On Algorithms for Efficient Data Migration *

Joseph Hall † Jason Hartline† Anna R. Karlin† Jared Saia† John Wilkes ‡

Abstract

The *data migration* problem is the problem of computing an efficient plan for moving data stored on devices in a network from one configuration to another. Load balancing or changing usage patterns could necessitate such a rearrangement of data. In this paper, we consider the case where the objects are fixed-size and the network is complete. The direct migration problem is closely related to edge-coloring. However, because there are space constraints on the devices, the problem is more complex. Our main results are polynomial time algorithms for finding a near-optimal migration plan in the presence of space constraints when a certain number of additional nodes is available as temporary storage, and a 3/2-approximation for the case where data must be migrated directly to its destination.

1 Introduction.

The performance of large storage systems (such as disk farms) depends critically on having an assignment of data to storage devices that balances the load across devices or that optimizes a more complex cost function. Unfortunately, the optimal data layout is likely to change over time, for example, when either the workloads (access patterns or client service requirements) change, when new devices are added to the system, or when existing devices go down. Consequently, it is common in such systems to periodically compute a new optimal (or at least very good) assignment of data to devices based on newly predicted workloads and storage device specifications (such as speed and storage capacity) [1, 2, 4, 9]. Once the new assignment is computed, the data must be migrated from its old configuration to its new configuration. This migration must be done as quickly as possible, since during the time the migration is being performed, the storage system is running suboptimally. In this paper, we consider the problem of finding a plan for performing this migration as efficiently as possible.

The input to the *migration problem* is an initial and final configuration of data objects on devices, and a description of the storage system (the storage capacity of each device, the underlying network connecting the devices, etc.) Our goal is to find a *migration plan* that uses the existing network connections between storage devices to move the data from the initial configuration to the final configuration in the minimum amount of time. For obvious reasons, we require that all intermediate configurations in the plan be valid: they must obey the capacity constraints of the storage devices as well as usability requirements of the online system. (The migration process can be stopped at any time and the online system should still be able to run and maintain full functionality.)

The time it takes to perform a migration is a function of the sizes of the objects being transferred, the network link speeds and the degree of parallelism in the plan. A crucial constraint on the legal parallelism in any plan is that each storage device can be involved in the transfer (either sending or receiving, but not both) of only one object at a time.

Most variants one can imagine on this problem are NP-complete. The migration problem for networks of arbitrary topology is NP-complete even if all objects are the same size and each device has only one object that must be moved off of it. The problem is also NP-complete when there are just two storage devices connected by a link, if the objects are of arbitrary sizes.[1]

In this paper, we focus on the following challenging (also NP-complete) special case of the problem. We assume that the objects are all the same size. (This assumption is quite reasonable in practice if we allow ourselves to subdivide the existing variable sized objects into unit sized pieces, since the time it takes to send the subdivided object is about the same as the time it takes to send the entire object.) We also assume that the network topology is fully connected, that is, there is a direct bidirectional link between each pair of devices. Finally, we assume that there is at least one free space on each storage device in both the initial

*This research was supported in part by NSF grant EIA-9870740, BSF grant No. 96-00247 and a grant from Hewlett-Packard Research Labs.

†Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195; email: {jkh,hartline,karlin,saia}@cs.washington.edu

‡Hewlett-Packard Research Laboratories, Palo Alto, CA 95014; email: wilkes@hplajw.hpl.hp.com

[1]This observation was made by Dushyanth Narayanan.

and final configurations of the data. (Again, this is a very reasonable assumption in practice, since free space somewhere is required in order to move any objects, and having only one free space in the entire network would limit the solution to being sequential.)

We are thus led to describe the input to our problem as a directed multigraph $G = (V, E)$ without self-loops that we call the *demand graph*. Each vertex in the demand graph corresponds to a storage device, and each directed edge $(u, v) \in E$ represents an object that must be moved from storage device u (in the initial configuration) to storage device v (in the final configuration).

Since we are considering fixed-size objects, our migration plan can be divided into *stages* where each stage consists of a number of compatible sends, i.e., each stage is a matching. Thus, we can observe that the special case of our problem when there are no capacity constraints on the storage devices and sends must be direct is precisely the multigraph edge coloring problem (the directionality of the edges becomes irrelevant). This problem is of course NP-complete, but there are very good approximation algorithms for it, as we shall review in Section 2.

The storage migration application introduces two very interesting twists on the traditional edge coloring problem. In the first of these variants, we consider the question of whether *indirect plans* can help us to reduce the time it takes to perform a migration. In an indirect plan, an object might temporarily be sent to a storage device other than its final destination. It is easy to see that potentially this can significantly reduce the number of stages in the migration plan. As a first step towards attacking the problem of constructing near-optimal indirect plans, we introduce the concept of a *bypass node*. A bypass node is an extra storage device that can be used to store objects temporarily in an indirect plan. (In practice, some of the storage devices in the system will either not be involved or will be only minimally involved in the migration of objects and these devices can be used as bypass nodes.) A natural question to then ask is: what is the tradeoff between the number of bypass nodes available and the time it takes to perform the migration? In particular, how many bypass nodes are needed in order to perform the migration in $\Delta(G)$ steps, where $\Delta(G)$ (or Δ where G is understood) is the maximum total degree of any node in the demand graph G. (Δ is a trivial lower bound on the number of steps needed, no matter how many bypass nodes are available.)

