Algorithmica
DOI 10.1007/s00453-008-9214-y

Algorithms for Data Migration

E. Anderson · J. Hall · J. Hartline · M. Hobbes · A. Karlin · J. Saia · R. Swaminathan · J. Wilkes

Received: 15 August 2007 / Accepted: 16 July 2008

Abstract The *data migration* problem is the problem of computing a plan for moving data objects stored on devices in a network from one configuration to another. Load balancing or changing usage patterns might necessitate such a rearrangement

E. Anderson · R. Swaminathan · J. Wilkes
Hewlett–Packard Laboratories, Palo Alto, CA 94304, USA

E. Anderson
e-mail: eric.anderson4@hp.com

R. Swaminathan
e-mail: ram.swaminathan@hp.com

J. Wilkes
e-mail: john.wilkes@hp.com

J. Hall · A. Karlin
Department of Computer Science, University of Washington, Seattle, WA 98195, USA

J. Hall
e-mail: jkh@cs.washington.edu

A. Karlin
e-mail: karlin@cs.washington.edu

J. Hartline
Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA
e-mail: hartline@eecs.northwestern.edu

M. Hobbes
School of Engineering and Information Technology, Deakin University, Geelong, VIC 3217, Australia
e-mail: mick@deakin.edu.au

J. Saia (✉)
Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, USA
e-mail: saia@cs.unm.edu

of data. In this paper, we consider the case where the objects are fixed-size and the network is complete. Our results are both theoretical and empirical. Our main theoretical results are (1) a polynomial time algorithm 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 (2) a $3/2$-approximation algorithm for the case where data must be migrated directly to its destination. We also run extensive experiments on several algorithms for various data migration problems and show that empirically, many algorithms perform better in practice than their theoretical bounds suggest. We conclude that many of the algorithms we present are both practical and effective for data migration.

Keywords Data migration · Edge coloring · Load-balancing · Space constraints · Approximation algorithms

1 Introduction

The performance of modern day large-scale storage systems (such as disk farms) depends critically on having an assignment of data to storage devices that balances the load across the devices or that optimizes a more complex cost function. Unfortunately, the optimal data layout is likely to change over time because of workload changes, device additions, or device failures. Thus, it is common to periodically compute a new assignment of data to devices [3, 5–7, 22], either at regular intervals or on demand as system changes occur. Once the new assignment is computed, the data must be migrated from the old configuration to the new configuration. This migration must be done as efficiently as possible to minimize the impact of the migration on the system. The large size of the data objects (gigabytes are common) and the large amount of total data (can be petabytes) makes migration a process which can easily take several days.

In this paper, we consider the problem of finding an efficient migration plan. We focus solely on the off-line migration problem, i.e., we ignore the load imposed by user requests for data objects during the migration. Our motivation for studying this problem lies in migrating data for large-scale storage system management tools such as *Hippodrome* [2]. Hippodrome automatically adapts to changing demands on a storage system without human intervention. It analyzes a running workload of requests to data objects, calculates a new load-balancing configuration of the objects and then migrates the objects. An offline migration can be performed as a background process or at a time when loads from user requests are low (e.g. over the weekend).

The input to our *migration problem* is an initial and final configuration of data objects on devices, and a description of the storage system such as the storage capacity of each device, and the underlying network connecting the devices. 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 space constraints of the storage devices as well as usability requirements of the on-line system. (The migration process can be stopped at any time and the on-line 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]

We will always make the following assumptions.

1. The data objects are all the same size;
2. There is at least one free space on each storage device in both the initial and final configurations of the data;
3. There is a direct bidirectional link between each pair of devices and each link has the same bandwidth; and
4. Every device has the same read and write speed.

The first 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. The second assumption is also reasonable 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.

The third and fourth assumptions are reasonable if we assume that the storage devices are connected in a local area network (SAN), for example in a disk farm. The third assumption is reasonable since the topologies in SANs are generally close to being complete. The fourth assumption is reasonable since an organization building a SAN of storage devices would typically buy devices which have similar performance characteristics.

With these assumptions, we are 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 node 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). An example of how a demand graph is defined based on an initial and goal configuration is given in Fig. 1.

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 space constraints on the storage devices, and sends must be direct, is precisely the multigraph edge coloring problem (the directionality of the edges becomes irrelevant

[1] This observation was made by Dushyanth Narayanan.

