5-29-2013

An Algorithm for Computing Edge Colorings on Regular Bipartite Multigraphs

Andrew S. Hannigan
Dartmouth College

Dartmouth Computer Science Technical Report TR2015-769

An Algorithm for Computing Edge Colorings on Regular Bipartite Multigraphs

Andrew Hannigan

May 29, 2013

Abstract

In this paper, we consider the problem of finding an edge coloring of a d-regular bipartite multigraph with $2n$ vertices and $m = nd$ edges. The best known deterministic algorithm (by Cole, Ost, and Schirra) takes $O(m \log d)$ time to find an edge coloring of such a graph. This bound is achieved by combining an $O(m)$-time perfect-matching algorithm with the Euler partition algorithm. The $O(m)$ time bound on the Cole, Ost, and Schirra perfect-matching algorithm has been shown to be optimal.

In this paper we present an alternative perfect-matching algorithm called QuickMatch. Empirical analysis shows that QuickMatch finds a perfect matching in $O(m)$ time in the average case. The QuickMatch algorithm allows us to compute an edge coloring in $O(m \log d)$ time in the average case. Due to its simplicity, the presented method is easy to implement and the constants in the time bound are small. Because of these features, QuickMatch is a highly practical and competitive method for finding edge colorings of d-regular bipartite multigraphs.

1 Introduction

This paper presents a heuristic algorithm for computing optimal edge colorings on d-regular bipartite multigraphs. A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two disjoint subsets L and R such that every edge $e \in E$ connects a vertex in L to a vertex in R. We may also denote a bipartite graph as $G = (L, R, E)$. A graph is *d-regular* if every vertex has exactly d incident edges. Because G is regular, $|L| = |R| = n$ and $|E| = nd$. A *multigraph* is a graph that is permitted to have multiple edges between the same two vertices.

An *edge coloring* of a graph $G = (V, E)$ is a mapping $E \to C$ that assigns a color $c(e) \in C$ to every edge $e \in E$ such that for every vertex $v \in V$, all edges incident on v have different colors. The objective of the edge-coloring problem is to compute an edge coloring using the minimum possible number of colors. In the bipartite case, the edge-coloring problem is closely related to the problem of computing a perfect matching. A *matching* $M \subseteq E$ is a set of edges with the property that every vertex in V appears on the endpoint of at most one edge in M. A *perfect matching* is a matching in which every vertex in V appears on

the endpoint of exactly one edge. By Hall's Theorem, a d-regular bipartite graph can be decomposed into d disjoint perfect matchings [2]. Decomposing a d-regular bipartite graph into d perfect matchings defines a minimum edge coloring of d colors.

In order to compute an edge coloring, it suffices to find a decomposition of d disjoint perfect matchings. When the degree of the graph is even, we partition the edges in E by finding an Euler orientation of G and performing an Euler split of G. An *Euler orientation* of a graph G with even degree is an orientation of all edges such that each vertex has an in-degree of $d/2$ and an out-degree of $d/2$ [12]. Past literature on perfect matchings suggests computing an Euler orientation by finding an Euler tour on G and orienting every edge by tracing the Euler tour [4,6,12]. It is actually only necessary, however, to partition the graph into edge-disjoint sets of cycles $\{C_1, C_2, \ldots, C_i\}$ such that $C_1 \cup C_2 \cup \cdots \cup C_i = E$. Because no vertex is at the end of an open path, we can produce an Euler orientation of the graph by tracing each of these cycles instead of a full Euler tour. This method is a simpler way to find a Euler orientation. Once an Euler orientation of G is found, we perform an Euler split of G. An *Euler split* creates two new graphs G_L and G_R using an Euler orientation of graph G. Both G_L and G_R share the same vertex set as G. The edges in G_L consist of the edges in the Euler orientation of G that are directed from R to L, and the edges in G_R consist of the edges in Euler orientation of G that are directed from R to L. Both G_L and G_R are now $d/2$-regular graphs.

The Euler-split method implies that if we start with a graph G with a power-of-2 degree then we can repeat the method until we have d 1-regular graphs, at which point finding d perfect matchings in G is trivial. We name this process an *Euler decomposition*. The edges of a 1-regular graph form a perfect matching. If d is not a power of 2, there will be a point at which we come to an odd-degree graph with no possible Euler orientation. If we had a method of computing a single perfect matching, however, then we could remove one matching from the odd-degree graph and continue the Euler decomposition on the resulting even-degree graph. Thus, if we have a method for computing a perfect matching in a d-regular bipartite graph, we can use Euler decomposition to compute a optimal edge coloring for any d-regular bipartite graph.

The edge-coloring problem and the perfect-matching problem for bipartite graphs are both well studied. Perhaps the most well known algorithm for computing perfect matchings is the Hopcroft-Karp algorithm, which terminates in $O(E\sqrt{V})$ time [8]. A long history of published papers on perfect matchings led to a paper by Cole, Ost, and Schirra in 2001, which achieves an $O(E)$ time bound [4]. A different paper showed that this bound is optimal for deterministic algorithms [6]. The $O(E)$ time bound on finding a perfect matching implies an $O(E \log d)$-time algorithm for computing an edge coloring using Euler decomposition. However, to the knowledge of one of the authors (Cole), there is no known implementation of the Cole, Ost, Schirra algorithm [3].