The first result of the paper (Section 2.2) is a very simple algorithm that uses bypass nodes to reduce the time it takes to do a migration when there are no capacity constraints on the devices. We show how to efficiently find a migration plan for any directed multigraph G that completes in at most $2\lceil\Delta/2\rceil$ stages using at most $n/3$ bypass nodes, where n is the number of vertices in G. It is also easy to construct graphs for which $n/3$ bypass nodes are needed in order to complete the migration in Δ steps.

When capacity constraints are introduced (and we consider here the limiting case where there is the minimum possible free space at each vertex, including bypass nodes, such that there is at least one free space in both the initial and final configurations), we obtain our second, more complex, variant on the edge coloring problem. We can define this problem more abstractly as the *edge coloring with space constraints* problem:

The input to the problem is a directed multigraph G, where there are initially $F(v)$ free spaces on vertex v. (By the free space assumption, $F(v) = \max(d_{\text{in}}(v) - d_{\text{out}}(v), 0) + 1$, where $d_{\text{in}}(v)$ (resp. $d_{\text{out}}(v)$) is the in-degree (resp. out-degree) of vertex v.) The problem is to assign a positive integer (a color) to each edge so that the maximum integer assigned to any edge is minimized (i.e., the number of colors used is minimized) subject to the constraints that

- no two edges incident on the same vertex have the same color, and
- for each i and each vertex v, $c_{\text{in}}^{(i)}(v) - c_{\text{out}}^{(i)}(v) \leq F(v)$, where $c_{\text{in}}^{(i)}(v)$ (resp. $c_{\text{out}}^{(i)}(v)$) is the number of in-edges (resp. out-edges) incident to v with color at most i.

The second condition, which we refer to as the *space constraint condition*, captures the requirement that at all times the space consumed by data items moved onto a storage device minus the space consumed by data items moved off of that storage device can not exceed the initial free space on that device.

Obviously, not all edge-colorings of a multigraph (with edge directionality ignored) will satisfy the conditions of an edge-coloring with space constraints. However, it remains unclear how much harder this problem is than standard edge coloring.

The main results of our paper are the following:

- an algorithm for edge coloring with space constraints that uses at most $n/3$ bypass nodes and at most $4\lceil\Delta/4\rceil$ colors (presented in Sections 3.1 and 3.3);
- an algorithm for edge coloring with space constraints that uses no bypass nodes and at most $6\lceil\Delta/4\rceil$ colors (presented in Sections 3.2 and 3.3).

Interestingly, these are essentially the same as the worst case bounds for multigraph edge coloring *without* space constraints.

2 Migration without memory constraints.

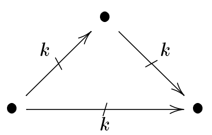
As discussed in the introduction, the direct migration problem in the absence of capacity constraints is precisely equivalent to the problem of edge-coloring a multigraph using the minimum number of colors. The chromatic index of a graph, χ', is the number of colors in the optimal edge-coloring. Δ, the maximum degree of any vertex in the graph, is a trivial lower bound on χ'. Edge-coloring is of course NP-complete [6]. The classic result of Vizing [8] shows that there is a polynomial time $(\Delta+1)$-approximation algorithm for simple graphs. For multigraphs, the best known result is a polynomial time algorithm that uses at most $9\chi'/8 + 3/4$ colors [3, 5]. (This approximation algorithm actually colors the graph optimally if $\chi' \geq 9\Delta/8 + 3/4$.) It is also well-known that χ' is bounded from above by $3\Delta/2$, and this is tight [7].

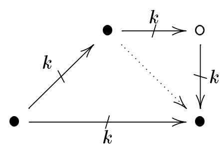
2.1 Bypass Nodes. As stated above, an optimal direct migration takes at least χ' parallel steps. If our solution is not required to send objects directly from source to destination it is possible that there is a migration plan that takes less than χ' stages. In general, our goal will be to use a small number of bypass nodes, extra storage devices in the network available for temporarily storing objects, to perform the migration in Δ stages.

Definition 2.1. *A directed edge (v, w) in a demand graph is* bypassed *if it is replaced by two edges, one from v to a bypass node, and one from that bypass node to w.*

An extremely important constraint that bypassing an edge imposes is that the object must be sent to the bypass node before it can be sent from the bypass node. In this sense, edges to and from bypass nodes are special.

The following example, shows how a bypass node might be used. In the graph, G, on the left, each edge is duplicated k times and clearly $\chi' = 3k$. However, using only one bypass node, we can perform the migration in $\Delta = 2k$ stages as shown on the right. (The bypass node is shown as $\circ$.)





More generally, by replicating the graph G $n/3$ times, we see that there exist graphs which require $n/3$ bypass nodes in order to complete a migration in Δ steps.

