Dartmouth College Computer Science Technical Report TR2015-771

Two Algorithms for Finding Edge Colorings in Regular Bipartite Multigraphs

Patricia Neckowicz

May 20, 2015

Abstract

For a d-regular bipartite multigraph, an edge coloring is equivalent to a decomposition of the edge set into d perfect matchings. When d is a power of 2, we can recursively perform Euler partitions to find the perfect matchings. When d is not a power of 2, however, we eventually reach a subproblem graph of odd degree where we can no longer perform an Euler partition. We propose two different algorithms that address this case. Both algorithms make use of an auxiliary matching, called *dummy edges*, to make the degree of the graph even. In the first algorithm, dummy edges are added prior to tracing out cycles. In the second algorithm, we add dummy edges as a way to close paths that have already been traced. We will analyze both algorithms separately and also consider a hybrid version.

1 Introduction

This paper presents two algorithms for computing edge colorings on d-regular bipartite multigraphs. Our research here was motivated by the problem of performing out-of-core radix sort on the parallel disk model, one application of bipartite edge colorings. Let $G = (L, R, E)$ be a d-regular bipartite multigraph with $N = |L| = |R|$. Let the vertices in L, called left vertices, be labeled from 0 to $N-1$, and let the vertices in R, called right vertices, be labeled from N to $2N-1$. An edge coloring of G is equivalent to a decomposition of E into d perfect matchings. One algorithm for identifying these d perfect matchings uses a divide-and-conquer approach. When d is even, we split the edge set E into two sets E_1 and E_2 such that each of the multigraphs $G_1 = (L, R, E_1)$ and $G_2 = (L, R, E_2)$ is $d/2$-regular. We form this partition, called an *Euler partition*, by tracing out disjoint cycles that include each edge in E exactly once and separating those edges taken left to right from those taken right to left. The number of times we enter a vertex equals the number of times we exit a vertex, and so each vertex has exactly $d/2$ incident edges taken left to right and $d/2$ incident edges taken right to left. Thus, the resulting multigraphs will be $d/2$-regular. Because the resulting multigraphs are now smaller instances of our original problem, we can recurse; our base

case is $d = 1$, where the remaining edges form a perfect matching. If the degree of the multigraph is a power of 2, we can perform $\lg d$ Euler partitions to find all d of these matchings. When d is not a power of 2, however, eventually a partition will result in a multigraph of odd degree, and we are no longer guaranteed that we can decompose the graph into cycles. The two algorithms we propose in this paper specifically address the case in which a subproblem graph has odd degree.

2 Background

Previous attempts to solve the edge-coloring problem share a common procedure. When the degree of a graph is odd, they identify and remove one perfect matching from the edge set; the resulting even-degree graph can then be split via regular Euler decomposition. In 2001, Cole, Ost, and Shirra [COS01] proposed an algorithm for identifying a perfect matching. The proposed algorithm achieves an optimal running time of $O(E)$. As a result, we can write the recurrence for the entire divide-and-conquer approach as $T(N,d) = 2T(N,d/2) + O(E)$, which implies an optimal running time of $O(E \lg d)$ for computing an edge coloring. Unfortunately, there is no known implementation. Another attempt at identifying a perfect matching is Quickmatch, a heuristic algorithm proposed by Andrew Hannigan in 2013 [Han13] that finds a near-maximum matching and converts this matching to a perfect matching by finding alternating paths. Although experimental results suggest that Quickmatch runs in $O(E)$ time, no analysis proved this bound. Chainmatch, proposed by Stefanie Ostrowski in 2014 [Ost14], decomposes a bipartite multigraph into even-length vertex-edge chains, which can be converted to a perfect matching by picking out the edges that pair every two vertices. Like the algorithm proposed by Cole et al., Chainmatch has yet to be implemented.

We see that all attempts to solve the edge-coloring problem have been overly complicated or have failed to prove the running-time bound. With these shortcomings in mind, let us consider a new approach.

3 Approach

Instead of removing a perfect matching to achieve even degree, we can add a perfect matching. Let us call the added edges *dummy edges*, and assume for now that all dummy edges are included in the same subproblem after an Euler partition is formed; we will see in a moment why this property is important. Each of the two subproblem graphs will fall into one of four cases after the partition:

1. an even-degree graph with no dummy edges,
2. an odd-degree graph with dummy edges,
3. an odd-degree graph with no dummy edges, or
4. an even-degree graph with dummy edges.

In case 1, we can perform Euler partitions with no constraint. In case 2, we can remove the dummy edges to get case 1. In case 3, we are left with our original problem, and so we add dummy edges to achieve even degree, arriving at case 4. We see that in case 2, we can remove the dummy edges

because they were included in the same subproblem of the Euler partition; if we had not required this property, the subproblems with only some dummy edges would be meaningless, as they could not be eventually decomposed into the original perfect matchings. Therefore, we will require that dummy edges be placed in the same subproblem and enforce this property by traversing all of the dummy edges in the same direction during the Euler partition. If our algorithm can satisfy this constraint and perform the partition in $O(E)$ time, we will achieve the optimal running time.

We will address this problem in two ways. In our first algorithm, we will add dummy edges prior to tracing out cycles and require that they be taken left to right at the start of each cycle. In the second algorithm, we will use dummy edges to close paths that have been traced from the left side of the graph to the right side; these dummy edges will all be taken from right to left.

4 Static dummy edges

In our first algorithm, we add dummy edges prior to tracing out cycles in the Euler decomposition. Let $G = (L, R, E)$ be a d-regular bipartite multigraph with d odd, and let us arbitrarily choose the dummy edges to be the set of edges that pair each left vertex x with its counterpart $x+N$ on the right. In general, we say that a left vertex x and a right vertex y are *counterparts* if they are connected by a dummy edge. Now that G is $(d+1)$-regular (it has even degree), we can perform an Euler partition with the additional requirement that we traverse all dummy edges in the same direction. Without loss of generality, we require that the dummy edges be taken from left to right.