We classify the methods presented in this paper as heuristics because theoretical analysis did not yield a complete theoretical bound on the running time of these methods. This paper presents an empirical analysis and shows that on average these methods find a perfect matching in $O(E)$ time.

2 QuickMatch Heuristic

This section presents QuickMatch, a heuristic algorithm for computing a near-maximum matching in a d-regular bipartite graph G. QuickMatch is a greedy algorithm that inserts edges into a matching M until M is maximal[1] with respect to G. The rule for selecting the next edge in the matching emerged from the following observations.

If M is a matching with respect to $G = (L, R, E)$, we will define $G_M = (L_M, R_M, E_M)$. Let L_M and R_M denote the sets of vertices in L and R, respectively, that are unmatched by M. E_M denotes the set of edges $e \in E$ that connect a vertex in L_M to a vertex in R_M. We call G_M the *unmatched subgraph* of G with respect to M, and we say that L_M and R_M denote the *unmatched vertices* of G. Note that an edge e can be inserted into the matching M if and only if $e \in E_M$. Additionally, for $X \subseteq V_M$, we let $NB(X, G_M)$ denote the neighborhood of X in G_M. If there is a vertex v with an empty neighborhood in G_M, then we say that v is a *hermit*. A hermit will be unmatched by any matching containing the current matching. QuickMatch emerged from the intuition that we should insert edges into the matching in such a way that avoids creating hermits in G_M.

A natural way to prevent hermits would be to simply select the edge $e \in E_M$ that connects two vertices that have the smallest neighborhoods in G_M. By inserting e into the graph, the two vertices s and t that are closest to becoming hermits are matched and removed from G_M. We considered two possible ways to implement this strategy. The first way selects the edge $(s, t) \in E_M$ that minimizes $|NB(s, G_M)| + |NB(t, G_M)|$. The second way selects the vertex $s \in V_M$ with the smallest nonempty neighborhood and matches it to its neighbor t with the smallest neighborhood. Early testing showed that this second way, QuickMatch, outperforms the first way. Additionally, QuickMatch can easily be implemented to run in $O(E)$ time. Thus, the present paper focuses on QuickMatch:

QuickMatch(V, E)
1 $M \leftarrow \emptyset$
2 **while** there exists a vertex $v \in V_M$ such that $NB(v, G_M) \neq \emptyset$
3 **do** Let s be the vertex in G_M with the smallest nonempty neighborhood
4 Let t be the vertex in $NB(s, G_M)$ with the smallest neighborhood
5 $M \leftarrow M \cup (s, t)$

3 QuickMatch Analysis

In this section, we analyze the solution quality of QuickMatch empirically, and we analyze the running time of QuickMatch theoretically. We tested QuickMatch using randomly generated, d-regular bipartite multigraphs for varying sizes of d and n. These are labeled graphs, in which all vertices in L are labeled 0 to $n-1$, and all nodes in R are labeled n to $2n-1$. The graphs are generated by randomly permuting an array containing d copies of each vertex in R using a Knuth shuffle [10]. To build the graph, the first d vertices of the permuted array are inserted into the adjacency list of vertex 0, the next d vertices of the permuted array are

[1]Set S is maximal with respect to some property if there is no set $T \supset S$ that also possesses that property.