2.2 Indirect Migration without memory constraints. We warm up with a simple algorithm for indirect migration without memory constraints that requires at most $2\lceil\Delta/2\rceil$ stages and uses at most $n/3$ bypass nodes on any multigraph G. Although this result is essentially subsumed by the analogous result with memory constraints, the simple ideas of this algorithm are important building blocks as we move on to the more complicated scenarios.

Algorithm 2.1. Bypass Algorithm without memory constraints

1. Add dummy self-loops and edges to G to make it regular and even degree $(2\lceil\Delta/2\rceil)$. (This is trivial – for completeness it is described in Appendix A.)

2. Compute a 2-factor decomposition of G, viewed as undirected. (This is standard – for completeness it is described in Appendix B.)

3. Transfer the objects associated with each 2-factor in 2 steps using at most $n/3$ bypass nodes. This is done by bypassing one edge in each odd cycle, thus making all cycles even. Send every other edge in each cycle (including the edge to the bypass node if there is one) in the first stage and the remaining edges in the second.

This algorithm uses a total of $2\lceil\Delta/2\rceil$ stages – two for each 2-factor of the graph. The bypass nodes are in use only after every other stage and can be completely reused. Thus, no more that $n/3$ bypass nodes are used total, at most one for every odd cycle.

Notice that we can perform the migration in $3\lceil\Delta/2\rceil$ stages without bypass nodes, if we use three stages for each 2-factor instead of two (a well-known folk result, see [7]). However, the best multigraph edge coloring approximation algorithms achieve better bounds.

3 Migration with memory constraints.

We now turn to the problem of migration (or edge coloring) with space constraints. For this problem, we will show how to compute a $6\lceil\Delta/4\rceil$ stage direct migration plan and a $4\lceil\Delta/4\rceil$ stage indirect migration with $n/3$ bypass nodes. As mentioned in Section 2, these bounds essentially match the worst case lower bounds for the problem without space constraints.

Our strategy for obtaining these results is to reduce the problem of finding an efficient migration plan with space constraints in a general multigraph to the problem of finding an efficient migration plan with space constraints for 4-regular multigraphs. We first present efficient algorithms for finding migration plans for regular multigraphs of degree four. Specifically, we show how to find a 4-stage indirect migration plan using at most $n/3$ bypass nodes and a 6-stage direct migration plan. We will then give the reduction.

3.1 Indirect Migration of 4-Regular multigraphs with memory constraints.

Algorithm 3.1 presents our construction of an indirect migration plan for 4-regular multigraphs with space constraints. We begin with some intuition for the algortihms.

ALGORITHM 3.1. The bypass algorithm for 4-regular multigraphs

1. Split each hard vertex into two representative vertices with one having two in-edges and the other having two out-edges. This breaks the graph into connected components (when the edges are viewed as undirected). (Figure 1(a) shows an example graph, and Figure 1(b) shows the result of splitting hard vertices, shown as $\star$.)

2. Construct an Euler tour for each component (ignoring the directionalities of the edges) (Figure 1(c) shows the resulting Euler tours.)

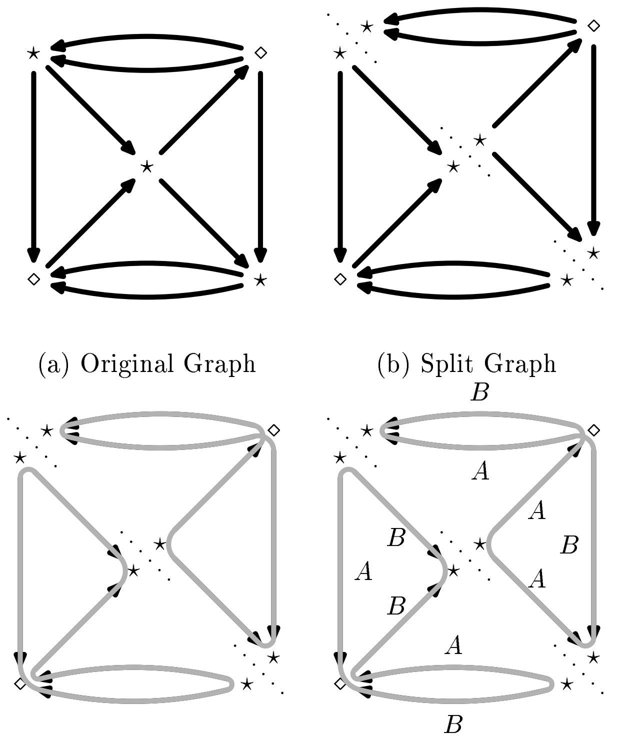
3. Alternately A/B label the edges along the Euler tour of each of the even components.

4. While there exist a pair of odd components that share a vertex (each component contains one of the split hard vertices), label the two out-edges of the split vertex A, label the two in-edges of the split vertex B, and alternately A/B label the remaining portions of the two Euler tours. (Figure 1(d) shows an example.)

