6-1-2020

# Push-relabel algorithms for computing perfect matchings of regular bipartite multigraphs

Benjamin J. Coleman
*Dartmouth College*

Dartmouth Computer Science Technical Report TR2020-887

# Push-relabel algorithms for computing perfect matchings of regular bipartite multigraphs

Benjamin Coleman

June 2020

**Abstract**

We seek to compute perfect matchings of a $d$-regular bipartite multigraph $G = (V, E)$. If $d$ is a power of 2, we can perform Euler decomposition, which recursively separates a graph of even degree into subgraphs of smaller degree. When $d$ is not a power of 2, however, Euler decomposition eventually returns a subgraph of odd degree. At this point, we can manually remove a perfect matching to return the graph to even degree and continue Euler decomposition.

In this paper, we explore push-relabel algorithms as potential solutions to removing a perfect matching from a regular bipartite multigraph. Empirical analysis shows that these maximum-flow approaches, augmented by preflow-conditioning heuristics, find a perfect matching in $O(E)$ time. Our experiments also suggest that the relabel-to-front algorithm is optimal for bipartite maximum-flow networks.

## 1 Introduction

Given a $d$-regular bipartite multigraph, we seek an algorithm to compute and isolate a perfect matching. An undirected graph $G = (V, E)$ is *bipartite* if $V$ can be partitioned into two disjoint subsets, $L$ and $R$, such that each edge in $E$ connects a vertex in $L$ to a vertex in $R$. Furthermore, $G$ is *$d$-regular* if each vertex in $V$ has $d$ incident edges. It is a *multigraph* if it allows multiple edges to be placed between the same two vertices.

Suppose that $G = (V, E)$ is a $d$-regular bipartite multigraph. A *perfect matching* of $G$ is a subset of $E$ that covers $V$ with exactly one incident edge on each vertex. It can otherwise be thought of as a bijective map $L \rightarrow R$. Perfect matchings have several applications in routing, disk mapping, and the class-teacher timetable problem.

Because $G$ is regular, Hall's Marriage Theorem [4] implies that the edges of $G$ can be decomposed into $d$ disjoint perfect matchings. A general computational problem is then to isolate a set of these $d$ disjoint perfect matchings. When $d$ is even, we may iteratively identify an Euler orientation and perform an Euler split of $G$ to separate disjoint perfect matchings into two partitions of $d/2$-regularity. An *Euler orientation* is a directing of edges such that each vertex has an in-degree and out-degree of $d/2$. We find one by simply tracing out a set of edge-disjoint cycles. Performing an *Euler split* of $G$ then partitions the graph in two based on each edge's Euler orientation—whether the edge was traced from a vertex in $L$ to a vertex in $R$, or vice versa. In this iterative process, termed *Euler decomposition*, each of $\lg d$ iterations considers all edges in $E$, which results in a running time of $O(E \lg d)$.

If $d$ is not a power of 2, however, then there will be an iteration where $d$ is odd, and no Euler orientation is possible. At this point, we may remove a perfect matching to return the graph to even degree. Doing so in $O(E)$ time maintains the overall $O(E \lg d)$ running time of Euler decomposition.

This paper deals with the case where $d$ is odd and we wish to isolate a perfect matching in $O(E)$ time. We begin by exploring current approaches and propose a solution based on the push-relabel method. Experiments show that this method, combined with several heuristics, can compute a perfect matching in $O(E)$ time. We were, however, unable to analytically bound the running time to $O(E)$.

## 2 Background

Previous studies on the perfect-matching problem are well documented. A 2001 paper by Cole, Ost, and Schirra outlines a theoretical approach to identify a perfect matching in $O(E)$ time [1]. According to one of the authors (Cole), however, there is no known implementation of the algorithm. More recent work by Hannigan [5] led to an algorithm that builds a partial perfect matching by selecting edges based on vertex neighborhood size. Unfortunately, this greedy approach does not guarantee a perfect matching, especially in sparse graphs. A paper by Ostrowski [8] proposes heuristics to find disjoint edge sets—chains—that cover each vertex twice, and so form an Euler split. Again, however, the heuristics do not meet the $O(E)$ time constraint. In a 2015 paper [7], Neckowicz explores adding a perfect matching of dummy edges and then performing an Euler split. But again, separating dummy edges from the resulting partitions was not feasible in $O(E)$ time.

Approaches to the maximum bipartite matching problem have either failed to prove an $O(E)$ time bound or lack an implementation. The Hopcroft-Karp algorithm [6] is one of the best-known algorithms in the field, finding a perfect matching in $O(\sqrt{V}E)$ time. In addition, a paper by Goel, Kapralov, and Khanna [3] discusses an algorithm that finds a perfect matching in $O(V \lg V)$ expected time by tracing random walks of the graph.

## 3 Maximum-flow approach

We can reduce a perfect matching of a regular bipartite multigraph to the maximum flow in a related directed graph. As described in *Introduction to Algorithms* [2], given a regular bipartite multigraph $G = (V, E)$ where $V = L \cup R$ is the vertex partition and $n = |V|/2 = |L| = |R|$, we define a *corresponding flow network* $G' = (V', E')$. The set $V'$ includes each vertex in $V$ and two new vertices, a source $s$ and sink $t$. The set $E'$ comprises directed edges from $s$ to each vertex in $L$, edges in $E$ directed from $L$ to $R$, and edges from each vertex in $R$ to $t$. More precisely, $V' = V \cup \{s, t\}$ and $E' = \{(s, l)\} \cup \{(l, r) : (l, r) \in E\} \cup \{(r, t)\}$ for each $l \in L$ and $r \in R$. As shown in figure 1, we then assign a unit capacity to each edge, so that the maximum flow from $s$ to $t$ is a perfect matching of $G$ [2].

A side effect of our definition of $E'$ is that there are no multiedges in the corresponding flow network. This result is intentional: we are guaranteed that there still exists a perfect matching when we remove duplicate edges by Hall's Theorem [4]. Removing duplicate edges preserves the *neighborhood*, or set of adjacent vertices, of each vertex, which means the size of a maximum matching is unaltered.