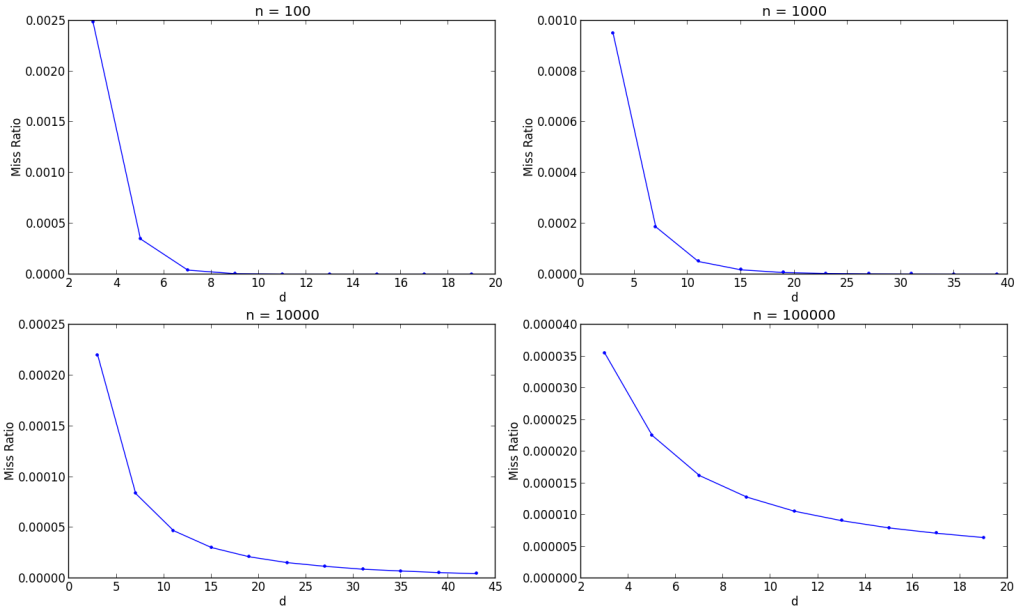

Figure 1: The average miss ratio for increasing degree. For four different sizes of n, this figure shows that the QuickMatch miss ratio decreases as d increases. The plot for $n = 100$ represents 5,026,103 trials, $n = 1000$ represents 2,565,357 trials, $n = 10000$ represents 914,178 trials, and $n = 100000$ represents 759,321 trials. In all cases, $d = 3$ was the worst-case miss ratio.

inserted into the adjacency list of vertex 1, and so on. For small values of d, this method of graph generation is known to have a near-uniform distribution over all possible graphs [13]. We measure solution quality of QuickMatch by the *miss ratio*, $(n - |M_{QM}|)/n$, where M_{QM} is the matching returned by QuickMatch.

Average Matching Size

First, we tested QuickMatch on random d-regular bipartite graphs with varying degree and a constant number of vertices. We considered only graphs of odd degree because we need to compute perfect matchings in only graphs of odd degree when finding a Euler decomposition. Figure 1 shows how QuickMatch performed on average, and Figure 2 shows the worst-case performance. The plot for $n = 100$ represents 5,026,103 trials, $n = 1000$ represents 2,565,357 trials, $n = 10000$ represents 914,178 trials, and $n = 100000$ represents 759,321 trials. We see that for every case, as the degree of the graph increases, the average miss ratio of QuickMatch approaches 0.

Next, we tested QuickMatch for increasing values of n. Figures 3 and 4 each represent 1,720,650 trials of data. Because we observed that the miss ratio of QuickMatch tends to decrease as d increases, we analyze the worst case where $d = 3$ when examining how the miss ratio is affected by increasing sizes of n. Testing showed that as n increases, the miss ratio of QuickMatch decreases as well. For larger values of d, the miss ratio decreased more

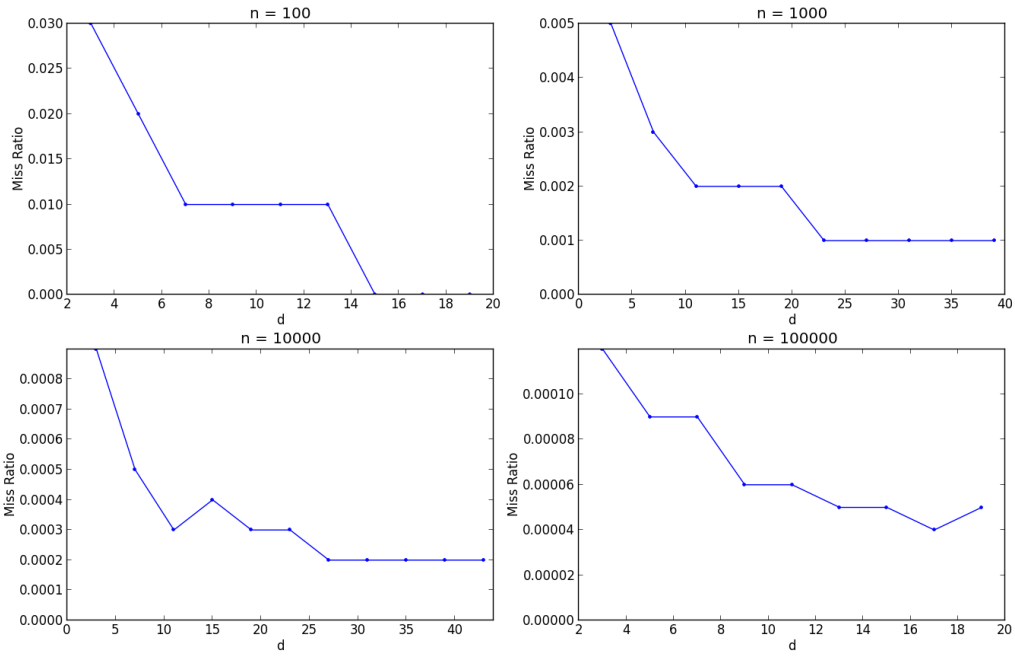

Figure 2: The worst-case miss ratio for increasing degree. For four different sizes of n, this figure shows that the QuickMatch miss ratio tends to decrease as d increases.

rapidly. It may seem counterintuitive that QuickMatch would perform better as n increases. This result, however, is supported by a theorem published by Alan Frieze [5]. For a random bipartite graph $B^{\delta\geq 2}_{n,cn}$ with $2n$ vertices and cn edges where $c > 2$ and δ denotes the minimum degree in B,

$$\lim_{n\to\infty} \Pr(B^{\delta\geq 2}_{n,cn} \text{ has a perfect matching}) = 1 \ .$$

The implications of this theorem match the performance of QuickMatch. Figure 5 shows the number of number of missed matches as n increases. Normalization tests suggest that there were $O(n^{0.4})$ hermits on average.