5. Repeatedly select an unlabeled odd component and perform the following step:

   Within that component, bypass exactly one edge, say (u, v), where the edge is chosen using Procedure 3.1. Label A the edge from u to the new bypass node and B the edge from the bypass node to v. Alternately A/B label the remaining edges in the tour.

6. The resulting A and B subgraphs have maximum degree 2. (The only vertices in either graph of degree 1 are bypass nodes.) Bypass an edge in each odd cycle that occurs in either the A or B graph, converting all cycles to even-length cycles. Alternately color the edges in each A cycle 1 and 2, and alternately color the edges in the B cycle 3 and 4.



(a) Original Graph (b) Split Graph

(c) Euler-Tours (d) Step 4 Applied

Figure 1: Illustration of first few steps of the algorithm.

The difficulty in constructing an efficient migration plan arises from dealing with the vertices with exactly two in-edges and two out-edges. We call such vertices *hard vertices*, since we are required to send at least one of the out-edges from such a vertex before we send both in-edges. We refer to all other vertices as *easy vertices* since they have at least as much free space initially as they have in-edges, and hence their edges can be sent in any order.[2]

We formalize in the following proposition the high-level construction that we use to ensure that space constraints are never violated.

PROPOSITION 3.1. *Let G be a 4-regular multigraph. Suppose that the edges of G are A/B labeled such that each hard vertex has two of its incident edges labeled A, and two of its incident edges labeled B, with at least one out-edge labeled A. Then if all edges labeled A are sent, in any order, before any edge labeled B, there will never be a space constraint violation.*

[2]Recall that our free space assumption is that each vertex has one free space at the start and finish of the migration.

Thus, our goal is reduced to finding an A/B labeling that meets the conditions of Proposition 3.1, and that can be performed in as few stages as possible.

Interestingly, if there are no *odd vertices* (shown in the figures as $\diamond$), vertices such that the parity of their in-degree (and out-degree) is odd, then the problem is easy: We split each vertex into two with the property that each new vertex has exactly two edges of the same orientation. This new graph need not be connected. We construct an Euler-tour of each component (ignoring the directionality of the edges) and alternately label edges along these tours A and B. No conflicts arise in the A/B labeling because the tours have even length – each vertex has either only in-edges or only out-edges so the tour passes through an even number of vertices. The A and B induced subgraphs are a 2-factor decomposition of the original graph with the property that exactly one out-edge is labeled A. We can thus use our standard method for performing migration with or without bypass nodes given a 2-factor decomposition. With bypass nodes, this method sends the A-edges in stages one and two and the B-edges in stages three and four.

When there are both odd vertices and hard vertices, the problem becomes more difficult. In particular, it is not hard to show that there exist 4-regular multigraphs in which *no* A/B labeling of the graph ensures that *every* vertex has two incident A edges and two incident B edges, with at least one A-labeled out-edge from each hard vertex. To solve the problem, we will need to bypass some of the edges in the graph.

Our algorithm starts out very much like the algorithm just described for graphs with no odd nodes, but now we split only the hard vertices into two representative vertices with one having two in-edges and the other having two out-edges.

Each resulting component (disregarding edge directionality) still has an Euler tour of course, but not all components have even length. We call those with even length tours *even components* and those with odd length tours *odd components*. Those that do have even length can be alternately A/B labeled. We could then bypass one edge in each odd component, and A/B label the resulting even-length tour. Note that the choice of bypassed edge determines the A/B labeling of the tour – as discussed in Section 2.1 the incoming edge to the bypass node must be labeled A and the outgoing edge must be labeled B.

Unfortunately, this will not give us a good bound on bypass nodes, since there can be $2n/5$ odd components (Figure 1). We get around this problem by observing that the A/B labeling so constructed satisfies a more restrictive property than that needed to obey space constraints – it guarantees that every hard vertex has *both* an in-edge and an out-edge labeled A. This excludes perfectly legal labelings that have hard vertices with two out-edges labeled A. Indeed, it is not possible in general to beat the $2n/5$ bound on bypass nodes if we disallow both out-edges from being labeled A.

Therefore, the algorithm will sometimes have to label both out-edges from a hard vertex A. In our algorithm, this happens whenever we find a pair of odd components that share representatives of the same hard vertex. We can merge the two odd components into a single even component which can be A/B labeled such that both out-edges of the shared hard vertex are labeled A. When no remaining unlabeled odd components can be merged in this fashion, we are guaranteed that there are at most $n/3$ odd components remaining.

Unfortunately, our work is not done, since in addition to the bypass nodes introduced for each remaining odd component (which have one incident edge labeled A and one incident edge labeled B), there may be odd cycles in the A and B induced graphs. We will also need to bypass one edge in each of these odd cycles. If we are not careful about which edge we bypass in the odd component, we will end up with too many bypass nodes used to break odd A or B cycles. The heart of our algorithm and analysis is judiciously choosing which edge to bypass in each odd component. With carefully accounting for these bypass nodes in the analysis, we show that the total number of bypass nodes used is at most $n/3$.

