoms we need only look for length-three cycles among v and the new edges. □

Thus, in the post-processing in lines 11-24 in Algorithm 8, we only check each of the new edges in parallel to see if it is part of a three-cycle (which would be a blossom). This can be done in $O(1)$ depth. If any processor finds a blossom, all processors break and we pursue the one call to PAR_BLOSSOM_RECURSION.

Algorithm 9 can also be called during the main parallel for loop, in which we decide what case an edge falls into. This function is very similar to Algorithm 5, the sequential version, except that we can perform the contractions in parallel. To contract the node set of G, we simply assign a processor to each node in G, and if the node is in $B \setminus w$, delete it. For the edge set of G, for each edge (i, j), a processor can call a mapper function to map each edge appropriately as follows:

```
procedure MAPPER(B, (i, j))
    if i ∈ B&j ∈ B then
        return Null
    else if i ∈ B then
        return (w, j)
    else if j ∈ B then
        return (i, w)
    else
        return (i, j)
```

Note that this allows us to contract the node and edge sets of G in $O(1)$ depth. Similarly, for the matching, we can perform a parallel filter operation, filtering on $i \notin B$ & $j \notin B$.

Algorithm 9 Parallel Blossom Algorithm: Blossom Recursion

```
1:  procedure PAR_BLOSSOM_RECURSION(G, M, F, v, w)
2:      B = SHORTEST_PATH(F, v, w) + [v]
3:      G' = G with all blossom nodes contracted into w in parallel
4:      M' = M with all blossom nodes contracted into w in parallel
5:      P' =PAR_FIND_AUG_PATH(G', M')
6:      if w ∈ P' then
7:          P = P' lifted with blossom B
8:          return P
9:      else
10:         return P'
```

Finally, we have the function to return an augmenting path. This case is simply two calls to a parallel shortest path function, which is the complexity bottleneck. As with the sequential case, we concatenate the two shortest paths to get our augmenting path.

Algorithm 10 Parallel Blossom Algorithm: Return Aug Path

```
1: procedure PAR_RETURN_AUG_PATH(F, v, w,node_to_root)
2:     root_v = node_to_root(v)
3:     root_w = node_to_root(w)
4:     P1 ← SHORTEST_PATH(F,root_v, v)
5:     P2 ← SHORTEST_PATH(F, w, root_w)
6:     return P1 + P2
```

Now, let us summarize the complexity requirements of the above algorithms.

3.1 Complexity Analysis

As in the sequential case, let us first consider the three main cases of the algorithm individually.

CASE 1: Add to Forest Recall that Case 1 in our parallel algorithm is divided into two portions. The first (Case 1.1) is simply collecting the edges we wish to add into a temporary array. This has $O(1)$ depth. The second (Case 1.2) is the post-processing, which as described above is also $O(1)$ depth. This brings the total depth of Case 1 to $O(1)$.

CASE 2: Blossom Recursion Each blossom recursion requires only $O(1)$ depth to contract the graphs in parallel. However, worst case scenario, there could be $\deg v \leq O(n)$ of these blossom recursions for a single forest node v.

CASE 3: Return Augmenting Path As with the sequential algorithm, there will only be one call to Case 3 per iteration. However, Case 3 itself requires two calls to a sequential shortest path algorithm. As in the sequential version, we are working with a tree, so can use Depth First Search to make this cost $O(n)$. It may be possible to find a clever way to parallelize these shortest path calls as well, but since this is not the complexity bottleneck of depth's computational cost, we leave them as sequential calls.

Total Cost

Now, for the total cost we take the sum of these costs per iteration, multiplied by the number of iterations (the number of times we call PAR_FIND_AUG_PATH). Hence, we can write this as:

$$T_\infty = \underbrace{O(n)}_{\text{Iterations}} * \left[\underbrace{O(n)}_{\text{Forest Nodes}} * \left(\underbrace{O(1)}_{\text{CASE 1}} + \underbrace{O(1)}_{\text{CASE 2}} \right) * \underbrace{O(n)}_{\text{Blossom Recursions}} + \underbrace{O(n)}_{\text{CASE 3}} \right]$$

$$= O(n^3)$$

Notice that our depth now scales only with the number of nodes, as we have removed dependence on m, the number of edges. In Section 5, we will see the benefits of this independence in our implementation.