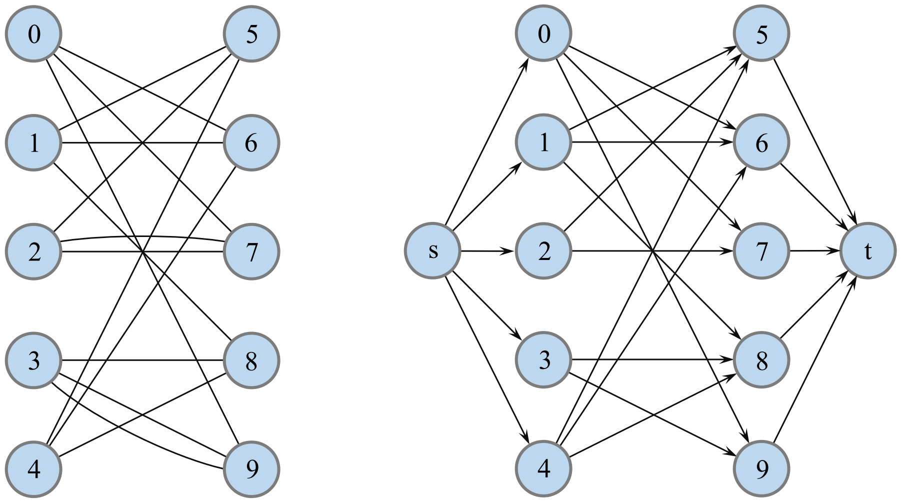

Figure 1: Example of a 3-regular bipartite multigraph $G$ (left) and its corresponding flow network $G'$ (right). Each vertex in $L$ is numbered 0 to $n-1$, and each vertex in $R$ is numbered $n$ to $2n-1$. Each edge in $G'$ is assigned unit capacity and initially has no flow. Note the absence of multiedges in $G'$.

One of the asymptotically fastest ways to solve the maximum-flow problem is the push-relabel method. This approach governs a class of algorithms that relaxes flow conservation during execution and instead maintains a preflow. Whereas *flow conservation*, as defined in algorithms such as the Ford-Fulkerson method [2], requires the total flow into each vertex (in-flow) to equal the total flow out of the vertex (out-flow), a *preflow* allows the in-flow of a vertex to exceed the out-flow of the vertex [2]. The method first "floods" a flow network with a preflow and then subsequently restores flow conservation using push and relabel operations. Though the method relaxes conservation of flow, it maintains the *capacity constraint* for each edge, which means that the flow in an edge never exceeds the capacity of the edge.

Push and relabel operations rely on several definitions. The *excess*, denoted by $u.e$, of a vertex $u$ equals its in-flow minus its out-flow. We also define the *residual network* of $G'$ to be a directed graph with *residual capacity* $c_f(u,v)$ equal to be the amount of flow that can be pushed from vertex $u$ to vertex $v$.[1] We also define the *height*,[2] denoted by $u.h$, of a vertex $u$ to be a non-negative integer such that the source height $s.h = |V'|$, the sink height $t.h = 0$, and $u.h \leq v.h + 1$ for every edge in the residual network $(u,v)$ with non-zero residual capacity.

Push-relabel algorithms may perform a *push operation* on an edge $(u,v)$ if vertex $u$ has excess, the residual capacity $c_f(u,v) > 0$, and $u.h = v.h + 1$. This operation reduces the excess of $u$ by $\Delta$ and increases the flow through $(u,v)$ by $\Delta$ such that $\Delta$ is maximized. Furthermore, the operation may leave $(u,v)$ *saturated* or *unsaturated* depending on whether the flow through the edge equals its capacity after the operation. Push-relabel algorithms perform a *relabel operation* on a vertex $u$ with excess if no incident edge meets the criteria for a push operation. The operation sets the height of $u$ to 1 plus the minimum height of its *residual neighbors*, or adjacent vertices $v$ for which the residual capacity $c_f(u,v) > 0$.

The push-relabel method "relabels" the heights of vertices to "push" flow downhill,[3] thereby resolving excess and restoring flow conservation. Indeed, push-relabel algorithms eliminate excess from each vertex and compute a maximum flow in $G'$ at termination.

Subject to the capacity constraint, a valid flow in this construction assigns each edge a flow of 0 or 1. Furthermore, a maximum flow assigns a flow of 1 to one edge in $E$ incident on each vertex of $V$. We construct a maximum matching of $G$ by noting the edges $(u,v) \in E$ for which there exists non-zero flow in the computed maximum flow [2]. The maximum matching must be a perfect matching since $G$ is regular and bipartite.

## 4 Computing an initial preflow

To run a push-relabel algorithm on $G'$, we first initialize a preflow. For a generic graph, we set the flow of each edge from the source to the edge's capacity, the height of each vertex in $V' - \{s\}$ to 0, and the height of $s$ to $|V'|$ [2]. We can be more intentional with our initialization of a preflow, however, because $G$ is regular and bipartite.

[1]The residual capacity of an edge $(u,v)$ present in the flow network $G'$ is the amount of additional flow we can push through $(u,v)$. By contrast, the residual capacity of the reverse edge $(v,u)$ is the amount of flow already in the edge—the amount of flow we can "cancel" if we push flow from vertex $v$ to vertex $u$. Under our construction of $G' = (V', E')$, which assigns unit capacity to all edges, the residual capacity of an edge is either 0 or 1. If the edge $(u,v) \in E'$ has a unit of flow, then we can push 0 more flow through $(u,v)$, but we can "cancel" the unit of flow by pushing back through the reverse edge $(v,u)$. Thus, the residual capacity of edge $(u,v)$ is 0, and the residual capacity of $(v,u)$ is 1.

[2]Here, we stick to naming convention in *Introduction to Algorithms, 3rd Edition* [2], which discusses "pushing" flow and "relabeling" vertex "heights."

[3]We use the term "downhill" to convey the intuition of water flowing in a river.

We begin building the initial preflow as mentioned before: we assign a unit of flow for each edge out of the source so that every edge $(s,l)$ is saturated for $l \in L$. We then augment this initial preflow by computing a maximal matching[4] of $G$. We assign a unit of flow from each matched vertex in $L$ to its matched vertex in $R$, as well as a unit of flow from the matched vertices in $R$ to the sink $t$. We also set the height of each vertex in the matching to 1, add a unit of excess to $t$ for each pair of matched vertices, and add a unit of excess to each vertex in $L$ not in the matching. Figure 2 shows an example of an initial-preflow computation.