We call a left vertex *available* if its incident dummy edge has not been put on a cycle. We start each cycle at an available vertex, which we call our *home* vertex. Note that committing an edge to a cycle is equivalent to assigning that edge a direction, either left to right or right to left. From this point forward, we will refer to an edge taken from left to right as a $+1$ edge and to an edge taken from right to left as a -1 edge. Edges with no direction will be 0 edges. Ultimately, each dummy edge should be a $+1$ edge. In this algorithm, it is possible for an edge assignment to be undone; for example, if we traverse a $+1$ edge from right to left, we will "add" a -1 to a $+1$, resulting in a 0 edge.

With dummy edges added, we repeatedly trace out cycles until all dummy edges have been put on a cycle. Starting at an available vertex $x \in L$ (our home vertex), take the dummy edge from x to its counterpart $x+N$; this edge is now a $+1$ edge, and so our direction constraint has been satisfied for this dummy. Search for a path from $x+N$ back to x, keeping in mind the following rules:

1. We cannot take a $+1$ edge from left to right, and we cannot take a -1 edge from right to left. Thus, all edge assignments will remain in the range -1, 0, $+1$.

2. We cannot take any dummy edge from right to left. Thus, we cannot change a $+1$ dummy edge to a 0 dummy edge, and we cannot have a -1 dummy edge. In other words, each dummy edge must be taken exactly once, from left to right.

Once we find a path back to the home vertex, we commit these edges to a cycle by assigning each edge either a $+1$ or -1 depending on the direction it was taken. If an edge in the path already has a $+1$ or -1, we must "add" the new direction to the old one as demonstrated above; rule 1 guarantees that adding these directions will always result in a 0 edge. We then begin the next cycle at another available vertex. Figure 1 shows some possible edge assignments after tracing two cycles. By

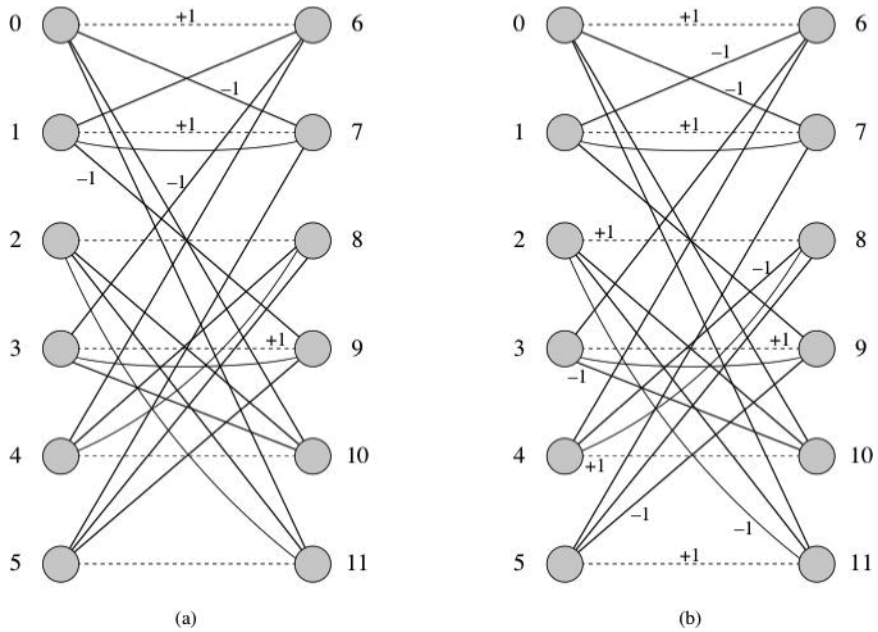

Figure 1: A 4-regular bipartite multigraph with $N = 6$. Dashed lines indicate dummy edges. **(a)** Edge assignments after tracing the cycle 0-6-3-9-1-7-0. **(b)** Edge assignments after tracing another cycle 2-8-4-10-3-6-1-9-5-11-2. Edges $(3,6)$ and $(1,9)$ go from -1 to 0.

following our two rules, we guarantee that when the algorithm terminates, all non-dummy edges will either be 0 edges, $+1$ edges, or -1 edges and that all dummy edges will be $+1$ edges. We shall see shortly that we can easily assign directions to the remaining 0 edges.

Of course, this algorithm will not successfully perform the Euler partition if there is no path that satisfies our rules back to the home vertex. The following lemma guarantees that we will always be able to find such a path. For this lemma, is useful for us to introduce the concept of vertex orientation. The *orientation* of a vertex x is the sum of all edge assignments incident on x. For example, if a vertex has one incident 0 edge, two incident $+1$ edges, and one incident -1 edge, its orientation is $+1$.

Lemma 1 *When we take a dummy edge from left to right at the start of tracing out a cycle, there is always a path to the home vertex that follows the two rules.*

Proof: Let x be an available vertex, and let G' be the connected component of G that contains x and its counterpart. Note that G' is d-regular with d even and $d \geq 4$; we add dummy edges only to make d even, and if $d = 2$, we would remove the dummy edges to get a perfect matching. To prove the lemma, we will first show that if the orientation of each vertex is 0 when we start tracing a cycle from x, we will be able to close the cycle with a path that follows our rules. We will then show by induction that every time we start a cycle, the orientation of each vertex in the graph is 0. Taken together, these two properties prove the lemma.