The algorithm's work remains the same as in the sequential case. So by Brent's Theorem, we can conclude

$$\frac{n^2 m}{p} \leq T_p \leq \frac{n^2 m}{p} + n^3 = n^2(n + m/p).$$

4 The Distributed Algorithm

Similar to the case of the parallel implementation we note that the only part of the algorithm that can be done in a distributed fashion is the searching the neighborhood of a node, because the rest is inherently sequential. As shown in the parallel version, using a naive parallelization method will work. We proved this by showing that every case done in the sequential version is still correctly done in the parallel or distributed environment. This is the idea for the distributed case too.

4.1 Data Structures

We store the graph as two DataFrames, for the edge list and the node list. Note, we are storing each edge between i and j, as (i, j) and (j, i), since we are dealing with undirected graphs. We store the edges of the forest in the same undirected way. The nodes of the forest are also stored as a DataFrame, but each node holds some additional information. We have columns storing the additional feature of which tree of the forest the node belongs to, and the additional feature of its parity within in the tree (odd or even distance from root). The DataFrame of undirected forest edges has the additional feature column storing the root of the tree to which each the edge belongs. Below, we show the data structures that we are using:

Node	**root**	**parity**
1	10	1
2	10	2
3	18	2
4	18	1
5	19	1

Forest Nodes

Node1	**Node2**	**root**
1	2	10
2	1	10
3	4	18
4	3	18
5	8	19

Forest Edges

id	**Node**
1	10
2	11
3	14
4	12
5	1

Graph Nodes

Node1	**Node2**
1	2
2	1
3	4
4	3
5	8

Graph Edges

Memory Assumptions

For the distributed case, we are assuming that all the quantities that are $O(n)$ can be stored locally, on every machine. Hence, we assume that the matchings, which are at most n nodes, can be stored locally. Additionally, the node lists of the graph G and of the forest F (which we are creating at each iteration) are present with each of the individual machines, as the driver broadcasts this over the network to each of the worker machines. We only distribute the edges of G and F.

4.2 The Algorithm

Because we can fit the matching locally, the driver machine will call Algorithm 1. Below is the distributed version of the main algorithm to find augmenting paths.

Algorithm 11 Distributed Blossom Algorithm: Find Augmenting Path

```
1: procedure DIS_FIND_AUG_PATH(G, M)
2:     F = empty forest
3:     F.nodes = leftOuterJoin(G.nodes, M.nodes.map(x ↦ (x, x))
4:     for v in F.nodes() do
5:         root_v = F.nodes.filter(node == v).collect()
6:         BROADCAST(root_v)
7:         # Case 1: Blossom Recursion
8:         blossom_forest = F.nodes.filter(F.root == root_v).filter(F.parity == 2)
9:         edge_list = G.edges.filter(node1 == v)
10:        B_edges = edge_list.map((i, j) ↦ UNION (i, j,blossom_forest))
11:        B_edges = B_edges.filter(x ≠ Null)
12:        B_edges.collect()
13:        if B_edges == Null then
14:            # Do Nothing
15:        else
16:            B_node = B_edges[1][2]
17:            return DIS_BLOSSOM_RECURSION(v, B_node, F, G, M)
18:        # Case 2: Add to forest
19:        edges_not_in_forest = edge_list.map((i, j) ↦ UNION(i, j)).filter(x ≠ Null)
20:        edges_added = edges_not_in_forest.map((i, j) ↦ UNION(i, j, M.edges())
21:        edges_added = edges_added.filter(x ≠ Null)
22:        edges_added.collect()
23:        for (i, j) in edges_added do
24:            if (j, i) in edges_added then
25:                return DIS_BLOSSOM_RECURSION(v, (i, j, v), F, G, M)
26:            else
27:                F.edges.append(edges_added.map((a, b) ↦ ((a, b), root_v)&((b, a), root_v)))
28:                F.nodes.append(edges_added.filter(node1 ≠ v)
29:                F.nodes.map((a, b) ↦ (a, root_v, 1)&(b, root_v, 2))
30:                BROADCAST(F)
31:        # Case 3: Return Aug Path
32:        filtered_forest = F.nodes.filter(F.root ≠ root_v).filter(F.parity == 2)
33:        aug_edges = edge_list.map((i, j) ↦ Union(i, j,filtered_forest)).filter(x ≠ Null)
34:        w = aug_edges[1][2].collect()
35:        if w == Null then
36:            # Do Nothing
37:        else
38:            return DIS_RETURN_AUG_PATH(v, root_v, w, F, G, M)
39:    return empty path
40: procedure UNION(i, j,edge_set)
41:    if (i, j) ∈edge_set then
42:        return (i, j)
43:    else
44:        return Null
```

And below are the distributed functions for the blossom recursion and return augmenting paths (the case of adding edges to the forest is done directly in Algorithm 11).