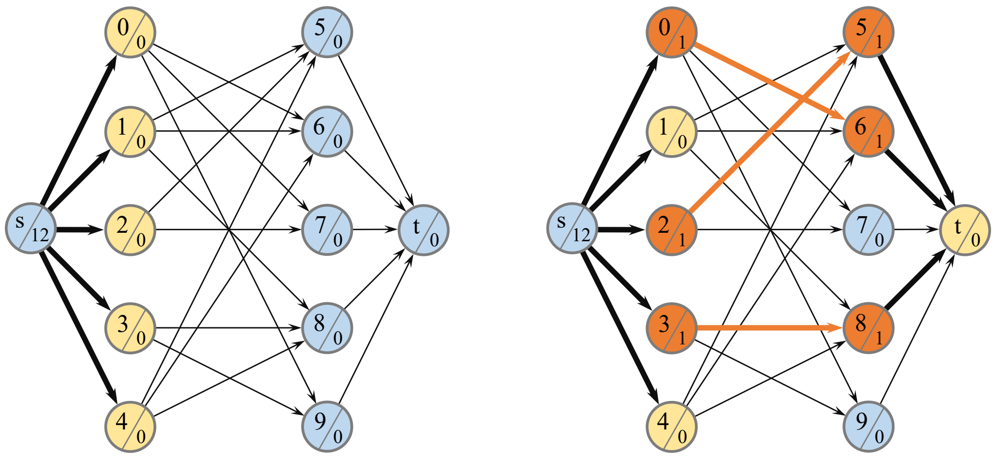

Figure 2: We assign a unit of flow for each edge out of the source (left) and compute an initial preflow (right) augmented by a maximal matching of edges $(0,6)$, $(2,5)$, and $(3,8)$ (colored in orange). We then assign a unit of flow through each edge in the matching, as well as a unit of flow through each edge that connects a matched vertex in $R$ to the sink. Edges drawn in bold are assigned a unit of flow (saturated), and vertices colored yellow have excess. The height of each vertex is shown below the vertex number label—note that each vertex incident on an edge in the matching is assigned a height of 1 (right).

This method produces a valid initial preflow by maintaining the capacity constraint for each edge in $E'$ and ensuring that out-flow never exceeds in-flow on any vertex $v \in V$. The process satisfies the capacity constraint since it assigns unit flow to edges in $E'$, which all have unit capacity. The process satisfies in-flow and out-flow conditions since a maximal matching is a bijective map $L \to R$. Each matched vertex in $L$ has unit in-flow from the source and unit out-flow to a vertex in $R$, and each matched vertex in $R$ has unit in-flow from a vertex in $L$ and unit out-flow to the sink. The unmatched vertices in $L$ have unit in-flow from the source and no out-flow, and the unmatched vertices in $R$ have no in-flow or out-flow.

Salting the initial preflow with a maximal matching allows for greater flexibility to decide a push-relabel algorithm's first order of operations. Furthermore, it allows us to compare matching heuristics in terms of matching size and subsequent push-relabel efficiency. The assumption is that more initially matched vertices means less excess, fewer push operations, and fewer relabel operations.

[4] A *maximal matching* is a matching that is not a subset of another (larger) matching. In particular, a maximum matching is always maximal, but the reverse does not always hold.

To test this idea, we compute the maximal matching in the initial preflow using either a greedy matching algorithm or the quickmatch algorithm [5]. Let $M$ be a set of edges in the matching, initially empty, and $U$ be the set of vertices not incident on an edge in $M$. We call the vertices in $U$ *unmatched*. In the greedy matching algorithm, we repeatedly add an edge to $M$ incident on two random adjacent vertices in $U$. Each iteration, we thus increase $|M|$ by 1 and reduce $|U|$ by 2. At termination, no adjacent vertices in $U$ remain, and we call the remaining vertices in $U$ *hermits*.

The quickmatch algorithm is still a greedy algorithm; however, it adds edges to $M$ more intelligently. In the quickmatch algorithm, we define the *unmatched neighborhood* of a vertex to be the set of vertices in $U$ that are adjacent to the vertex. We repeatedly select an unmatched vertex $u$ with the smallest nonempty unmatched neighborhood. Then, we choose a vertex $v$ from the unmatched neighborhood of $u$ so that $v$ has a minimum-size unmatched neighborhood. We add $(u,v)$ to $M$, increasing $|M|$ by 1 and decreasing $|U|$ by 2 until no such unmatched $u$ and $v$ remain. By prioritizing vertices with the smallest unmatched neighborhood, the quickmatch algorithm computes a near-maximum matching [5], which is typically larger than that computed by the greedy algorithm.

## 5 Push-relabel variants

As mentioned before, the push-relabel method governs a class of push-relabel algorithms. In this paper, we experiment with several variants of the push-relabel method to evaluate run-time efficiency. The relabel-to-front variant is popular because it offers an $O(V^3)$ asymptotic running time [2]. We also try a relabel-to-back variant and a relabel-to-random variant for comparison.

Each of these methods performs a *discharge* operation for vertices with excess, removing all excess at once with repeated *push* and *relabel* operations. They differ, however, in the order of vertex discharges. When a vertex is relabeled, the relabel-to-front algorithm processes the vertex first, the relabel-to-back algorithm processes it last, and the relabel-to-random algorithm processes it randomly. At termination, all variants are guaranteed to compute a maximum flow of $G'$ [2], and by extension, a perfect matching of $G$. See figure 3 for a sequence of push and relabel operations that produce a maximum flow.

Another two variants we test are a min-height and max-height selection method. These methods do not perform discharge operations, but instead order individual push and relabel operations based on vertex height. In the min-height variant, we order vertices with excess in a min-priority queue based on vertex height, performing either a push or relabel operation on a vertex of minimum height that has excess. By contrast, we choose a vertex with the maximum height to push or relabel in the max-height variant; correspondingly, we implement a max-priority queue based on vertex height.