Assume that each vertex in G' has orientation 0 when we start tracing a cycle from x. To prove that a path that follows our rules exists from x's counterpart to x, we can show that if we enter a vertex via one edge, we can always leave via a different edge. Consider a left vertex v with orientation 0. If we enter v via an edge e, we know that e must be a $+1$ edge or a 0 edge.

Therefore, v must have another incident edge that is either a -1 or 0 edge, because there are at least three remaining incident edges and they cannot all be $+1$ edges (or the orientation would not be 0). Similarly, if v is a right vertex that we enter via e, then e must be either a 0 edge (potentially a dummy edge) or a -1 edge. Because $d \geq 4$, we must have at least two incident $+1$ or 0 edges by similar reasoning, so that even if one of these edges is a dummy edge, we are not forced to take the dummy from right to left. Therefore, our search will always find a path back to x.

When we start a cycle, either this cycle is the first cycle traced on G' or we just closed another cycle. In the first case, we know that the orientation of each vertex is 0, because all edges start as 0 edges. For the second case, let's assume that the orientation of each vertex is 0 prior to tracing the previous cycle. By closing the previous cycle, we enter a given vertex the same number of times that we exit, and so we add the same number of $+1$ edges to its orientation as we do -1 edges. Thus, when we start the next cycle, each vertex has orientation 0. ∎

Corollary 2 *The algorithm will terminate after tracing at most N cycles.*

Proof: After we take a dummy edge, it is never untaken, and we always find a path back to the home vertex. Therefore, every time we trace a cycle, we remove at least one left vertex from the set of available vertices, and so we terminate after tracing at most N cycles. At this point, the number of 0 edges incident on each vertex is even, and so we can easily trace the remaining cycles from the set of 0 edges. ∎

Search options

Although we did not specify the type of search to use when searching for a path back to a home vertex, preliminary results suggest that depth-first search (DFS) is the most efficient search option. Compared with breadth-first search (BFS) and some hybrid options, DFS examines significantly fewer edges.

Additional heuristics

Various heuristics can improve the running time of the algorithm. We present the most successful heuristics below, all of which prioritize certain edges over others during a DFS.

- If at a left vertex:
  - If the vertex is available, take the dummy edge.
  - Otherwise, take a -1 edge if possible.
- If at a right vertex:
  - If the vertex is adjacent to the home vertex, close the cycle by taking the edge to the home vertex.
  - Otherwise, if the vertex is adjacent to an available vertex, take the edge to the available vertex.
  - Otherwise, take a $+1$ edge if possible.

In other words, finish a cycle as soon as possible, always take an untaken dummy edge, and bias edge assignments towards 0.

5 Dynamic dummy edges

In our second algorithm, we add dummy edges dynamically as a way to close paths that have already been traced from a left vertex to a right vertex. For example, if we start a path at a left vertex x and end at a right vertex y, we add a dummy edge (x,y), which we then take from right to left to close the cycle. Recall that these dummy edges must form a perfect matching. As in the previous algorithm, we call a left vertex x our *home* vertex if start a path at x. A vertex is *dummied* if it has an incident dummy edge; otherwise, it is *undummied*.

Instead of performing a search to close a cycle, we will *walk*. A walk differs from a search in that we commit edges to cycles as we traverse them, not waiting until we have completed a path, and we traverse only edges that have not been traversed before. A walk will thus result in a *trail* of disjoint edges, which differs from a path in that vertices can be repeated. We also introduce the concept of *directed* edges, which differ from $+1$, -1, and 0 edge assignments. When we direct an edge, either left to right or right to left, we are requiring that, *if* this edge is traversed, it must be traversed in this direction. Once an edge has been traversed, we assign either a $+1$ or -1 as in the previous algorithm. Because we do not traverse edges more than once, we do not undo edge assignments.

To trace one cycle, start a walk at an undummied left vertex. At a given vertex, we can choose any incident edge to traverse as long as it has not already been traversed as part of this or any other cycle and is not directed in the opposite direction. We keep in mind the following rule: after leaving a vertex x, if there is one incident edge remaining on x, we direct this edge from left to right, regardless of whether x is a left or right vertex.

Because the degree of the graph is odd, eventually we will get stuck. That is, we will not be able to leave the vertex that we enter because all incident edges have been taken or are directed in the opposite direction. We know that this vertex must be a right vertex, however, for if we were to get stuck at the left vertex, we must have gotten there by taking its last untraversed, incident edge, but we must have already directed it from left to right. Therefore, if we start a cycle at a left vertex x, we will get stuck at a right vertex y, and so we can add a dummy edge (x,y) to the graph, which we then take from right to left to close the cycle. We can then start the next cycle at new, undummied left vertex. Figure 2 shows edge assignments and directions after tracing two paths from left to right.

A problem arises when we get stuck at a dummied right vertex. We cannot add a dummy edge because the dummy edges must form a perfect matching. In this case, we discard the trail that we traced from our home vertex x and start at a new undummied vertex, putting x to the side. We also declare the edges in the discarded trail as having not been traversed. Eventually, we no longer have a new left vertex on which to start a path, and so we need another way to add dummy edges to the vertices that we set aside. At this point, we start a search from an undummied left vertex, looking for any undummied right vertex. We now allow edge assignments to be undone, keeping in mind the same rules as in the previous algorithm: we must keep edge assignments within the range -1, 0, $+1$, and we cannot take a dummy edge if it has already been taken. All directed edges are treated as 0 edges during a search. When we find an undummied right vertex during a search, we add a dummy edge, take the dummy edge from right to left, and commit the traversed edges to a cycle. We continue to perform searches until all dummy edges are added. We can apply the same logic from the previous lemma to prove that a search will always be successful; that is, within a connected component, there is always a path from an undummied left vertex to any undummied