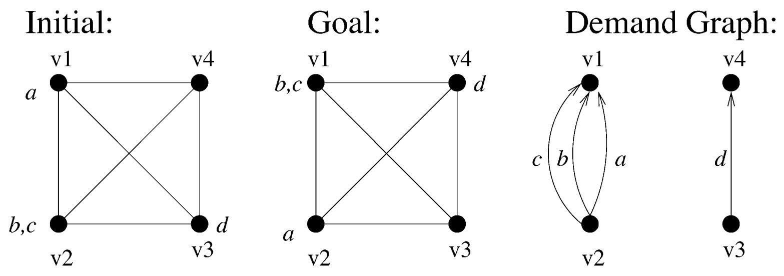


Fig. 1 An example demand graph. v1, v2, v3, v4 are devices in the network and the edges in the first two graphs represent links between devices. a, b, c, d are the data objects which must be moved from the initial configuration to the goal configuration

and the colors of the edges represent the time steps when objects are sent). This problem is of course NP-complete, but there are good approximation algorithms for it, as we shall review in Sect. 1.4. The storage migration application introduces two interesting twists on the traditional edge coloring problem: edge coloring with bypass nodes and edge coloring with space constraints. We review these variants briefly in the next section.

1.1 Data Migration Variants

1.1.1 Bypass Nodes

In the first variant on edge coloring, 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 this might 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 may 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 trade-off 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.)

An optimal direct migration takes at least χ' parallel steps where χ' is the chromatic index of the demand graph (the minimum number of colors in any edge coloring). 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 essentially Δ stages.

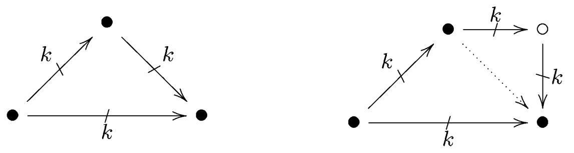


Fig. 2 Example of how to use a bypass node. In the graph on the *left*, each edge is duplicated k times and clearly $\chi' = 3k$. However, using only one bypass node, we can perform the migration in $\Delta = 2k$ stages as shown on the *right*. (The bypass node is shown as "$\circ$")

Definition 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 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. Figure 2 gives an example of how a bypass node can be used.

By replicating the graph on the left side of Fig. 2 $n/3$ times, we see that there exist graphs which require $n/3$ bypass nodes in order to complete a migration in Δ steps.

1.1.2 Space Constraints

When space constraints are introduced, we obtain our second, more complex, variant on the edge coloring problem. We consider here the limiting case where there is the minimum possible free space at each node, including bypass nodes, such that there is at least one free space in both the initial and final configurations. 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 node v. By the free space assumption, $F(v) = \max(d_{\text{in}}(v) - d_{\text{out}}(v), 0) + 1$, where $d_{\text{in}}(v)$ is the in-degree and $d_{\text{out}}(v)$ is the out-degree of node 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 node have the same color, and
- for each i and each node v, $c^{(i)}_{\text{in}}(v) - c^{(i)}_{\text{out}}(v) \le F(v)$, where $c^{(i)}_{\text{in}}(v)$ (resp. $c^{(i)}_{\text{out}}(v)$) 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 cannot 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.

Table 1 Theoretical bounds for our algorithms

Algorithm	Type	Space constraints	Plan length	Worst case maximum bypass nodes
Edge-coloring [17]	Direct	No	$3\lceil\Delta/2\rceil$	0
2-*factoring*	Indirect	No	$2\lceil\Delta/2\rceil$	$n/3$
4-*factoring direct*	Direct	Yes	$6\lceil\Delta/4\rceil$	0
4-*factoring indirect*	Indirect	Yes	$4\lceil\Delta/4\rceil$	$n/3$
Max-Degree-Matching	Indirect	No	Δ	$2n/3$
Greedy-Matching	Direct	Yes	Unknown	0

1.2 Theoretical Results

Table 1 gives a summary of the theoretical results for all algorithms presented in this paper. Below we list the algorithms with the best theoretical bounds.

- 2-*factoring*: an algorithm for edge coloring that uses at most $n/3$ bypass nodes and at most $2\lceil\Delta/2\rceil$ colors (presented in Sect. 2.1).
- 4-*factoring direct*: an algorithm for edge coloring with space constraints that uses no bypass nodes and at most $6\lceil\Delta/4\rceil$ colors (presented in Sects. 2.2.2 and 2.2.3).
- 4-*factoring indirect*: 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 Sects. 2.2.1 and 2.2.3).

Interestingly, the worst-case bounds for the algorithms with space constraints are essentially the same for multigraph edge coloring *without* space constraints.

1.3 Empirical Results

While the algorithms presented in this paper have near optimal worst case guarantees, it still remains to see how well they perform in practice. In Sect. 3, we evaluate a set of data migration algorithms on the types of data migration problems which might typically occur in practice. In addition to testing the algorithms from Sect. 2, we also describe and test two new migration algorithms which perform well empirically.