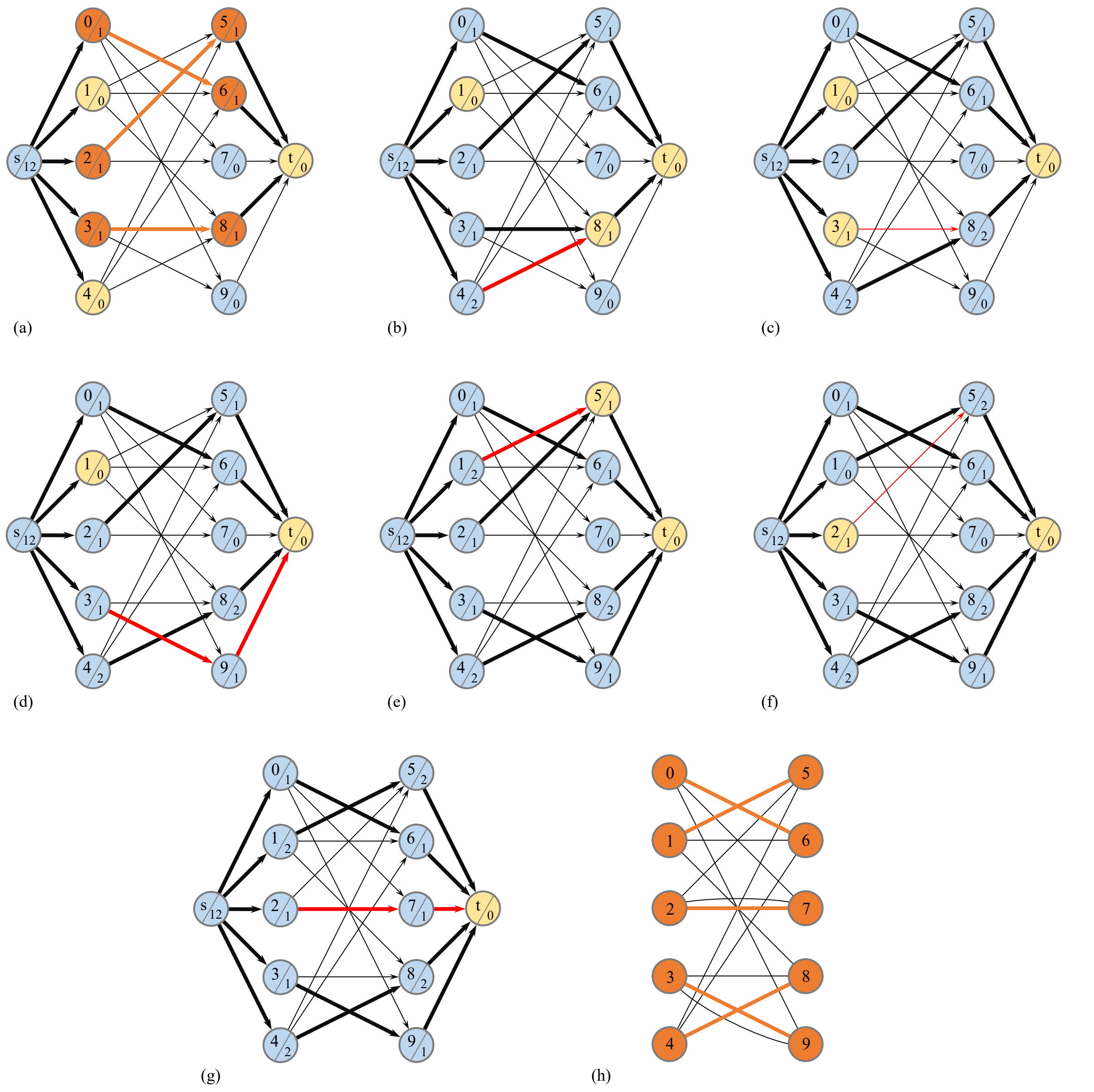

Figure 3: A sequence of push and relabel operations computes a perfect matching of a 3-regular bipartite graph using the relabel-to-front algorithm. As in figure 2, vertices from the original bipartite graph are numbered 0 to 9, along with the source vertex $s$ and sink vertex $t$. Vertices with excess are colored yellow, edges with non-zero flow are drawn in bold, and the height of each vertex is notated under the vertex number. Part (a) shows the initial preflow as computed in figure 2, with the maximal matching colored in orange. In subsequent parts (b) through (g), edges through which we push flow are colored in red. In part (b), we relabel vertex 4 and push a unit of flow through edge $(4,8)$. In part (c), we relabel vertex 8 and push a unit of flow back through edge $(8,3)$. In part (d), we push a unit of flow through edge $(3,9)$, relabel vertex 9, and push a unit of flow through edge $(9,t)$. In part (e), we relabel vertex 1 and push a unit of flow through edge $(1,5)$. In part (f), we relabel vertex 5 and push a unit of flow back through edge $(5,2)$. In part (g), we push a unit of flow through edge $(2,7)$, relabel vertex 7, and push a unit of flow through edge $(7,t)$. Part (g) achieves a maximum flow since no vertices besides the sink have excess. We relate the maximal flow to a perfect matching in figure (h), with the matched edges colored in orange.

## 6 Push-relabel theoretical analysis

The generic push-relabel algorithm performs three operations to produce a maximum flow: relabels, saturating pushes, and nonsaturating pushes [2]. A push operation is deemed *saturating* if it sets the residual capacity of an edge to 0 and is deemed *unsaturating* otherwise. Since all edge capacities in $E'$ are 1 and a push operation decreases the residual capacity of an edge by at least 1, every push operation on $G'$ is saturating. It suffices to bound the number of relabel and saturating-push operations.

We begin with a bound on relabel and saturating-push operations performed by a generic push-relabel method on a generic flow network $G = (V, E)$. Below, we present a general intuition for the bound as described in Chapter 26 of *Introduction to Algorithms* [2].

To bound the number of relabels, we assert that there exists a simple path in the residual network from any vertex with excess to the source (see Lemma 26.19 [2]). By virtue of this simple path and the fact that relabel operations apply to only vertices with excess, the maximum height of a vertex is the height $|V|$ of the source plus the maximum height increase over the simple path from $x$ to $s$, which is $|V|-1$. The height of each vertex in $V - \{s,t\}$ starts at 0 and never decreases, and a relabel operation increases the height of a vertex by at least 1. Thus, the maximum number of relabels on a vertex is $2|V|-1$, which gives an $O(V^2)$ bound on the number of relabels performed on all vertices [2]. We perform a relabel operation by iterating through each of a vertex's $O(V)$ neighbors, which leads to $O(V^3)$ time spent relabeling vertices.

To bound the number of saturating pushes, we observe that every saturating push over a given edge is separated by at least one relabel. Thus, each edge may undergo only as many pushes as its incident vertices can perform relabels. There are two vertices incident on each edge, which implies that each edge in $E$ undergoes fewer than $2|V|$ saturating-push operations. This relation bounds the total number of saturating-push operations by $O(VE)$. We perform a push operation using $O(1)$ computations, which leads to $O(VE)$ time spent performing saturating pushes.

Assuming that $G$ is a corresponding flow network constructed from a $d$-regular multigraph, each vertex in $V - \{s,t\}$ has $O(d)$ neighbors. We never relabel the source or sink, which means we spend $O(d)$ time during a relabel operation. We can thus implement a push-relabel algorithm to run in $O(VE)$ time for these corresponding flow networks. During a relabel operation on a vertex $u$, we find the minimum height of the vertex's residual neighbors, update the height of the vertex, and then maintain an attribute for each neighbor $v$ indicating whether a push operation can be performed form $v$ to $u$. We store this attribute in a way that allows a push operation to check for a viable adjacent vertex in constant time. Furthermore, we can implement the greedy maximal-matching or quickmatch algorithm to compute an initial preflow in $O(E)$ time.

Theoretically, our understanding of the push-relabel algorithm is limited to $O(VE)$ running time for regular bipartite multigraphs. Experimentally, however, we could not reproduce this worst-case scenario. Instead, by generating random bipartite graphs, computing an initial preflow with a maximal matching, and running a push-relabel variant, our tests indicate an $O(V)$ bound on the total number of relabel operations and an $O(E)$ bound on the total number of push operations.