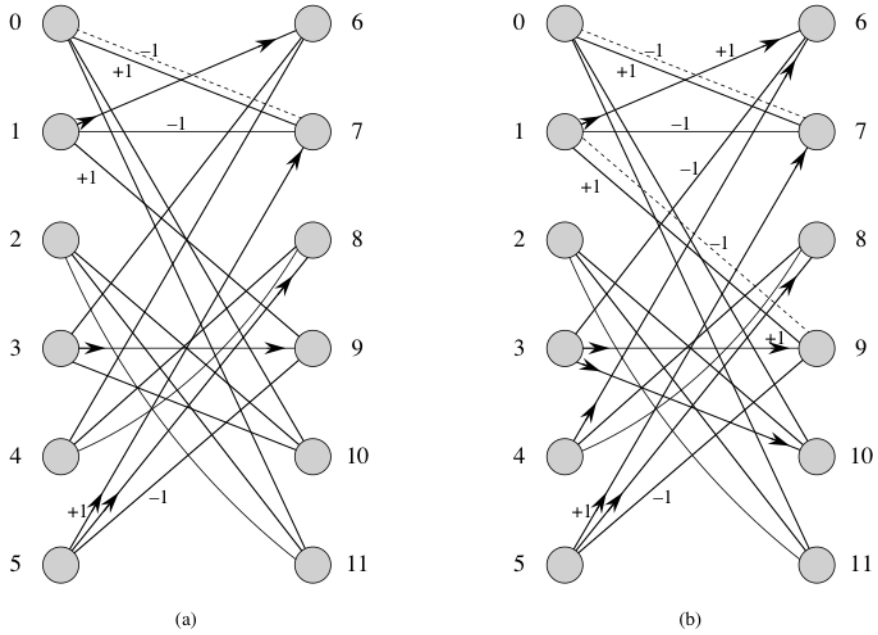

Figure 2: A 3-regular bipartite multigraph with $N = 6$. Arrows indicate directed edges, and dashed lines indicate dummy edges **(a)** Edge assignments and directions after tracing the path 0-7-1-9-5-7. The dummy edge is $(0,7)$. **(b)** Edge assignments and directions after tracing the path 1-6-3-9. The dummy edge is $(1,9)$.

right vertex.

Directing more edges

It turns out that during a walk, we can do better than directing just the last incident edge on a vertex. Let's consider a hypothetical example to illustrate our motivation behind a new edge-directing strategy. Imagine a d-regular bipartite graph $G = (L,R,E)$ where d is odd and each vertex v in L or R somehow has exactly $\lfloor d/2 \rfloor$ incident edges directed from right to left and $\lceil d/2 \rceil$ incident edges directed from left to right. (Our new strategy will not actually predefine edge directions.) If we then add a perfect matching of dummy edges to E, with each dummy edge directed from right to left, it turns out that we have already solved the problem of dividing E into two subproblem graphs. Each edge will be traversed either left to right or right to left as part of a cycle, and at a given vertex, the number of edges directed from left to right equals the number directed from right to left. Thus, the edges directed from right to left will end up in one subproblem and the edges directed from left to right will end up in the other.

Of course, our algorithm does not predefine edge directions, but we can use this example to motivate an improved strategy for directing edges. We say that an edge is *committed* if it has either been traversed or directed; otherwise, an edge is *uncommitted*. Our discussion above can be summarized with the following observations:

- An undummied vertex should have $\lfloor d/2 \rfloor$ incident edges committed right to left and $\lceil d/2 \rceil$ incident edges committed left to right.
- A dummied vertex should have $d/2$ incident edges committed right to left and $d/2$ incident edges committed left to right.