Running Time

We implemented QuickMatch to run in $O(E)$ time. It runs for at most n iterations and, during each iteration, it takes at most d steps to find the vertex t in $NB(s, G_M)$ with the smallest neighborhood by maintaining an array that holds the neighborhood sizes for each vertex. Thus, we must define a way of identifying the vertex s with the smallest nonempty neighborhood in $O(E)$ amortized time. This time bound is possible if we maintain d linked lists representing each of the possible non-zero neighborhood sizes of a vertex in the graph. We say that $list(i)$ will contain all vertices in V_M with a neighborhood of size $i = 1, 2, \ldots, d$, and we call i the *index* of $list(i)$. At the start of the algorithm, we add every vertex in

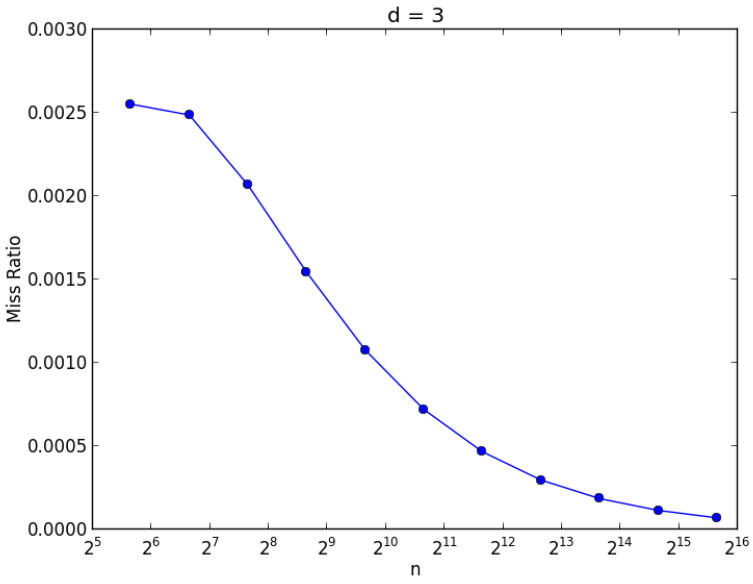

Figure 3: The average miss ratio as n increases for graphs of degree 3. 1,720,650 trials are represented in this plot.

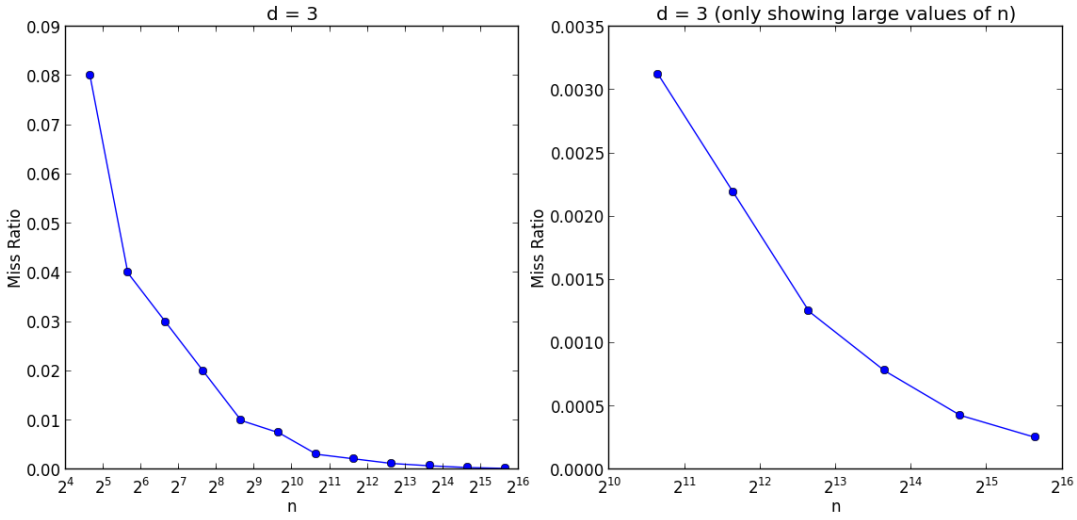

Figure 4: The worst-case miss ratio as n increases for graphs of degree 3.