## 7 Experimental results

We tested push-relabel algorithms on randomly generated $d$-regular bipartite multigraphs for varying graph size $|V|$ and varying degree $d$. To generate the graphs, we assign each vertex in $L$ a label from 0 to $n-1$ and each vertex in $R$ a label from $n$ to $2n-1$. We then insert edges based on a random permutation $\pi$ of the integers 0 to $n-1$. For each vertex $l \in L$ with a vertex label $i$, we add an edge $(l,r)$, where $r \in R$ and $r$ has the label $\pi(i)+n$. Given the desired vertex degree $d$, we perform $d$ of these permutation-based edge assignments, accumulating the edges to form a $d$-regular bipartite multigraph. Since a permutation of vertices in $L$ to vertices in $R$ is equivalent to a perfect matching, we can otherwise think of this process as constructing the $d$-regular graph by superimposing $d$ randomly generated perfect matchings. Our results focus on cases where $d$ is odd, since Euler decomposition provides an $O(E)$ time bound where $d$ is even.

Our goal is to experimentally bound the number of relabels performed during the computation of a maximum flow. As we have shown in our theoretical analysis, we only perform relabel operations and saturating-push operations, and the number of relabel operations also bounds the number of saturating-push operations. To this end, we randomly generate graphs of varying $d$ and $|V|$ and perform an experimental amortized analysis on the relabel operations carried out. In the interest of emulating worst-case performance, we randomly generate and find a maximum flow of 100 graphs; we show only the maximum of the average number of relabel operations from those 100 tests.

We begin by running the relabel-to-front algorithm on graphs with initial preflows computed using the greedy maximal matching algorithm. In figure 4, we note that the average number of relabel operations performed stays relatively constant with increasing graph size. In fact, for five orders of magnitude of $|V|$, the average number of relabel operations is under 2. Moreover, the relabel cost appears inversely proportional to $d$, with the average number of relabels approaching 0.1 as $d$ increases. This result suggests that highly connected graphs of a fixed size require less push-relabel computation than their less-connected counterparts.

We can see from figure 5 that computing the initial preflow with a maximal matching produced by quickmatch greatly increases the fraction of matched vertices over the greedy method. By computing an initial preflow with a maximal matching, we are able to match many of the vertices using the greedy approach and nearly all of the vertices using the quickmatch algorithm. We also note that the quickmatch algorithm often computes a maximum matching for small $|V|$ and large $d$, leading to the near zero relabel operations on average.

Additionally, the matching performance of both greedy and quickmatch algorithms approaches 1 as $d$ increases. A matching performance of 1 is equivalent to a maximum (perfect) matching, so as performance approaches 1, we say that the algorithm finds a "near-maximum" matching. From figure 5, we observe that the quickmatch algorithm always produces a near-maximum matching; however, the greedy algorithm produces a near-maximum matching only on graphs with high degree. Larger graph size does not appear to change matching performance on average, but does seem tighten the performance distribution. This phenomenon is likely a result of normalization: a small variance in maximal matching size is more significant when normalizing by $10^2$ than when normalizing by $10^6$.

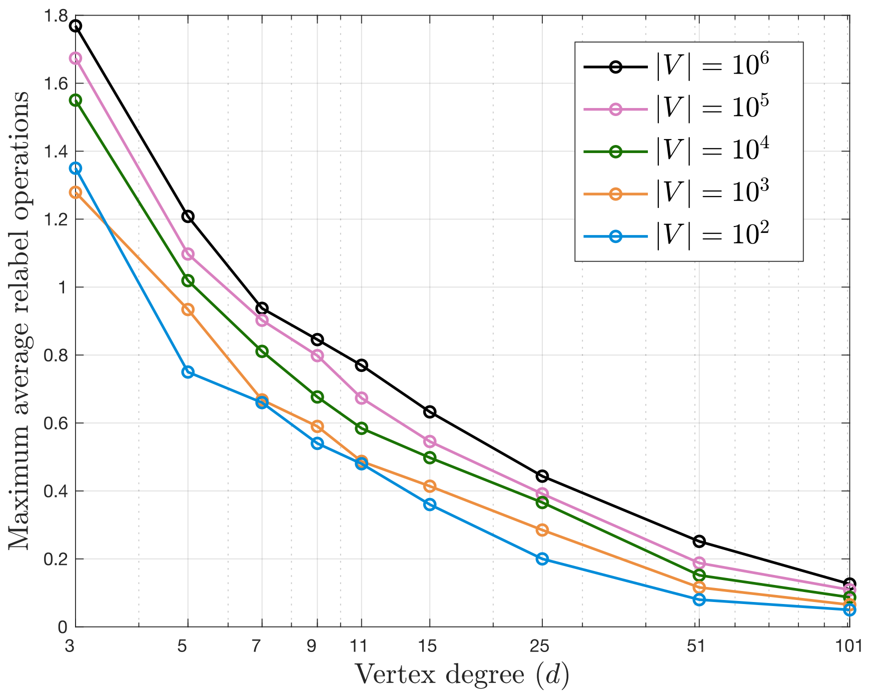

Figure 4: For each value of $d$ and $|V|$, we plot the maximum average number of relabel operations over 100 randomly generated graphs. We compute the initial preflow of each graph with a greedy maximal matching and then run the relabel-to-front algorithm to find a maximum flow. We average the number of relabel operations performed over all vertices and plot the maximum of these averages for each value of $d$ and $|V|$.