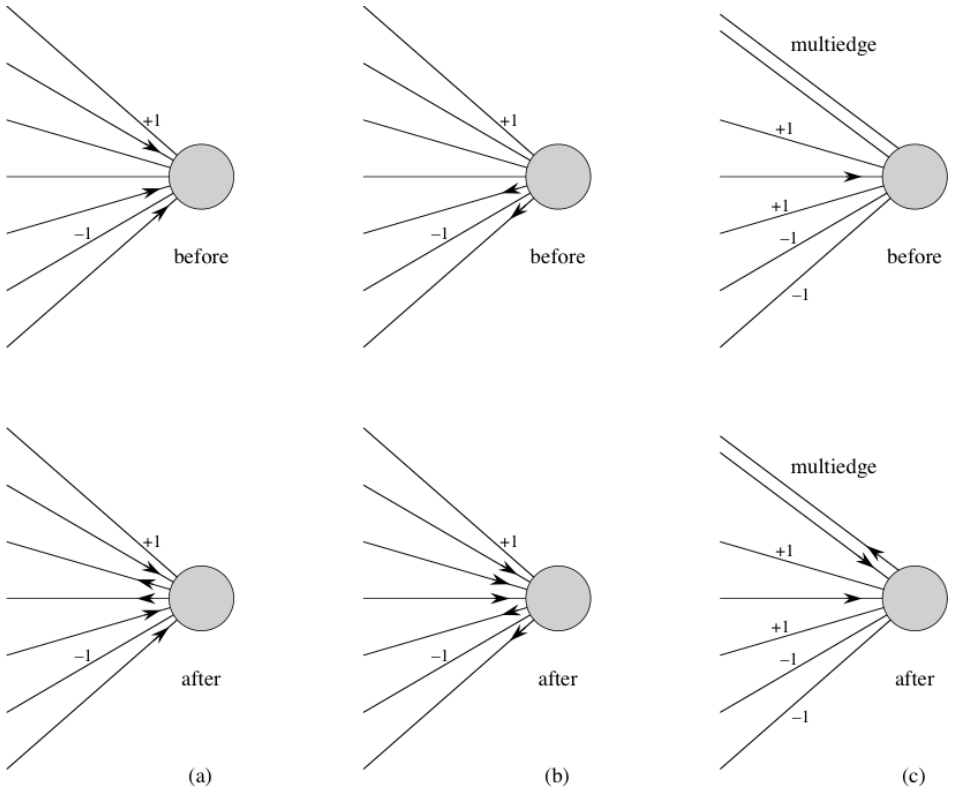

Figure 3: Cases where we would direct edges on a d-regular bipartite multigraph for $d = 7$. For each case, we direct edges at the current vertex either because we just took an edge out of this vertex or because we arrived at this vertex via direction propagation. **(a)** We have four ($\lceil d/2 \rceil$) edges committed from left to right, and so we direct the two uncommitted edges from right to left. **(b)** We have three ($\lfloor d/2 \rfloor$) edges committed from right to left, and so we direct the three uncommitted edges from left to right. **(c)** We have three edges committed from left to right and two edges committed from right to left. Normally, this information is not enough to direct the two uncommitted edges. Because the uncommitted edges are a multiedge, however, we know that one instance must be directed from right to left and the other instance must be directed from left to right.

With these observations in mind, we can change our rule so that instead of directing only the last incident edge, every time we leave a vertex during a walk, we check to see whether there is only one way to direct the remaining uncommitted incident edges in order to satisfy the desirable conditions above. Figure 3 shows some situations where our strategy will direct edges. It is possible that we will not satisfy these conditions for all vertices. If we direct any edges at vertex x, we then propagate the direction forward until we are no longer forced to direct any edges. In other words, for every edge (x, y) that we direct, we direct edges from vertex y, keeping in mind the same goals. Then, for every edge (y, z) that we direct, direct edges from vertex z, and so on. We stop when all edges at a given vertex are already committed or we are not forced to direct any edges to achieve our desired conditions. As previously mentioned, it is possible that some vertices will not have the desired number of incident edges committed left to right and right to left. The search step of the algorithm addresses this problem in that it treats directed edges as 0 edges, and so we no longer have to obey the edge directions. Once again, this strategy does not guarantee that our conditions

will be satisfied for each vertex; it is just a heuristic that in general performs better than our original edge-directing strategy.

Figure 4 illustrates an example of direction propagation. In part (a), we start at vertex 0 and proceed to form the path 0-7-1. When leaving vertex 7, we direct the edge $(5,7)$ from left to right in order to satisfy the conditions for vertex 7. We go to direct edges from vertex 5 and see that we are not forced to direct any edges, as the two remaining edges could each be taken in either direction. We go from vertex 1 to vertex 9 in part (b), directing edge $(1,6)$ from left to right, and then go from vertex 9 to vertex 5 in part (c), directing edge $(3,9)$ from left to right. When we take the edge $(5,7)$ in part (d), we get stuck and direct the edge $(5,8)$ from left to right. We are not done, however. At vertex 8, we have a multiedge $(4,8)$, and so we know that one instance of the edge must be taken from left to right and one from right to left. Thus, in part (e), we direct each instance of edge $(4,8)$ as such and go to direct edges at vertex 4. At vertex 4, we direct edge $(4,6)$ from left to right, as shown in part (f), and arrive at vertex 6, where we direct edge $(3,6)$ from right to left in part (g). At vertex 3, we direct edge $(3,10)$ from left to right, shown in part (h), and we are done, because we are not forced to direct edges at vertex 10. We see that the path 1-6-3-9 depicted in part (i) satisfies these additional directed edges.

Divide-and-conquer

Compared with the previous strategy, the dynamic dummy edge strategy is slightly trickier when performing the entire divide-and-conquer algorithm. Consider a even-degree subproblem graph with inherited dummy edges. This case is easy to address using the static strategy; we skip the step where we add the dummy edges and perform the rest of the algorithm normally, by repeatedly taking the inherited dummy edges from left to right to close cycles. In other words, the static strategy naturally handles this case because it relies on dummy edges being placed prior to tracing cycles. On the other hand, the dynamic strategy adds dummy edges *after* tracing cycles, and so the algorithm does not easily handle the case where dummy edges are included in the graph from the start. The algorithm requires that we start tracing cycles on a graph with odd degree and no dummy edges. Thus, in the case where we have an even-degree subproblem graph with dummy edges, we simply remove the inherited dummy edges from the graph and proceed to add new dummy edges dynamically.

6 Results

We now have two strategies to address the case where a subproblem graph has odd degree. In the first strategy, we add dummy edges to achieve even degree and then trace cycles while traversing the dummy edges from left to right. In the second strategy, we trace paths from left to right, adding dummy edges (taken from right to left) to close out each path into a cycle. We now want to compare the strategies on graphs with odd degree to see which is more efficient.

To compare the strategies, we randomly generate regular bipartite multigraphs with varying values of N and odd degree d. We perform both strategies on the same graph and count the number of times that each strategy examines an edge. By counting the number of times an edge is examined, we are measuring the dominant cost, and so we get a good sense of efficiency without measuring the running time.