Algorithm 12 Distributed Blossom Algorithm: Blossom Recursion

procedure DIS_BLOSSOM_RECURSION(v, B_node, F, G, M)
  short_path = SHORTEST_PATH(B_node,v)
  short_nodes = short_path - {B_node}
  BROADCAST(short_path, short_nodes)
  $V' = G$.nodes.filter(node1 $\notin$ short_nodes)
  $\hat{E} = G$.edges.map($(i, j) \mapsto$ BLOSSOM_MAP(i, j,short_nodes,B_node)
  $E' = \hat{E}$.filter(node1 $\neq$ node2)
  $M' = M$.filter(node1 $\notin$ short_nodes & node2 $\notin$ short_nodes)
  $G' =$ Graph(V', E')
  $P' =$ DIS_FIND_AUG_PATH(G', M')
  $P = P'$ lifted sequentially with blossom
  return P
procedure BLOSSOM_MAP(i, j,short_nodes,B_node)
  if short_nodes.has(i) **then**
    i = B_node
  else if short_nodes.has(j) **then**
    j = B_node
  return (i, j)

Algorithm 13 Distributed Blossom Algorithm: Find Augmenting Path

procedure DIS_RETURN_AUG_PATH(v, root_v, w, F, G, M)
  root_w = F.nodes.filter(node == w).root.collect()
  BROADCAST(root_w)
  short_path = F.SHORTEST_PATH(root_v,v) + F.SHORTEST_PATH(w,root_w)
  return short_path

Notice again that in both Algorithm 12 and Algorithm 13, we perform computations like shortest path locally, which requires a broadcast afterwards to disseminate the results or requires a collect beforehand so that the driver can have access to the necessary information. The algorithm uses map and filter to divide each of the edges in the neighborhood of node v into the three cases. We then realized that each case is of the size $O(deg(v))$, and hence updating the forest or returning an augmenting path can both be done by the driver. If it is only updating the forest (CASE - 2), then we broadcast the updated forest. This way is not very efficient, and requires a lot of communication, but because of the inherently sequential nature of the algorithm, we bear the computational and communication costs.

4.3 Complexity Analysis

As in the previous case of parallel implementation, we consider the complexity within the three cases, individually. In the analysis below, let s be the number of machines and let p be the number of processors per machine.

CASE 1: Blossom Recursion By storing the forest locally on the driver, and broadcasting it before each iteration, we ensure that we are intelligently avoiding an all-to-all communication with a one-to-all broadcast and an all-to-one collect. This brings down the total communication cost and ensures that we can work with a network which supports only one-to-all and an all-to-one communication. We perform a few embarrassingly parallel filter operations which give a computational complexity of $O(\frac{n}{sp})$ and a local shortest path computation in a tree, which is $O(n)$, but only once per blossom recursion. Thus we see:

- Shuffle Size = 0
- Communication Cost = $O(\deg v)$
- Computational Complexity = $O\left(\frac{n}{sp}\right) + O\left(n\right)$

CASE 2: Add to Forest We avoid an InnerJoin and a LeftOuterJoin by the same trick as Case 1: storing the forest locally. Additionally, to keep the forest up-to-date, the driver adds the new edges to its local forest, and broadcasts the updated forest to the local machines, which again ensures we don't need an all-to-all communication. We need to broadcast at most $O(\deg v)$ number of edges, so the operations of updating the forest totals to $O(\frac{\deg v}{p})$ done locally on the driver and then broadcasted. Additionally, there are some embarrassingly parallel operations that must be performed on the entire edge list.

- Shuffle Size = 0
- Communication Cost = $O(\deg v)$
- Computational Complexity = $O\left(\frac{n}{sp}\right) + O\left(\frac{\deg v}{p}\right)$

CASE 3: Return Augmenting Path We run this portion of the algorithm exactly once per call to the FIND_AUG_PATH, hence we have at the most $O(n)$ calls to this function in total, throughout all iterations. Again, we have a computational cost of at most $O(\frac{n}{p})$ for the shortest path algorithm, which is performed locally. Notice that we avoid an InnerJoin operation by keeping the forest locally, and hence reducing the communication cost. We incur a computational cost of $O(\frac{n}{p})$. There is no shuffle involved.