3.1.1 Terminology.

The result of steps 1-4 is a decomposition of the graph into a collection of unlabeled odd components that are connected via A or B labeled paths (which correspond to edges that were in even components, or odd components that were merged and labeled in Step 4)

Within each unlabeled odd component, we have two types of vertices: *internal vertices*, which have all their edges inside the odd component, and *external vertices*, which have only two edges, either both directed towards the vertex or both directed away from the vertex. Since adjacent odd components have been labeled (Step 4), the other two edges incident to each external vertex are both already labeled (one A and one B).

Figure 2 shows a example of what the resulting graph might look like. There are four unlabeled components ($C_1, \ldots, C_4$). Classify the A and B paths emanating from each external vertex as either an *external loop* if its two endpoints (external vertices) are in the same unlabeled component, or as an *external path* if its two endpoints are in different unlabeled components.

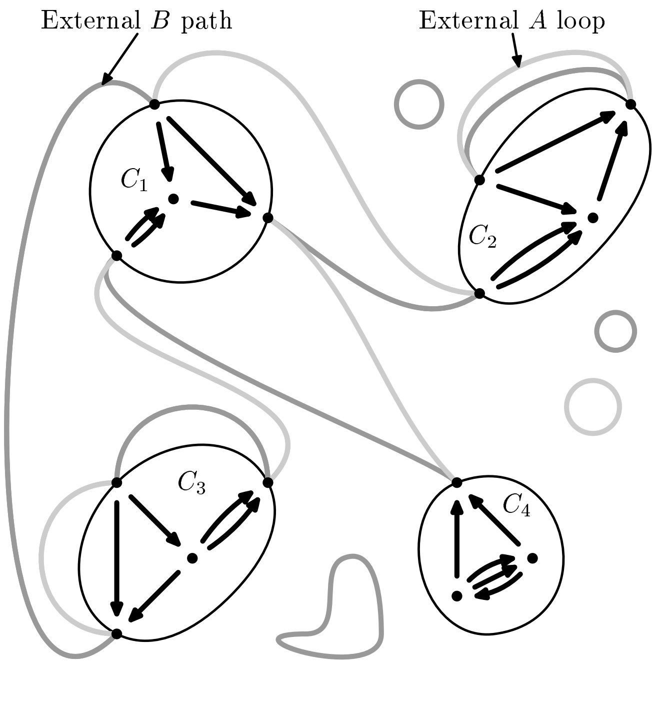


Figure 2: An example of what the graph might look like after Step 4.

Procedure 3.1. *A/B coloring odd components* There are two cases:

1. *There is only one external vertex.*

   Within this case, there are two subcases:

   - If there is a pair of internal vertices that are each not adjacent to the external vertex that have an edge between them, bypass that edge.
   - If not, there are only two possible graphs, shown below. We omit the justification of this fact. (The external vertex is on the left and the directionality of the edges is not shown.) Bypass the dashed edge.

2. *There are 3 or more external vertices.*

   Let v be the external vertex incident to the largest number of external loops. Bypass one of its incident internal edges. If the resulting A/B labeling of this component's Euler tour creates an A or B cycle with v (containing an external loop and an internal path connecting the endpoints of the external loop), switch which one of v's internal edges is bypassed.

Analysis. We now turn to the analysis of the algorithm. By construction, edges are A/B labeled so that the conditions of Proposition 3.1 are met, and hence we have:

Lemma 3.1. *Algorithm 3.1 computes a migration plan that respects space constraints.*

Our main theorem is the following:

Theorem 3.1. *The bypass algorithm for edge coloring 4-regular multigraphs with space constraints uses four colors and at most $n/3$ bypass nodes, where n is the number of nodes in the graph.*

Proof. By construction, the algorithm described uses 4 colors. We have only to show that it uses at most $n/3$ bypass nodes. We do this by "crediting" each bypass node used in the A stages with a distinct set of three vertices in the graph, and crediting each bypass node used in the B stages with a distinct set of three vertices in the graph.

A bypass node that is created in order to break an odd A cycle (or B cycle) will be credited with three vertices in that cycle. Notice that bypass nodes used to break A cycles can be reused to break B cycles. The tricky part of the argument will be to show that each bypass node that is created when an odd component is A/B labeled can also be credited with 3 vertices.

The accounting scheme we use is based on the following observations about the structure of what happens when step 5 is performed. Prior to performing this step, we have exactly one A external path or loop and exactly one B external path or loop connected to each external node. When we pick an edge inside the component to bypass, and A/B label the component, every internal vertex (which is of degree 4) gets two of its incident edges labeled A and two of its incident edges labeled B, and every external vertex gets one of its incident internal edges labeled A and one labeled B. Thus the *internal* A path (or B path) emanating from an external node either terminates at a bypass node, in which case we call it an *end path*, or it terminates at another external node, in which case we call it an *inscribed path*. Thus, looking at the A subgraph (or similarly the B subgraph) of an odd component with $2k+1$ external vertices, we obtain precisely k disjoint inscribed A paths and one A end path. (Note that the odd component must have an odd number of external vertices, since each internal vertex appears twice in the Euler tour of the component and each external vertex appears once and the length of the tour is odd.)