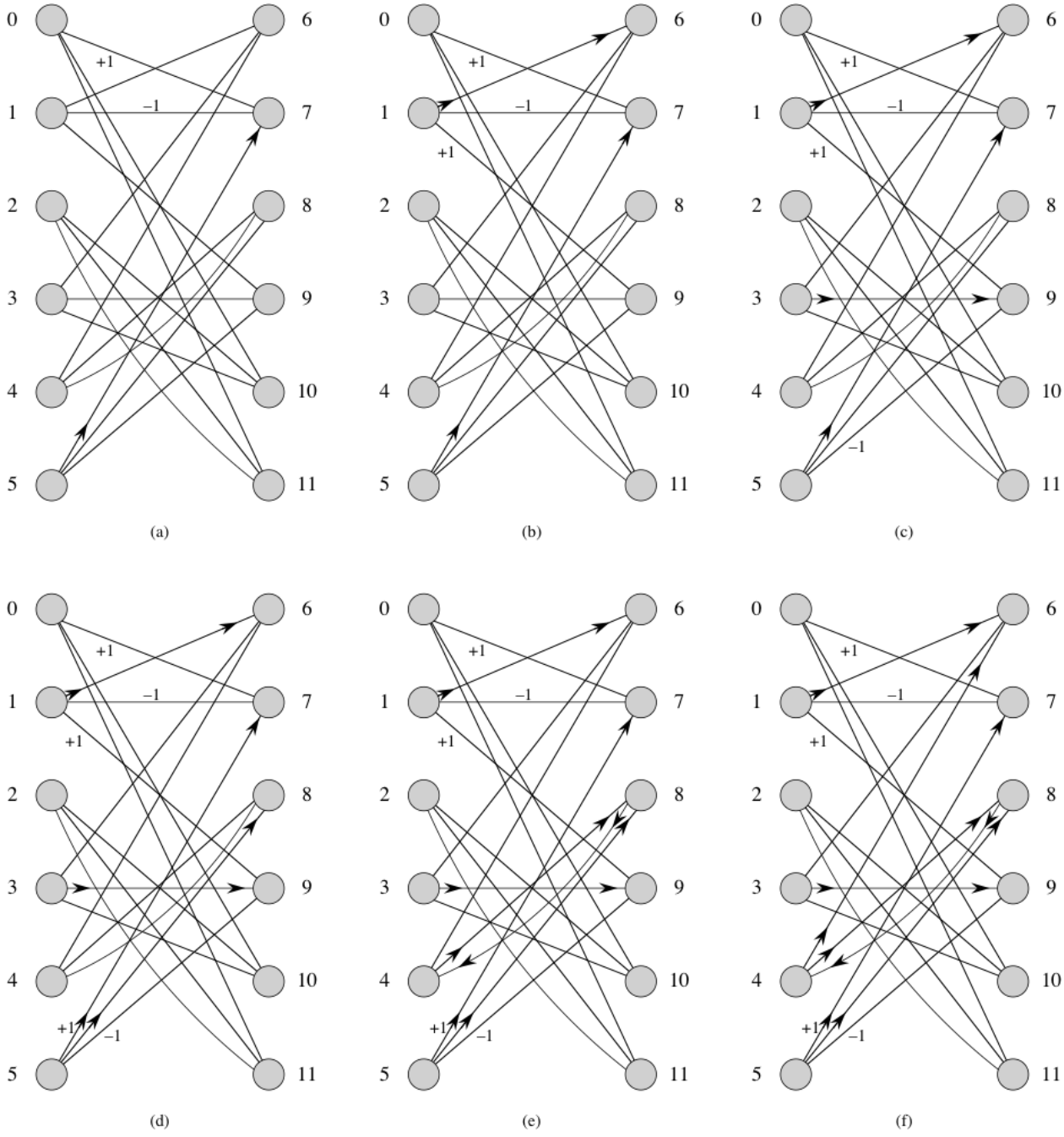

Figure 4: The same graph as Figure 2 with additional edge directions. **(a)** Starting at vertex 0, we go to vertex 7 and then vertex 1, directing edge $(5,7)$ from left to right. **(b)** Take the edge from vertex 1 to vertex 9 and direct edge $(1,6)$ from left to right. **(c)** Take the edge from vertex 9 to vertex 5, directing edge $(3,9)$ from left to right. **(d)** Take the edge from vertex 5 to vertex 7 and get stuck. We direct edge $(5,8)$ from left to right. **(e)** At vertex 8, we have a multiedge $(4,8)$, and so we direct one instance from left to right and the other from right to left. **(f)** At vertex 4, we direct edge $(4,6)$ from left to right.