- Shuffle Size = 0
- Communication Cost = $O(1)$
- Computational complexity = $O\left(n\right)$

Total Communication cost

By using an All-to-one and One-to-all communications we cleverly avoid the use of any join or an all-to-all communication. The total communication cost is the sum over the communication cost of each of the case per call to the FIND_AUG_PATH multiplied with the $O(n)$ at most calls to the function. Hence, we have:

$$\underbrace{O(n)}_{\text{Iterations}} * \left[\underbrace{O(n)}_{\text{Blossom Recursion}} * \underbrace{\sum_{v \in \text{Forest\_nodes}}}_{} \left(\underbrace{O(deg(v))}_{\text{CASE 1}} + \underbrace{O(deg(v)}_{\text{CASE 2}} \right) + \underbrace{O(1)}_{\text{CASE 3}} \right] = O(mn^2)$$

Total Shuffle Size

By storing the forest locally, we were able to switch from using a join to a local filter. This removed the need for any joins or shuffles.

Total Computational Cost

For the total computational cost we have at most $O(n)$ calls to the FIND_AUG_PATH, an iteration over all the nodes in the graph, and the cost of going through all the cases analyzed above. Hence, we have the cost as follows:

$$\underbrace{O(n)}_{\text{Iterations}} * \left[\underbrace{O(n)}_{\text{Blossom Recursion}} * \left(\underbrace{O(n)}_{\text{CASE 1}} + \sum_{v \in \text{Forest\_nodes}} \left(\underbrace{O\left(\frac{n}{sp}\right)}_{\text{CASE 1}} + \underbrace{O\left(\frac{n}{sp} + \frac{\deg v}{p}\right)}_{\text{CASE 2}} \right) \right) + \underbrace{O(n)}_{\text{CASE 3}} \right]$$

$$= O\left(\frac{n^4}{p}\right)$$

5 Results

Parallel vs Sequential Implementation

Due to the prohibitive communication costs of the distributed algorithm, we chose to implement only the sequential and parallel versions. The parallel algorithm was implemented using threading on our local machines with 4 cores, in a manner as described in Section 3. For measuring the performance we generated Erdős-Rènyi graphs with different number of nodes and density, and compared our parallel and sequential algorithms run times, averaging the run times over five runs. The following figure compares the parallel and sequential implementations as we vary the number of nodes in the graphs, as well as at different densities.