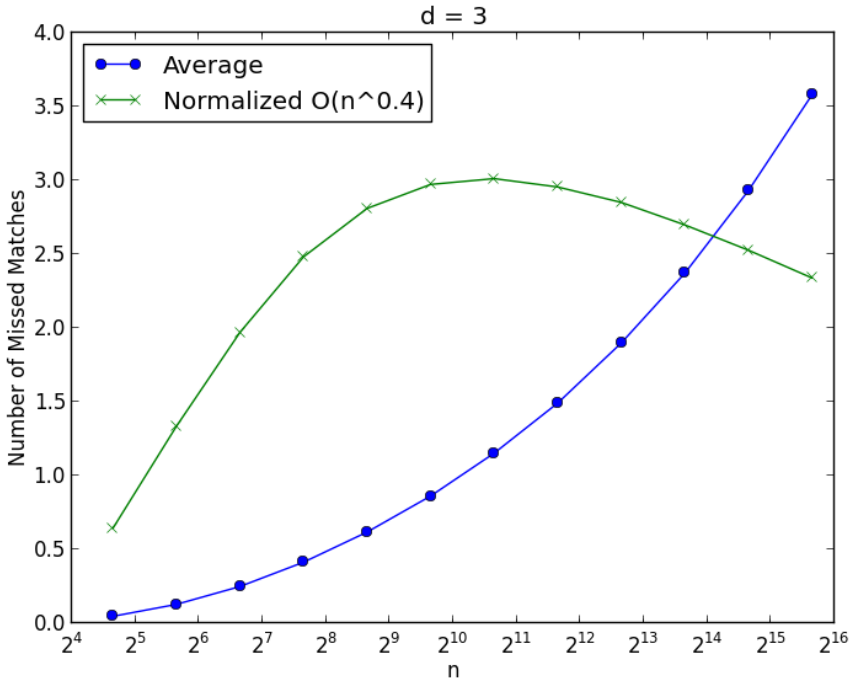

Figure 5: The average number of hermits in a graph where $d = 3$ and n is increasing. We also show a plot of this data that has been normalized by $0.02n^{0.4}$. Because this normalized plot has a negative slope for sufficiently large n, we know that the observed rate at which the average number of hermits increases is $O(n^{0.4})$.

the graph to the linked list corresponding to its neighborhood size. Note that because we allow for multiedges, not all vertices will be placed into $list(d)$ at the start of the algorithm. To find the vertex $s \in V_M$ with the smallest nonempty neighborhood in the graph, select a vertex from the nonempty list with the lowest index. When s and t have been selected, we delete s and t from their respective lists and move every vertex $v \in NB(s, G_M) \cup NB(t, G_M)$ down to the list with the next lowest index, which takes at most $2d$ steps. We also update the the array holding the neighborhood size of every vertex in the graph. Vertices moving down from $list(1)$ will become hermits, and so they no longer appear in any list. Finally, we insert (s, t) into the matching M. Each iteration takes $O(d)$ time to find the lowest-indexed nonempty list and select s, $O(d)$ time to select t from $NB(s, G_M)$, $O(d)$ time to update the neighborhood linked-lists and correct the neighborhood array, and $O(1)$ time to insert (s, t) into the matching. Because there are $O(n)$ iterations, QuickMatch runs in $O(nd) = O(E)$ time.

4 Augmenting Paths

We have shown that QuickMatch consistently returns large matchings, but it will not always return a perfect matching. Augmenting paths provide a way to convert a matching returned

by QuickMatch into a perfect matching. Given a graph G and a matching M, an *augmenting path* is a path from a vertex $l \in L_M$ to a vertex $r \in R_M$ that alternates between edges in $E \setminus M$ and edges in M.[2] Hopcroft and Karp show that, given an augmenting path P relative to a matching M, the matching $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$ [8].[3] Finding one augmenting path, therefore, allows us to increase the size of any matching by 1. Additionally, by Berge's Lemma, M is maximum if and only if there are no augmenting paths in G [1].

Hopcroft-Karp

The well known Hopcroft-Karp algorithm uses augmenting paths to compute a maximum matching, which is a perfect matching in the case of a regular bipartite graph.

HopcroftKarp(G)
1 $M \leftarrow \emptyset$
2 **while** there exists an augmenting path in G
3 **do** Let P be a maximal set of vertex-disjoint, shortest-length augmenting paths w.r.t. M
4 $M \leftarrow M \oplus P$

To implement one phase of the Hopcroft-Karp algorithm, we use a two-step process to find a maximal set of vertex-disjoint, shortest-length augmenting paths. Using a modified breadth-first-search (BFS) in $O(E)$ time, the algorithm constructs a directed graph G^* in which every path corresponds one-to-one with a shortest-length augmenting path in G. Then, the algorithm constructs a maximal set of vertex-disjoint, shortest-length augmenting paths by scanning paths in G^* using depth-first-search (DFS), inserting the discovered augmenting path into P, removing the examined edges, and repeating until there are no remaining paths in G^* that connect a vertex in L_M to a vertex in R_M. Because every time we examine an edge we remove it from G^*, this step also takes $O(E)$ time.