For the case where an odd component has exactly one external vertex, we can simply verify that breaking the proposed edge results in three vertices not in odd cycles that can be credited to the A and B end-path. In both graphs, bypassing any edge except the ones incident to the external vertex guarantees that there will be no odd cycle created inside the component. Since only the end paths leave the components all vertices inside the component are not in odd cycles and can be credited to the bypass node.

If the odd component has more than one external vertex, then it must have at least three. We will credit the bypass node with the external node on the end path terminating at the bypass node (which in general will be different for A and B), and with two other external vertices in the component. The difficulty is that if we are not careful about which edge in the component we bypass, the two other external vertices we select can have an inscribed A path between them and an external A loop, and thus might end up in a short odd A cycle. If this happens, we will violate our condition of crediting each bypass node with distinct vertices in the graph, since the bypass node created to break this short A cycle will also be credited with these vertices. Therefore, we choose an edge to bypass so that the other two external vertices we credit to the bypass node are not in a short cycle.

We find that it is sufficient to guarantee that for each odd component processed in Step 5, in the resulting A graph (resp. in the resulting B graph) one of the following situations holds:

1. There are at least two external paths labeled A (resp. B).

   If there are at least two external paths labeled A, one of them is not connected to the A end path. Thus, there is an inscribed A path that connects the external path to some other external path or loop. In this case, we credit the bypass node (in stage A) with the two external vertices on this inscribed path and with the external vertex on the A end path connected to it.

2. The component is labeled so that there is an A (resp. B) external loop that does not form a cycle with an inscribed A (resp. B) path.

   In this case, the A external loop is connected to some other A external loop or path via an inscribed A path. We can again credit the bypass node with the external vertices on this inscribed path and with the external vertex on the end path connected to it.

If the edge selected to bypass in Step 5 results in one of these two situations holding, we say that a *good* edge was bypassed.

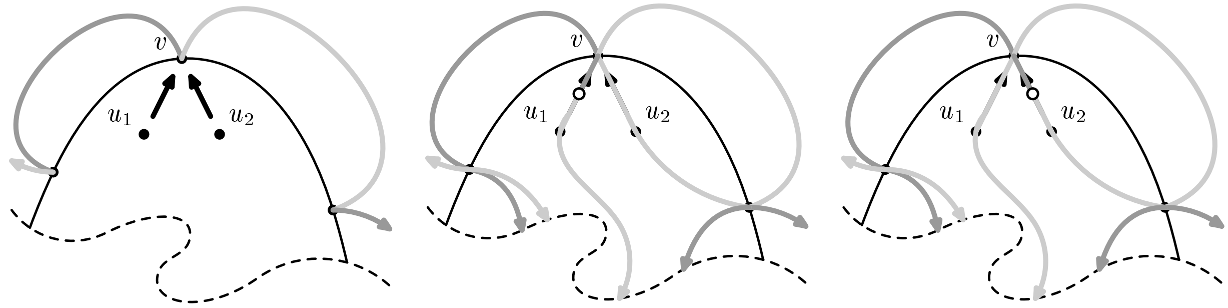
Notice that in both of these situations, the two external vertices credited to the bypass node may end up in an odd A (or B) cycle. We claim, however, that if this happens, it is an odd cycle created by two or more external A loops or paths and hence it has length at least five. Since we have only credited two of the external vertices on the cycle to the bypass node created in Step 5, we still have three vertices in the odd cycle that can be credited to the bypass node that will be used to break the cycle. In fact, the argument is slightly more complicated than this – we omit the details.

Finally, we must show that the procedure used to select an edge to bypass in each odd component with at least three external vertices results in bypassing a good edge.

Let v be the odd component's external vertex with the largest number of incident external loops as chosen by the algorithm. If there is no external A (likewise B) loop at v then the odd component has at least two A paths and, as such, any edge is good for A. To see why this is the case note that if v has no external loops then no external vertices have loops so clearly there are at least two external paths of both labels. If v has one external loop of label B then the other endpoint of the loop has a B loop (the same one). Since v has the largest number of loops, this other vertex can not have an A loop. Thus, both this vertex and v have A paths.

Now we argue that our choice of edge to bypass guarantees that any external loop emanating from v does not form a cycle with an inscribed path. Suppose v's internal edges are both in-edges[3] from vertices u_1 and u_2 and we bypass the edge (u_1, v) using bypass node b. The edges (u_1, b) and (u_2, v) are thus both labeled A and the edge (b, v) is labeled B (See Figure 3(b)). As such the B loop, if there is one, does not create a cycle. Assume that there is an A loop. If not, we are done. If labeling the rest of the odd component according the the Euler tour does not cause an incribed path to be created between the external A loop's endpoints then we are also done and (u_1, v) is good for A. Otherwise, this incribed path must go through the edge (u_2, v). The algorithm swaps which edge is bypassed so that edges (u_2, b) and (u_1, v) are both labeled A and the edge (b, v) is labeled B (See Figure 3(c)). The bypassed edge is still good for B. We now argue that is also good for A. The rest of the labeling remains the same as the edges incident on u_1 and u_2 have not changed labels. Since the rest of the labeling does not change, there is still an internal A path from the one endpoint of the external A loop to u_2. This path continues from u_2 to b and terminates. Thus, the edge (u_2, v) is good for A. ■

[3] The case where they are both out-edges is similar.