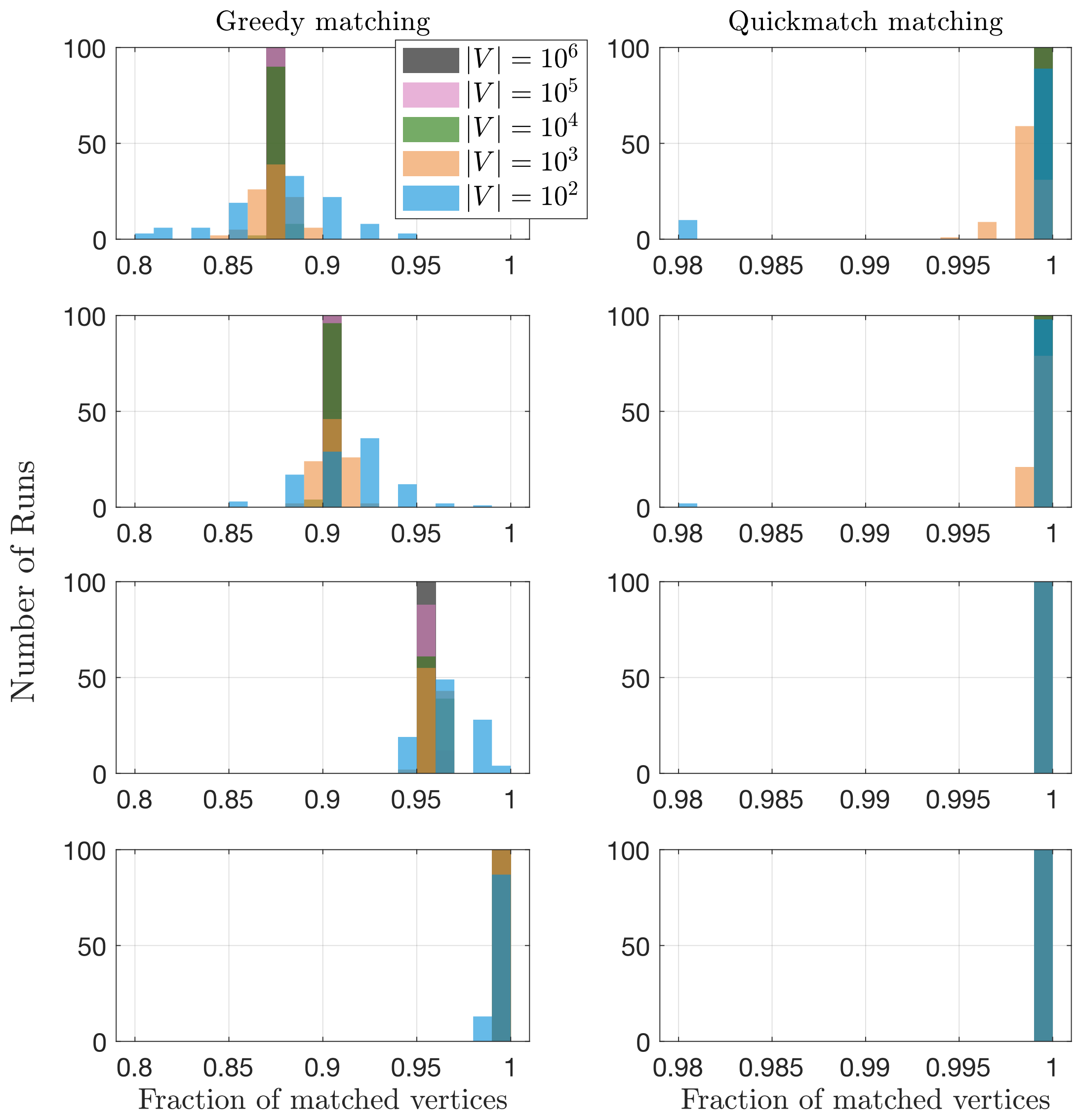

Figure 5: We compute maximal matchings of randomly generated graphs using the greedy algorithm and the quickmatch algorithm to view their relative matching performance. We measure matching "performance" by the fraction of (matched) vertices incident on an edge in the maximal matching. We then plot a distribution of the performance over 100 randomly generated graphs for each value of $d$ and $|V|$. Each row of the figure corresponds to a different graph degree: from top to bottom, we graph matching performance for $d = 3$, $d = 5$, $d = 15$, and $d = 101$. Note that the $x$-axis scales for the greedy algorithm (left column) and quickmatch algorithm (right column) are different: the worst-case performance for quickmatch matchings is 0.98, whereas the worst-case performance for greedy matchings is 0.8.

We next run the relabel-to-front algorithm on graphs with preflows initialized using the quickmatch algorithm. Again, we vary $d$ and $|V|$ and perform an experimental amortized analysis of relabel operations. In figure 6, we note similar trends as in figure 4: the worst-case average of relabel operations is bounded by 1.2. As $d$ increases, however, the quickmatch algorithm improves push-relabel performance over the greedy algorithm. Notably, for small, highly connected graphs where $|V| \leq 10^3$ and $d \geq 9$, the quickmatch algorithm computes a maximum matching, leading to no relabel computations after the initial preflow.

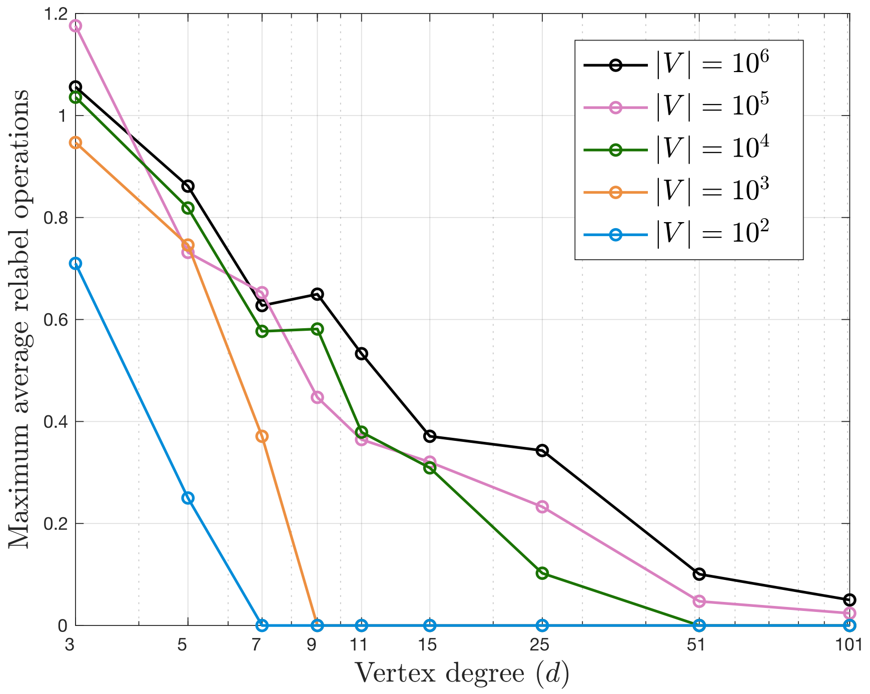

Figure 6: For each value of $d$ and $|V|$, we plot the maximum average number of relabel operations over 100 randomly generated graphs. We compute an initial preflow of each graph with the quickmatch algorithm and then run the relabel-to-front algorithm to find a maximum flow. We then average the number of relabel operations performed over all vertices and plot the maximum of these averages for each value of $d$ and $|V|$.

We perform the same analysis as in figures 4 and 6 with the relabel-to-back and relabel-to-random variants in figure 7. From this figure, we don't see significant differences between the performance of push-relabel variants. The relabel-to-front variant appears to perform slightly better than the others; however, each variant performs under 2 relabel operations on average per vertex across randomly generated graphs of all degrees and sizes.