The main advantage of the Hopcroft-Karp algorithm is that in $O(E)$ time, we can find many augmenting paths. As long as we find more than two augmenting paths per phase on average, this method will be faster than using a basic BFS or DFS to find a single augmenting path at a time. When augmenting paths are short at the beginning of the Hopcroft-Karp algorithm, there will tend to be many augmenting paths discovered in a single phase. For instance, when the shortest augmenting paths are of length 1 at the start of the algorithm, we are guaranteed that at least $n/2$ edges will be inserted into the matching [2]. With each phase, the length of the augmenting paths found strictly increases [8]. Because the augmenting paths discovered in each phase are vertex disjoint, the upper bound on the number of augmenting paths that can be found in a single phase strictly decreases with each phase.

The Hopcroft-Karp algorithm is one potential solution for completing the matchings returned by QuickMatch. We would simply modify Hopcroft-Karp by initializing M to M_{QM} at the start of the algorithm. Because M_{QM} is nearly maximum, however, it is not necessarily true that Hopcroft-Karp will average more than two augmenting paths per phase.

[2] $A \setminus B$ denotes all elements of A that are not in B.
[3] $A \oplus B$ denotes the union of $A \setminus B$ and $B \setminus A$.

In fact, experiments showed that Hopcroft-Karp found an average of only 1.253 edges per phase for a graph when $n = 100{,}000$ and $d = 3$. Therefore, we compare the running time of Hopcroft-Karp to BFS and DFS. We use a modified BFS that is guaranteed to find the shortest-length augmenting path in the graph. The BFS and DFS methods are each used to find a single augmenting path in $O(E)$ time.

New Methods for Augmenting Paths

Here, we present three new methods of computing augmenting paths. Because we know the sets of vertices at which our augmenting paths will start and end (L_M and R_M respectively), we look for augmenting paths by performing two separate searches simultaneously. We do so by inserting one dummy vertex u connected to each vertex in L_M and one dummy vertex v connected to each vertex in R_M. We dovetail two searches, beginning one at u and the other at v. Each search expands its search tree such that every path in the tree is an alternating path.[4] The searches alternate expanding their search trees until one of the searches discovers a vertex that was already discovered by the other search. This vertex is called the *joining vertex*. At this point, an augmenting path can be created by concatenating the two alternating paths that contain the joining vertex and removing the two edges connected to the dummy vertices u and v.

We call this method a *breadcrumb search*. As each search expands its tree, imagine that it leaves breadcrumbs at the vertices it visits as it searches through the graph. When one tree finds the breadcrumbs of the other, we have found a joining vertex and can construct an augmenting path. The three variations of the breadcrumb search are *BFS-to-BFS*, *DFS-to-DFS*, and *BFS-to-DFS*. In the case of BFS-to-BFS, we alternate after expanding the tree by one level. In the case of DFS-to-DFS, we alternate after popping a single vertex from the stack and checking its neighbors for undiscovered vertices. In the case of BFS-to-DFS, we expand a single level from the BFS search and then expand the DFS search until we visit the same number of edges that the previous BFS level did.

Why dovetail two searches rather than just performing a single search? The breadcrumb search examines far fewer edges than a single search. If a BFS finds an augmenting path of length k, then the BFS-to-BFS will find one alternating path of length $\lceil k/2 \rceil$ and another alternating path of length $\lfloor k/2 \rfloor$. The number of vertices at depth k of a BFS search tree is at most d^k, and for small values of k the number of vertices at depth k is usually close to this upper bound. Hence, constructing two search trees of depths $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ will cost far less than constructing one search tree of depth k.

Figures 6 and 7 show the average costs of each method, measured in terms of edges examined and CPU time, when finding a single augmenting path. As expected, Hopcroft-Karp performed worse than BFS or DFS and averaged 128,670 edges examined. BFS averaged 87,950 edges examined, and DFS averaged 61,671 edges examined. BFS-to-BFS averaged 2,533 edges, DFS-to-BFS averaged 2,132 edges, and DFS-to-DFS averaged 2,111 edges. These results provide compelling empirical evidence that the breadcrumb searches are the clear winner when compared with BFS, DFS, and Hopcroft-Karp.

[4] An *alternating path* has the same properties as an augmenting path, except it does not necessarily connect two hermits.