(a) Initial (b) (u_1, v) bypassed (c) (u_2, v) bypassed

Figure 3: Choosing which edge to bypass.

3.2 4-regular migration without bypass nodes. We next show how to compute a six stage migration plan of a 4-regular multigraph without using bypass nodes. The first 4 steps of the algorithm are the same as in Algorithm 3.1. Procedure 3.2 replaces Steps 5 and 6 of Algorithm 3.1, the only steps that used bypass nodes, with a construction that uses two extra colors instead. In this procedure, the label A will be for stages 1, 2, and 3, while the label B will be for stages 4, 5, and 6.

After completing Step 4 of the previous algorithm, the graph contains a number of unlabeled disjoint odd components connected by A/B labeled paths and loops (Figure 2). We make the following additional observations about the graph:

- Each odd component contains at least one odd vertex. (Otherwise, the tour would be of even length.)

- Since odd unlabeled components are disjoint, and odd vertices are always internal, every path between two odd vertices in different odd unlabeled components has length at least three.

PROCEDURE 3.2. Final steps in 4-regular migration without bypass nodes

5′. A/B label each remaining odd component by starting with an odd vertex, v, and the label B and following the Euler tour of the component labeling edges alternately A and B. Note: v will have three B labeled edges incident and one A labeled edge.

6′. Color the A and B induced subgraphs:

  (a) Color the A induced subgraph (note: the A induced subgraph is a set of paths and cycles):

    i. Break all odd length cycles by coloring one edge in them 3.

    ii. Color all remaining paths and even cycles with the colors 1 and 2.

  (b) Color the B induced subgraph (note: the B induced subgraph has vertices of degree two and degree three only):

    i. Color one edge 6 on each degree three vertex. We are left with cycles and paths.

    ii. Color one edge 6 in each odd length cycle to convert the cycle into a path. Choose an edge that is not incident on one of the degree three vertices (which is possible because of the second observation above).

    iii. The remaining graph is just paths and even cycles. Color them with colors 4 and 5.

The algorithm can be easily seen to meet the conditions of Proposition 3.1. Straightforward arguments (omitted in this extended abstract) thus give us the following theorem:

THEOREM 3.2. *The above algorithm computes a proper six coloring of the graph that respects the space constraints of the hard vertices.*

3.3 Reduction to 4-regular graphs. We next show how to reduce the general migration problem with space constraints to the problem of migration with space constraints on 4-regular multigraphs. Ideally, we might like to split the graph into 2-factors, such that sending the edges within a 2-factor in any order satisfies our space constraints, and so that after each 2-factor is sent, there is still one free space at each vertex. This is not always possible. What we are able to do is to split the graph into 4-factors such that after each 4-factor is sent, there is once again a free space at each vertex. To partition the edges of the graph in this way, we need, roughly speaking, to match up in-edges of a vertex with corresponding out-edges. Algorithm 3.2 gives the precise details.

The key lemma is the following:

LEMMA 3.2. *A migration that repeatedly picks one edge incident to v_{in} and one incident to v_{out} to send in either order will never violate the space constraints of v.*

Proof. We consider two cases, depending on which of v_{in} or v_{out} has incident edges only of one type (at least one of them must).

Case: v_{out} has only incident out-edges.

Then since one of the edges chosen is an out-edge there will be at least one out-edge sent for every in-edge so the free space after the two edges are sent is at least what it was before.

Case: v_{in} has only incident in-edges.

If $\ell = d_{\text{in}} - d_{\text{out}}$, then we know by the free space assumption that there are at least $\ell + 1$ free spaces initially. We allocate this free space as follows:

- The number of times that two in-edges are chosen is exactly $\ell/2$ we allocate two free spaces to each of these.
- The remaining times we choose an in-edge and an out-edge. All of these cases will share the one remaining free-space. Since both an in-edge and an out-edge are sent, we will regain the free space again after the two edges are sent.

Note that since we always have exactly one edge incident to v_{in} and exactly one incident to v_{out} if the edge happens to be a dummy self loop then it is the only edge chosen at this step of the migration. Since nothing happens in this case, the available free space remains unchanged. There is also at most one dummy edge and it is incident to v_{in} or v_{out}, whichever has less of its type of edge. Our argument above focused on the edges incident on v_{in} or v_{out}, whichever has more of its type of edge, so the arguments still hold when a dummy edge is present. ■

ALGORITHM 3.2. The reduction to 4-regular graphs

1. Make G regular with degree a multiple of four ($4k$) (using the procedure in Appendix A).
2. Split each vertex v into v_{in} and v_{out} assigning v's edges to either v_{in} or v_{out} to get G':
   (a) Assign dummy self loops, (v, v), to both v_{in} and v_{out} as $(v_{\text{out}}, v_{\text{in}})$.
   (b) Assign the remaining in-edges to v_{in} and the remaining out-edges to v_{out} (excluding the dummy edge).
   (c) Assign the dummy edge, if there is one, to the representative of v with the least number of adjacent edges.
   (d) Make the degrees of v_{in} and v_{out} equal by moving real edges from one to the other until they have equal degree.