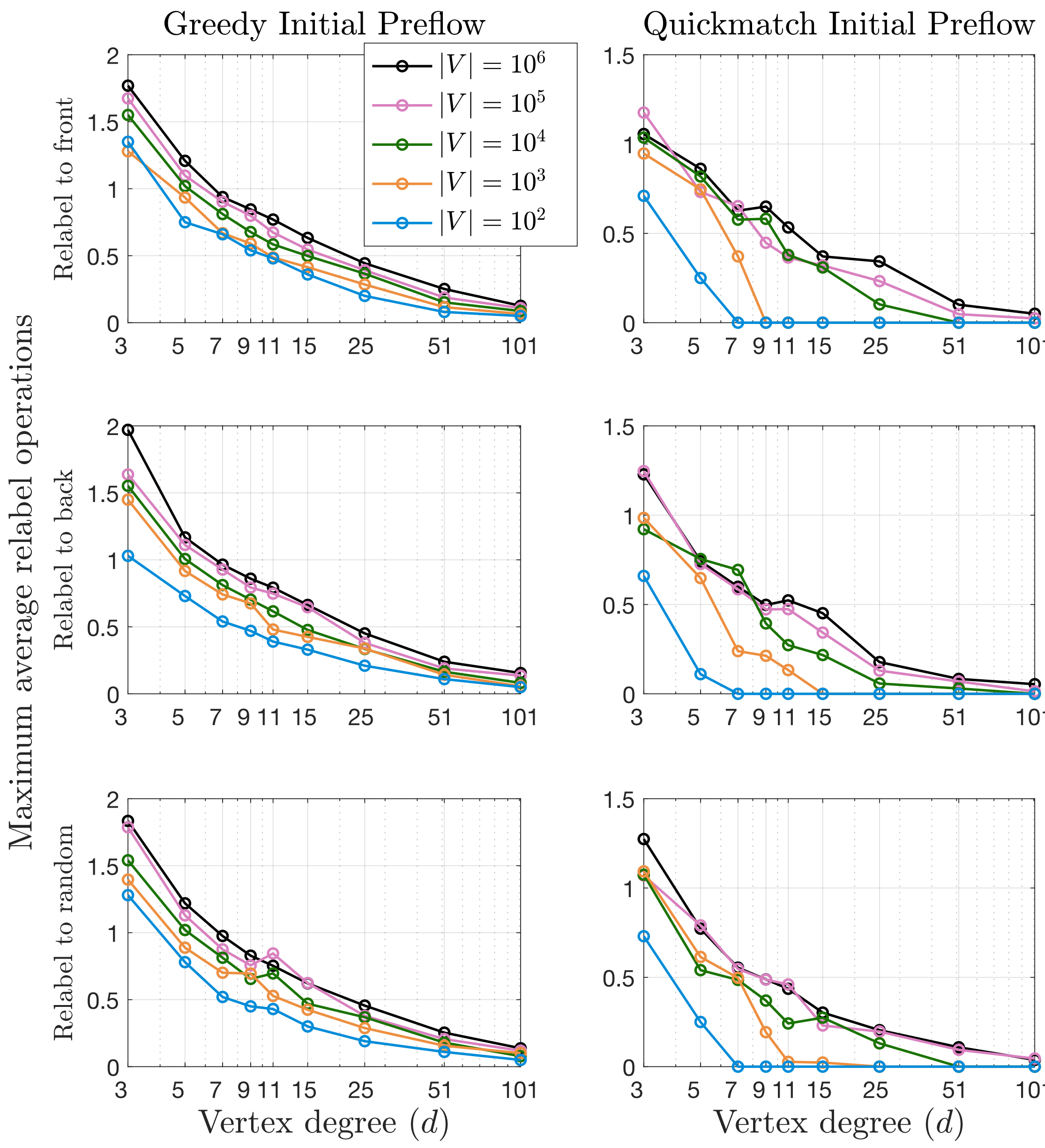

Figure 7: For each value of $d$ and $|V|$, we plot the maximum average number of relabel operations over 100 randomly generated graphs. We compute an initial preflow of each graph with the greedy (left column) and quickmatch (right column) maximal matching algorithms and then run the relabel-to-front (top row), relabel-to-back (middle row), and relabel-to-random (bottom row) algorithms to find a maximum flow. Afterwards, we average the number of relabel operations performed over all vertices, and we plot the maximum average over the 100 tests performed for each value of $d$ and $|V|$. Each data point plotted thus represents the worst-case performance over 100 push-relabel tests. The entire figure represents these tests for 9 values of $d$, 5 values of $|V|$, 3 variants of the push-relabel method, and 2 algorithms for salting the initial preflow—a total of 27000 randomly generated graphs and maximum-flow networks.

Similarly to figure 7, we analyze the worst-case number of relabels for the min-height and max-height variants in figure 8. As in figure 7, there are no major differences between push-relabel variants. Each variant performs under 2 relabel operations on average per vertex across several magnitudes of graph size and vertex degree. The relabel-to-front variant appears to have a slight edge over these methods as well.