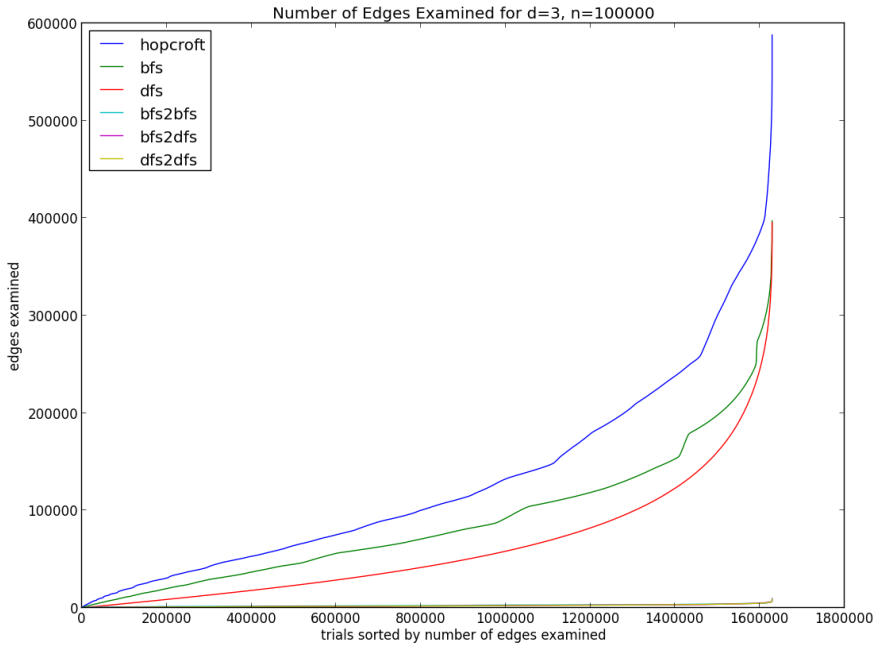

Figure 6: The distribution of edges examined by each method when finding a single augmenting path. To generate this data we generated a random graph, performed QuickMatch, and tried each of the six methods for turning M_{QM} into a perfect matching. We recorded the number of edges examined when finding a single augmenting path. Because Hopcroft-Karp can find multiple edges in a single phase, we divide the edge examination cost among all the paths found in one Hopcroft-Karp phase. All other methods find one augmenting path at a time, so that distributing edge examinations is not necessary. For each method, the trials are plotted in order of increasing edge-examination cost. BFS-to-DFS, BFS-to-DFS, and DFS-to-DFS all appear at the very bottom of the plot. This plot represents 1,600,000 trials for each augmenting-path method.

Figure 8 shows the average number of edges examined by a BFS-to-BFS search to find a single augmenting path. Additionally, the figure shows this plot normalized by $0.0023n^{0.6}$. Because this normalized plot decreases as n increases, we know that the observed rate at which the average number of examined edges to find an augmenting path is $O(n^{0.6})$ when $d = 3$. Combined with the average $O(n^{0.4})$ bound on the number of missed matches, these bounds imply that QuickMatch combined with BFS-to-BFS will take $O(n)$ time to complete on average when $d = 3$.

Conclusion

We have shown that a near-perfect matching can be quickly constructed using QuickMatch, a simple heuristic algorithm that greedily constructs a matching while preventing hermits. In combination with breadcrumb searches for the case where $d = 3$, we have shown that we can find a perfect matching in $O(n)$ time on average. Our implementation of QuickMatch

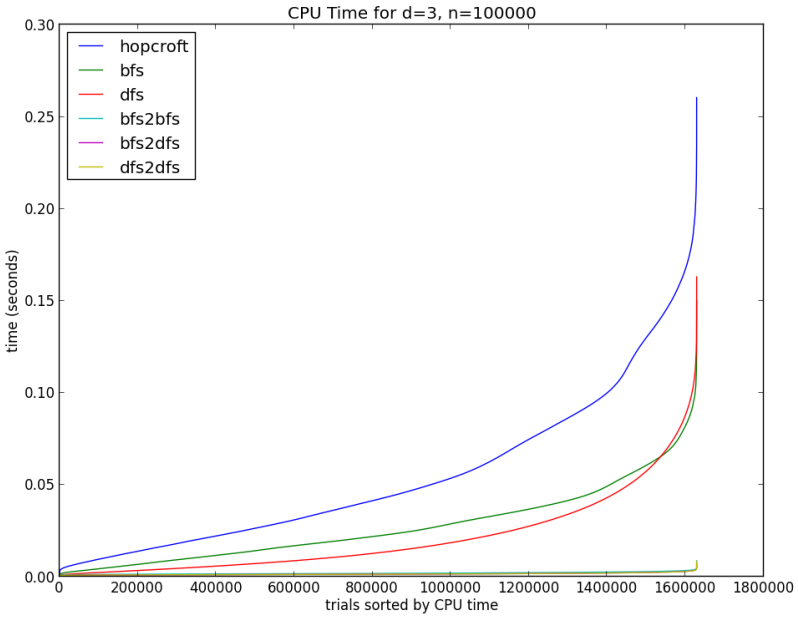

Figure 7: The distribution of CPU time spent by each method when finding a single augmenting path. This data was generated using the same methods as Figure 6.