   G' has $2n$ vertices and is $2k$-regular.
3. Compute a 2-factoring of G' (viewed as undirected). This gives k 2-factors.
4. In each 2-factor merge vertex representatives back together. That is, v_{in} and v_{out} become v again. The result is k 4-factors of our original graph G. The problem is thus reduced to computing a migration with space constraints on these 4-factors of G.

We thus obtain:

THEOREM 3.3. *Algorithm 3.2 reduces the problem of performing a migration with space constraints on an arbitrary graph to that of performing a series of migrations with space constraints on 4-regular multigraphs.*

Combining this theorem with Theorems 3.1 and 3.2 gives us the following corollaries:

COROLLARY 3.1. *There is an algorithm that takes as input an arbitrary directed multigraph of maximum degree Δ and finds a $4\lceil\Delta/4\rceil$ stage migration plan using at most $n/3$ bypass nodes.*

COROLLARY 3.2. *There is an algorithm that takes as input an arbitrary directed multigraph of maximum degree Δ and finds a $6\lceil\Delta/4\rceil$ stage migration plan without bypass nodes.*

4 Open Problems.

We have presented some first results on two intriguing and well-motivated twists on the traditional edge coloring problem. Numerous open problems remain including the following:

What is the relationship between the chromatic index of a graph and its "chromatic index with space constraints"? Are there better approximation algorithms for this latter problem than those presented here? Is there a "Vizing-like" theorem for edge coloring simple graphs with space constraints?

What is the tradeoff between the number of bypass nodes available and the number of stages (or colors) required? What if it is possible to bypass to vertices in the graph (as opposed to extra nodes)?

References

[1] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E.Shriver, M.Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *Presented at 5th Intl. Workshop on Quality of Service*, Columbia Univ., New York, June 1997.

[2] B. Gavish and O. R. Liu Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33:177–189, 1990.

[3] M. K. Goldberg. Edge-coloring of multigraphs: Recoloring technique. *J. Graph Theory*, 8:121–137, 1984.

[4] I. Golubchik, S. Khuller S. Khanna, R. Thurimella, and A. Zhu. Approximation Algorithms for Data Placement on Parallel Disks. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete ALgorithms*, pages 223–232, 2000.

[5] D. S. Hochbaum, T. Nishizeki, and D. B. Shmoys. A better than "Best Possible" algorithm to edge color multigraphs. *J. of Algorithms*, 7:79–104, 1996.

[6] I. J. Hoyer. The NP-completeness of edge coloring. *SIAM J. Comput.*, 10:718–720, 1981.

[7] C. E. Shannon. A theorem on colouring lines of a network. *J. Math. Phys.*, 28:148–151, 1949.

[8] V. G. Vizing. On an estimate of the chromatic class of a p-graph. *Diskret. Anal.*, 3:25–30, 1964.

[9] J. Wolf. The Placement Optimization Problem: a practical solution to the disk file assignment problem. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–10, 1989.

A Obtaining a Regular Graph.

Some of our algorithms require regular graphs of degree either a multiple of 2 or 4. Let Δ' be this desired degree, either $2\lceil\Delta/2\rceil$ or $4\lceil\Delta/4\rceil$. We construct such directed regular multigraphs as follows.

Algorithm A.1. Making a directed multigraph Δ'-regular

1. While there exists a vertex with degree less than $\Delta' - 1$, add a self loop to that vertex.
2. While there exist two vertices of degree $\Delta' - 1$, add an arbitrarily directed edge between them.

Every vertex in the resulting graph has degree Δ'.

B 2-factor decomposition.

It is well known that a $2k$-regular multigraph can be factored into k 2-factors. For completeness, we review an algorithm for doing this. This algorithm takes an undirected multigraph G with degree $\Delta = 2k$ and returns k 2-factors of G. We will be performing this operation on directed multigraphs. In this case, the directions of the edges are ignored during the factoring algorithm.

Algorithm B.1. 2-factoring a multigraph

1. Construct an Euler-tour of G.
2. Orient the edges according to the direction of the tour. That is, if the tour enters v on edge e_1 and leaves on edge e_2, then e_1 is an in-edge to v and e_2 is an out-edge. Thus we have $d_{\text{in}} = d_{\text{out}} = k$.
3. Set up a bipartite matching problem, B_G, with a representative of each vertex in the graph on both sides. Add in all directed edges going from left to right. Note that each edge is represented it the matching problem exactly once.
4. Find a matching (which is guaranteed to exist by Hall's Theorem). The matched edges induce a 2-factor of the original graph. Remove these edges from B_G and repeat this step until there are no edges left.