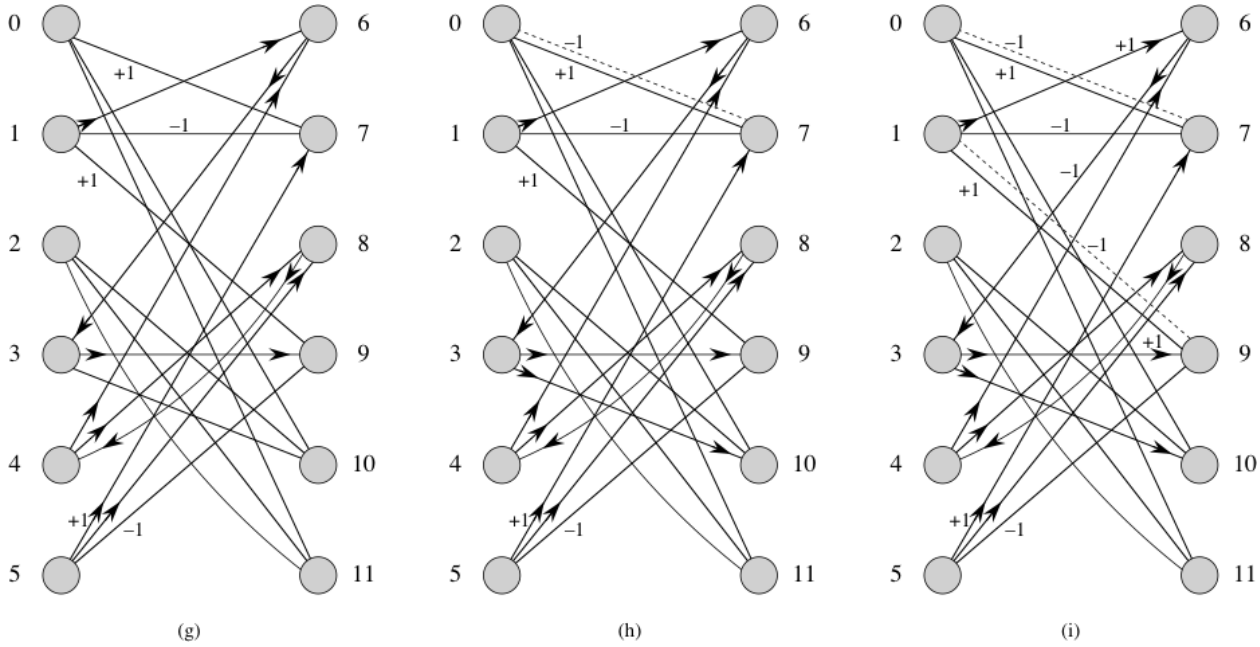

Figure 4, continued: **(g)** At vertex 6, we direct edge $(3,6)$ from right to left. **(h)** At vertex 3, we direct edge $(3,10)$ from left to right, and we are done directing edges. We then add the dummy edge $(0,7)$ to close the cycle. **(i)** We add the next dummy edge $(1,9)$ by tracing the path 1-6-3-9, which satisfies all added edge directions.

Figure 5 shows results for a fixed N ($N = 5000$) and varying odd d. We ran both strategies on 20 different graphs with the same value of d. Each dot represents one run on one graph; thus, there are two dots per graph (one for each strategy). The vertical axis shows the average number of times an edge is examined. The dynamic strategy examines fewer edges for low values of d, but for higher values of d, the static strategy examines fewer edges than the dynamic strategy. To explain this observation, let's think about how each strategy operates. The static strategy looks for a path from right to left that obeys a set of rules; as the degree increases, it should get much easier to find a path because at each vertex, there are more options for edges. On the other hand, the dynamic strategy relies on getting stuck, which means we want to traverse edges in the edge set. Thus, for higher-degree graphs, we still must look at roughly the same proportion of edges in the edge set.

As d increases, there must be a point at which it becomes beneficial to switch from the dynamic strategy to the static strategy. Based on Figure 5, we can guess that this point occurs around $d = 3$ and $d = 5$. To get a closer look at these problem sizes, we fix both $d = 3$ and $d = 5$ and vary N. Figures 6 and 7 show these results. For $d = 3$, the dynamic strategy usually examines fewer edges; there are, however, some outliers. For $d = 5$, the dynamic strategy still performs better in most cases, but it is no longer the obvious better choice.

With these results in mind, we can conclude that for $d = 3$, the dynamic strategy outperforms the static strategy. For $d = 5$, it is unclear which strategy is better, and so we choose the static strategy because it is easier recursively. For $d \geq 7$, we prefer the static strategy. With these results in mind, we can imagine a hybrid approach. When performing the entire divide-and-conquer algorithm, we perform the static strategy on graphs of degree 5 or higher and the dynamic strategy

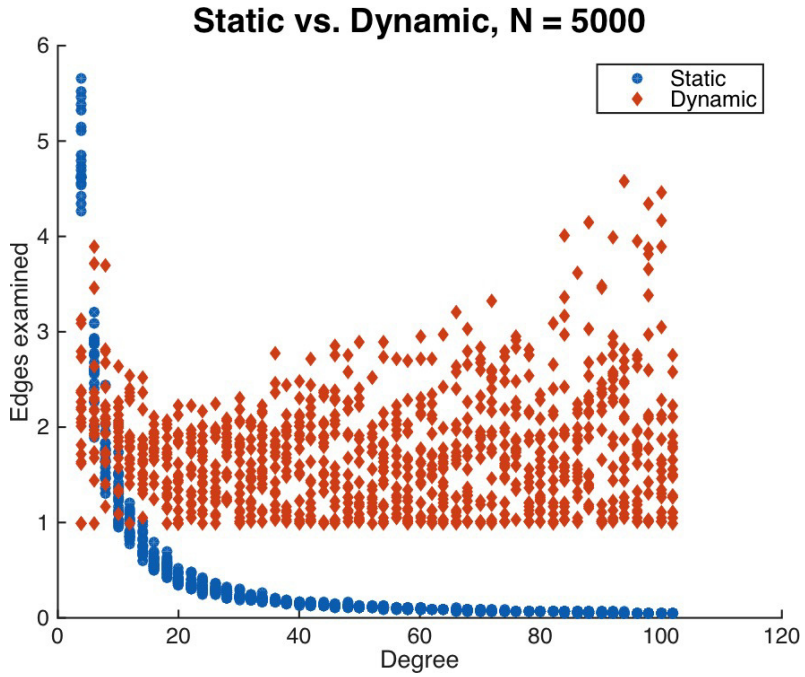

Figure 5: As the degree increases, the static strategy improves. The dynamic strategy examines roughly the same number of edges.