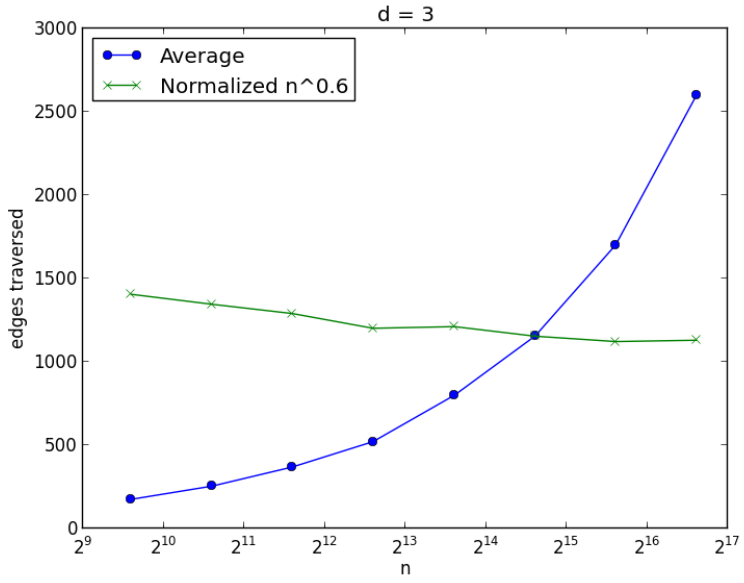

Figure 8: The average number of examined edges to find an augmenting path via BFS-to-BFS in a graph where $d = 3$ and n is increasing. We also show a plot of this data that has been normalized by $0.0023n^{0.6}$. Because this normalized plot has a negative slope as n increases, we know that the observed rate at which the average number of examined edges to find an augmenting path increases is $O(n^{0.6})$ when $d = 3$.

combined with DFS-to-DFS finds a perfect matching in 1.5 seconds when $n = 1{,}000{,}000$ and $d = 3$. Using Euler splits, we find edge colorings in 6.4 seconds when $n = 50{,}000$ and $d = 50$. These tests were run on a MacBook Pro with a 2.4 GHz Intel Core i7 processor and 4GB of RAM.

The initial research for this paper was motivated by a practical application of the edge-coloring problem. Our research group was developing an algorithm for out-of-core radix sort on the parallel disk model. In order to efficiently remap blocks of data between disks, we had to compute an edge coloring of a regular, bipartite multigraph. Other applications of edge-colorings on bipartite multigraphs include scheduling problems such as the the class-teacher timetable problem [7]. Cole, Ost, and Schirra also cite routing permutation networks [11], the k-k routing problem [15], the file transfer problem [14], and simulation of a PRAM on a distributed memory machine [9] as other applications of edge colorings in bipartite graphs.

Some of the work we pursued while researching this problem has not been presented in this paper because it is incomplete. It is currently unknown whether we can prove a theoretical $o(n)$ upper bound on the number of hermits resulting from QuickMatch. We were able to show an upper bound of $nd/(2d-1)$ on the size of a maximal matching in a regular bipartite graph multigraph, but this bound is linear with n. We have also begun exploring a different method of computing edge colorings in regular, bipartite multigraphs that uses a perfect matching consisting of dummy edges to convert odd-degree graphs into even-degree graphs. The method then uses a cycle-finding technique to find an Euler orientation of the graph that directs all dummy edges in the same direction. Early stages of research on this method appear promising.

Acknowledgments

Finally, I would like to thank the research group I have been working with this year, which consists of Professor Thomas Cormen, Lauren Tran, Elaine Levey, Shuyang Fang, and Steffi Ostrowski. Lauren Tran's ideas motivated the first iterations of QuickMatch. Together, their ideas, intuitions, and editorial comments have guided the development of the presented methods and have been invaluable while drafting this paper. Thank you and best of luck to you all!

References

[1] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842–844, 1957.

[2] Béla Bollobás. *Modern Graph Theory*. Springer, 1998.

[3] Richard Cole. Private communication, January 2013.

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

[5] Alan Frieze. Perfect matchings in random bipartite graphs with minimal degree at least 2. *Random Structures and Algorithms*, 26(3):319–358, 2005.

[6] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. Perfect matchings in $O(N \log N)$ time in regular bipartite graphs. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, STOC '10, pages 39–46, 2010.

[7] C. Gotlieb. The construction of class-teachers time-tables. In *Proceedings of the International Federation for Information Processing Congress 62*, pages 73–77, 1963.

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

[9] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *In Proceedings Of the Twenty-Fourth ACM Symposium on Theory of Computing*, pages 318–326, 1992.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*, volume 2. Addison-Wesley Longman Publishing Co., Inc., 1997.

[11] G. Lev, N. Pippenger, and L.G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computer Publications*, pages 93–110, 1981.

[12] Kazuhisa Makino, Takashi Takabatake, and Satoru Fujishige. A simple matching algorithm for regular bipartite graphs. *Information Processing Letters*, 84(4):189–193, November 2002.

[13] Brendan D. McKay and Nicholas C. Wormald. Uniform generation of random regular graphs of moderate degree. *Journal of Algorithms*, 11(1):52–67, February 1990.

[14] Shin-ichi Nakano, Xiao Zhou, and Takao Nishizeki. Edge-coloring algorithms. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1995.

[15] Job Sibeyn. Edge coloring bipartite graphs efficiently. *Proceedings of Computing Science in the Netherlands*, pages 269–279, 1992.