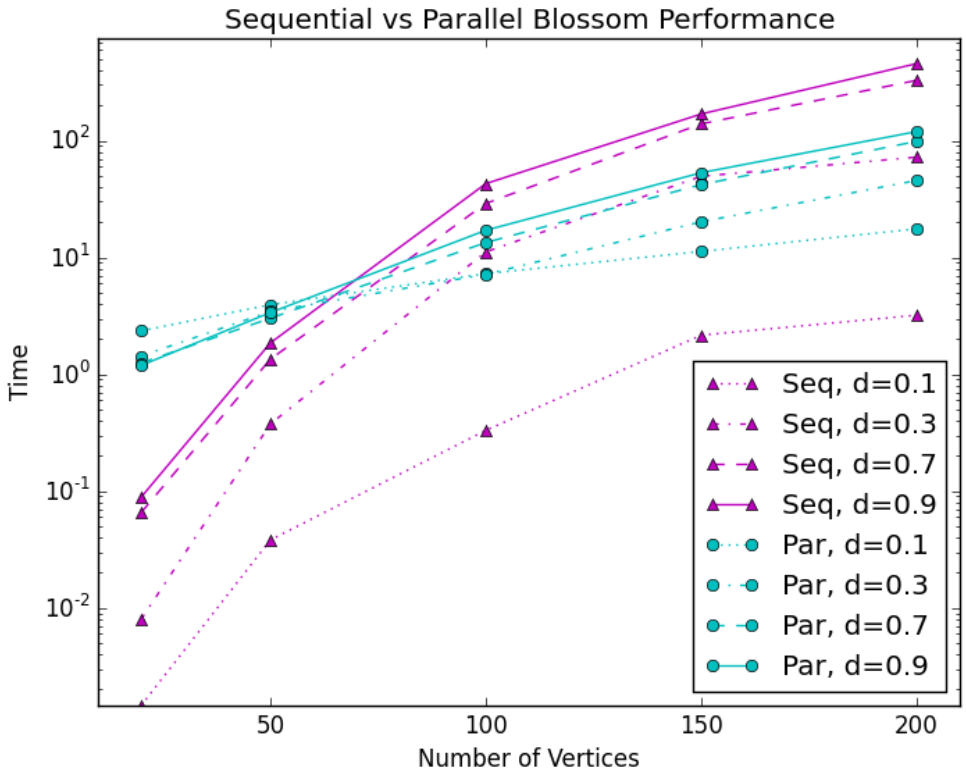

Figure 5.1: Parallel vs Sequential algorithms run times on different graphs

The graphs are plotted on a logarithmic time scale. The cyan curves are the parallel implementations and the magenta curves are the sequential implementations. We tested the run times with different densities and different number of nodes in the graphs. We see that we achieve a speed-up of about 3.5 on the dense graphs, which is close to 4, the number of processors available for running the algorithms, which matches our analysis.

The only graph where this speed up is not seen is the sparse graph with density = 0.1. The sequential algorithm is faster than the parallel one on sparse graphs. This is clear from our analysis, as well. On sparse graphs the neighborhoods of a node can be searched about as fast in parallel as in the sequential counterpart, hence giving no speed ups.

It is interesting that we can clearly see the difference in the run times over dense graphs, since this was precisely what our theoretical analysis predicted. The parallel implementation does not scale with the density of the graph or the number of edges but only with the number of nodes, and this can be seen with the parallel lines being close to each other while we see a huge variation in the run times of the sequential algorithms as the density varies.

Thus, we can conclude that even on a single machine and on relatively small graphs, the parallel algorithm runs much faster than the sequential one for dense graphs. It seems clear that testing the algorithms on graphs with larger values of n would corroborate these results, but the $O(n^4)$ order of computations in the sequential version on dense graphs and $O(\frac{n^4}{p} + n^3)$ in the parallel case are prohibitive for these experiments.

6 Conclusion

In this report, we presented the sequential Edmonds' Blossom Algorithm and proved that it finds a maximum matching. Next, we analyzed the run time of the sequential algorithm, and provided a family of graphs to show that our analysis was tight.

We then came up with a scheme to parallelize the blossom algorithm by searching the neighborhood of each node in parallel. We were able to prove that the algorithm finds the maximum matching correctly, and with this were able to bring down the total computational complexity, removing dependence on the number of edges, so that the parallel algorithm scales only with the number of nodes.

The same parallelizing idea was transferred over to the distributed case, but additional care was taken to avoid unnecessary communication. Specifically, to avoid an all-to-all communication the forest being built was stored locally and updated after each iteration, which led to a shuffle size of zero in the whole algorithm. We then analyzed the communication cost and the total computational complexity for the distributed algorithm.

Finally, we tested the run times of the parallel and the sequential algorithms on Erdős-Rènyi graphs, which revealed and a clear speed up proportional to the number of processors could be observed. The fact that parallel algorithm scales only with the number of nodes also was evident as it was able to beat its sequential counterpart for dense graphs.

6.1 Future Work

- It would be great to test how these algorithms scale with larger number of cores, as the testing was done only with 4 cores.
- It would also be interesting to test our algorithms on graphs with large node sets.
- In the future, it would be nice to implement the distributed algorithm as outlined, to see whether the communication costs are indeed prohibitive in practice, too.
- We have some ideas for using a warm start, where instead of throwing out the complete forest that was built, only part of it needs to be thrown out.
- The algorithm can be improved by not lifting the blossom every time but rather working completely with the smaller contracted graph, and lifting only when we get a maximum matching on the contracted graph. More work can be done on proving the correctness of doing so, and analyzing what complexity advantages this gives.
- Finally, we would like to expand our algorithms and analysis to accommodate graphs with weighted edges.

7 References

References

[1] Wikipedia contributors. "Blossom algorithm." Wikipedia, The Free Encyclopedia, 3 Mar. 2016. Web. 5 Jun. 2016.

[2] Edmonds, Jack. "Paths, trees, and flowers." Canadian Journal of mathematics 17.3 (1965): 449-467.

[3] Reza Zadeh. "Distributed Algorithms and Optimization." `http://stanford.edu/~rezab/dao/`

[4] Micali, Silvio, and Vijay V. Vazirani. "An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs." Foundations of Computer Science, 1980., 21st Annual Symposium on. IEEE, 1980. APA