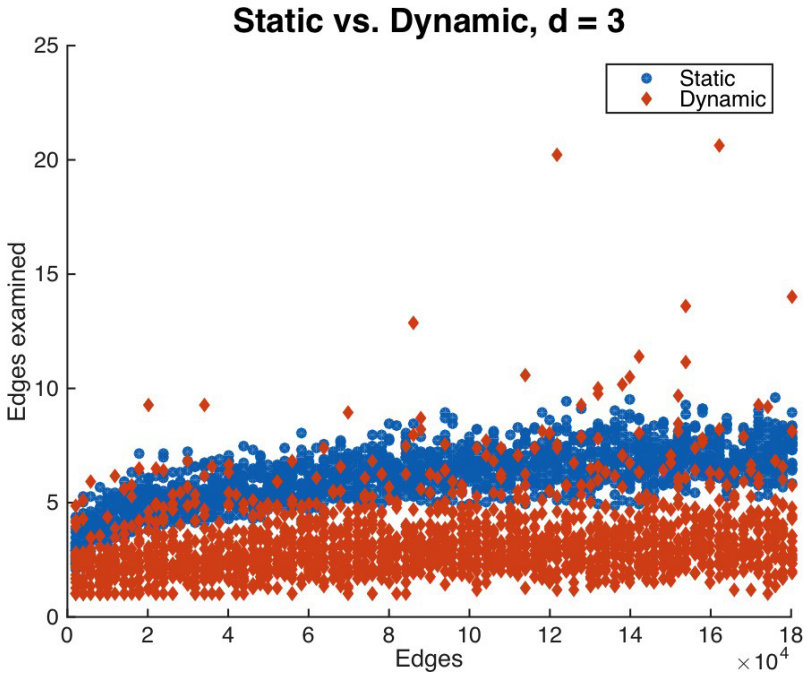

Figure 6: On degree-3 graphs, the dynamic strategy usually examines fewer edges.

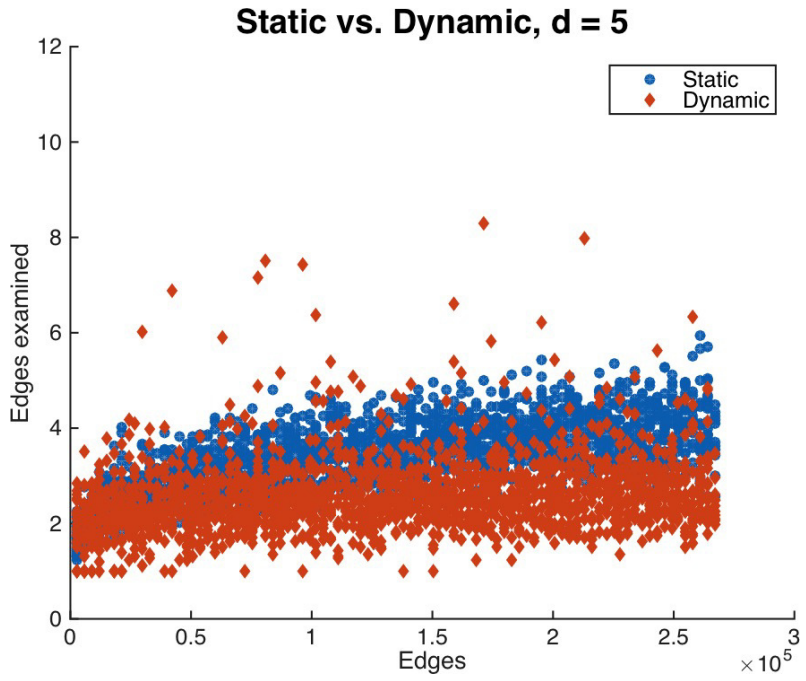

Figure 7: On degree-5 graphs, it is not clear which strategy is better.

on graphs with degree 3. If we reach a subproblem of degree 4 with dummy edges, we perform the dynamic strategy by removing the dummy edges as discussed above.

7 Conclusions

In this paper, we proposed two algorithms for finding edge colorings in regular bipartite multigraphs. These algorithms significantly differ from previous attempts in that they both add a perfect matching instead of removing one to achieve even degree. The static algorithm first adds a perfect matching of dummy edges and then requires these dummy edges to be traversed from left to right. The dynamic strategy repeatedly traces paths from a left vertex to a right vertex and adds dummy edges between the endpoints of these paths; these dummy edges are taken from right to left. We have implemented various heuristics to increase the efficiency of our strategies.

Results showed that the dynamic strategy is the better choice on 3-regular graphs, while the static strategy performs best on graphs with odd degree greater than 3. Thus, we suggest combining both strategies into a hybrid approach, where we perform the entire divide-and-conquer edge coloring by using the dynamic strategy on degree-3 subproblem graphs and the static strategy on subproblem graphs of higher, odd degree.

Looking forward, there is still work to be done. Although both proposed algorithms are guaranteed to terminate successfully, we do not yet have proofs that either strategy achieves the optimal running time of $O(E)$ on a single subproblem graph or that the entire divide-and-conquer hybrid approach can be performed in $O(E \lg d)$ time. In addition, we would like a better idea of how these strategies compare to other edge-coloring implementations. In particular, we are interested in how the proposed hybrid strategy performs against Quickmatch. To compare these strategies, we still must implement the entire edge-coloring algorithm for the hybrid approach.

8 Acknowledgments

Finally, I would like to thank Professor Tom Cormen, who had the idea for the original static dummy edge strategy. His intuitions guided the development of the presented methods, and his editorial comments proved invaluable while drafting this thesis.

References

[COS01] Richard Cole, Kirstin Ost, and Stefan Schirra. Edge-coloring bipartite multigraphs in $O(E \log d)$ time. *Combinatorica*, 21(1):5–12, 2001.

[Han13] Andrew Hannigan. A heuristic algorithm for computing edge colorings on regular bipartite multigraphs. Dartmouth College Computer Science Senior Thesis, May 2013.

[Ost14] Stefanie Ostrowski. Chain match: An algorithm for finding a perfect matching of a regular bipartite multigraph. Technical Report 753, Dartmouth College, May 2014.