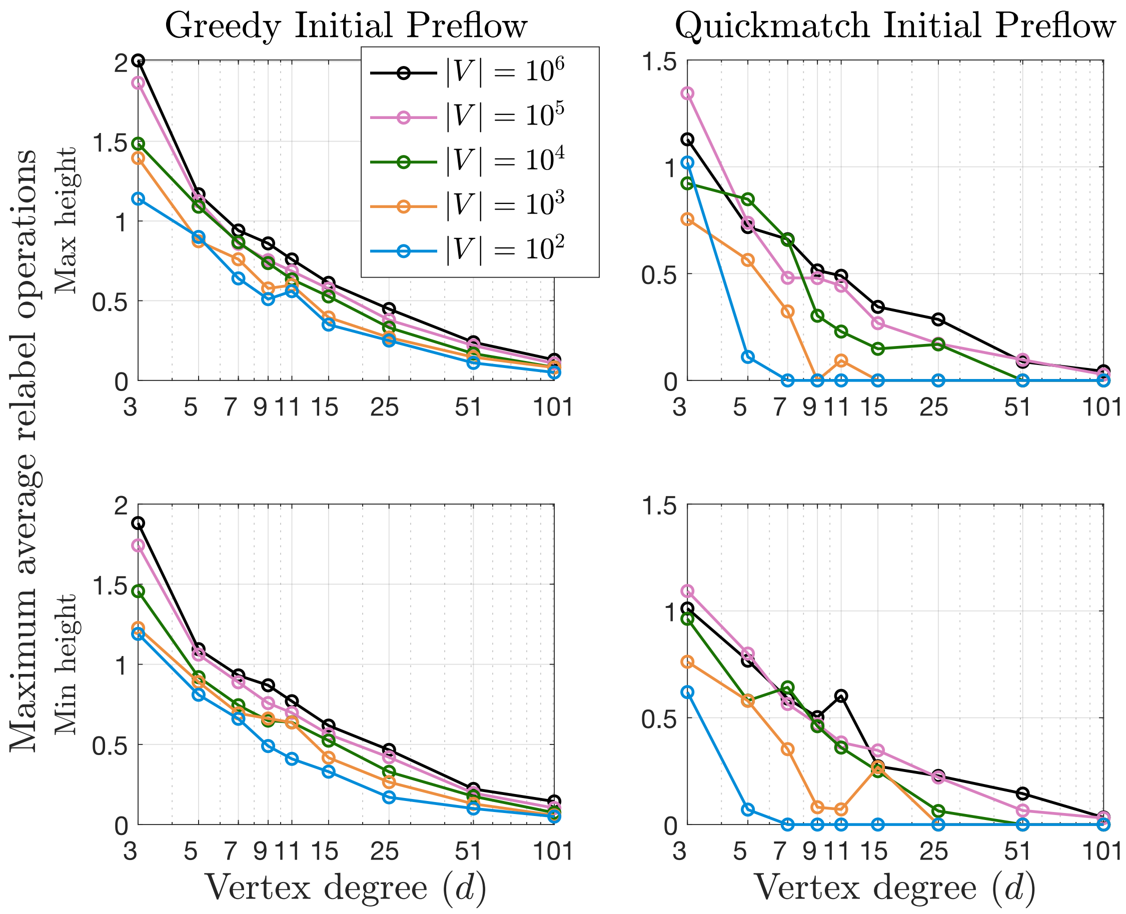

Figure 8: For each value of $d$ and $|V|$, we plot the maximum average number of relabel operations over 100 randomly generated graphs. We compute an initial preflow of each graph with the greedy (left column) and quickmatch (right column) maximal matching algorithms and then run the max-height (top row) and min-height (bottom row) algorithms to find a maximum flow. Afterwards, we average the number of relabel operations performed over all vertices, and we plot the maximum average over the 100 tests performed for each value of $d$ and $|V|$. Each data point plotted thus represents the worst-case performance over 100 push-relabel tests. The entire figure represents these tests for 9 values of $d$, 5 values of $|V|$, 2 variants of the push-relabel method, and 2 algorithms for salting the initial preflow—a total of 18000 randomly generated graphs and maximum-flow networks.

Our tests suggest that the relabel-to-front algorithm can find a maximum flow in a $d$-regular graph using $O(1)$ amortized relabel operations per vertex, or $O(V)$ relabel operations in total. By our theoretical analysis of the push-relabel algorithm, this finding also experimentally bounds the total number of amortized push operations to $O(E)$. Since we implement push operations to run in constant time and relabel operations to run in $O(d)$ time, we conclude that the push-relabel algorithm often computes a perfect matching of a regular bipartite multigraph in $O(E)$ time.

## 8 Conclusions

Our experiments indicate that several variants of the push-relabel method can compute a perfect matching of a regular bipartite multigraph in $O(E)$ time. We achieve this result with a multi-stage algorithm that constructs the corresponding flow network of a regular bipartite multigraph, salts the network's initial preflow with a maximal matching, finds a maximum flow in the network using push and relabel operations, and extracts a perfect matching from the maximum flow.

This algorithm has several advantages over a vanilla push-relabel algorithm. In particular, salting the initial preflow with a maximal matching greatly improves push-relabel efficiency, as we eliminate a relabel operation for each vertex covered by the maximal matching. For this reason, we employ the quickmatch algorithm [5] to obtain a larger matching size and further reduce the number of relabel operations we perform. Moreover, as $d$ increases, we expect larger matchings that cover nearly all vertices. In the special case that we salt the initial preflow with a maximum matching, we eliminate push and relabel operations entirely.

Unfortunately, our current understanding of these push-relabel methods is limited to empirical observations. In the context of finding a perfect matching using the push-relabel method, we are unsure if these results translate to an $O(E)$ worst-case theoretical bound.

## 9 Acknowledgements

Most importantly, I'd like to thank Professor Thomas Cormen for his invaluable guidance and unwavering support. Professor Cormen is immensely dedicated to his endeavors, and I am grateful that he added me to his busy schedule. His ideas, time, and wit were necessary for the completion of this project. Whether they were sufficient, I leave for the reader to decide.

In addition, I'd like to acknowledge several of Professor Cormen's former thesis students, whose well written insight into the perfect matching problem was invaluable to the newcomer. Andrew Hannigan, Stefanie Ostrowski, and Patricia Neckowicz each detailed a unique and compelling approach to the problem. This project stands on the shoulders of giants.

## References

[1] Richard Cole, Kirstin Ost, and Stefan Schirra. Edge-coloring bipartite multigraphs in $O(E \log d)$ time. *Combinatorica*, 21(1):5–12, 2001.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. The MIT Press, 2009.

[3] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. Perfect matchings in $O(n \log n)$ time in regular bipartite graphs. *STOC'10: Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, pages 39–46, 2010.

[4] Philip Hall. On representatives of subsets. *London Mathematical Society*, 1935.

[5] Andrew Hannigan. A heuristic algorithm for computing edge colorings on regular bipartite multigraphs. Technical Report 769, Dartmouth College Computer Science, 2013.

[6] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):255–231, 1973.

[7] Patricia Neckowicz. Two algorithms for finding edge colorings in regular bipartite multigraphs. Technical Report 771, Dartmouth College Computer Science, 2015.

[8] Stefanie Ostrowski. Chain match: An algorithm for finding a perfect matching of a regular bipartite multigraph. Technical Report 753, Dartmouth College Computer Science, 2014.

[9] Douglas B. West. *Introduction to Graph Theory, 2nd Edition*. Pearson Education, 2001.

## 10 Appendix: Maximal matching lower bound

Let $G = (V, E)$ be a $d$-regular bipartite multigraph, and let $M \subseteq E$ and $M^* \subseteq E$ be a maximal and maximum matching of $G$, respectively. In bipartite graphs, the König-Egerváry Theorem [9] implies $|M| \geq |M^*|/2$. Since $G$ is $d$-regular, we know that $M^*$ is a perfect matching, i.e., that $|M^*| = n$ where $n = |V|/2$ is the number of vertices in each partition of $G$. This property leads to the result $|M| \geq n/2$, and we conclude that any maximal matching must cover at least half of the vertices of $G$.

A proof of a result mentioned in [5] improves this bound to $|M| \geq nd/(2d-1)$. Suppose that the maximal matching $M$ covers $m$ vertices, so that $|M| = m/2$. Then, there are $h = |V| - m$ hermits not covered by the matching. Each vertex has degree $d$, and since $M$ is maximal, no two hermits share an edge. Therefore, each of the $h$ hermits has $d$ edges incident on matched vertices. Correspondingly, each of the $m$ matched vertices has $d$ incident edges, at most $d-1$ of which may be incident on hermits. The sum of edges from hermits to matched vertices must equal the sum of edges leaving matched vertices to hermits, which leads to the following inequality:

$$\sum_{i=1}^{m}(d-1) \geq \sum_{i=1}^{h} d \Rightarrow m \geq hd/(d-1) \ .$$

Substituting $n = (m+h)/2$ implies that $|M| \geq nd/(2d-1)$.