The first new algorithm is called *Max-Degree-Matching*. This algorithm can find a migration taking Δ steps while using no more than $2n/3$ bypass nodes. We compare this to 2-*factoring*. While 2-*factoring* has better theoretical bounds than *Max-Degree-Matching*, we will see that *Max-Degree-Matching* uses fewer bypass nodes on almost all tested demand graphs.

For migration with space constraints, we introduce a new algorithm, *Greedy-Matching*, which uses no bypass nodes. We know of no good bound on the number of time steps taken by *Greedy-Matching* in the worst case; however, in our experiments, *Greedy-Matching* often returned plans with close to Δ time steps and never took more than $3\Delta/2$ time steps. This compares favorably with 4-*factoring direct*, which also never uses bypass nodes but which always takes essentially $3\Delta/2$ time steps.

Even though these new algorithms have inferior worst case guarantees, we find that, in practice, they usually outperform the algorithms having better theoretical guarantees. In addition, we find that for all the algorithms, the theoretical guarantees on performance are much worse than what occurs in practice. For example, even though the worst case for the number of bypass nodes needed by many of the algorithms is $n/3$, in our experiments, we almost never required more than about $n/30$ bypass nodes.

1.4 Related Work

There is significant related work for the data migration problem. As noted previously, the basic problem is equivalent to the well-known edge coloring problem. There are several other problems which are equivalent to edge coloring including h-relations [8, 18], file transfer [4], edge coloring [17] and bi-processor task scheduling on dedicated processors [15].

There are at least three decades of work on the edge-coloring problem. The minimum number of colors needed to edge-color a graph G is called the chromatic index of G, denoted by $\chi'(G)$. For a given graph G, computing $\chi'(G)$ exactly is of course NP-Complete. For simple graphs Vizing [21] gives a polytime $(\Delta+1)$-approximation algorithm. For multigraphs, the current best approximation algorithm given by Nishizeki and Kashiwagi [17] uses no more than $1.1\chi'(G)+0.8$ colors. As stated previously, the maximum degree of the graph $\Delta(G)$ is a trivial lower bound on $\chi'(G)$.

Sanders and Solis-Oba [18] study the problem of edge-coloring with indirection. The primary motivation for their work is the study of a message passing model for parallel computing (the function MPI_Alltoallv in the Message Passing Interface). This underlying real world problem of exchanging packets among processing units is called the h-relation problem. Their main result is an algorithm which splits the data objects into 5 pieces and uses indirection (but no extra bypass nodes) to do migration in $6/5(\Delta+1)$ stages if n is even and $(6/5+O(1/n))(\Delta+1)$ stages if n is odd. They assume a half-duplex model for communication when objects are split into pieces.

The work described in this paper was first published in [1, 9]. There has been significant work done subsequent to these publications. Khuller, Kim, Wan and others (see e.g. [10–14]) have extensively studied a different type of data migration problem where each data item initially belongs to a subset of disks and needs to be moved to another subset. Their plans may require creating new copies of data items and storing them on additional disks. In contrast, the work in this paper considers the problem where there is always only a single copy of each data item.

The rest of this paper is organized as follows. In Sect. 2, we describe in detail our theoretical results. Section 3 tests the algorithms from Sect. 2, and describes and tests two new migration algorithms which perform well empirically although not theoretically. Finally, Sect. 4 concludes and gives directions for future work.

2 Theoretical Results

In this section, we consider algorithms for data migration on complete graphs where all nodes have the same speeds and all edges have the same capacities. In Sect. 2.1, we consider the problem of indirect migration without space constraints and describe the algorithm 2-*factoring*. In Sect. 2.2, we consider the problem of direct migration with space constraints and the problem of indirect migration with space constraints and present the algorithms 4-*Factoring Direct* and 4-*Factoring Indirect* (Sect. 2.2.4) respectively to solve these two problems.

2.1 Migration without Space Constraints

As previously stated, an optimal direct migration takes at least χ' parallel steps where χ' is the chromatic index of the demand graph. If there are bypass nodes available, it may be possible that there is a migration plan with Δ stages. However as stated in the last section, it may take up to $n/3$ bypass nodes to allow for this.

We now present the algorithm 2-*factoring* which is a simple algorithm for indirect migration without space 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 space constraints, the simple ideas of this algorithm are important building blocks as we move on to the more complicated scenarios.

Algorithm 1 The 2-*factoring* algorithm

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 Sect. 2.1.1.)
2. Compute a 2-factor decomposition of G, viewed as undirected. (This is standard—for completeness it is described in Sect. 2.1.2.)
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 result, see [19]). However, the best multigraph edge coloring approximation algorithms achieve better bounds.

2.1.1 Obtaining a Regular Graph

Let Δ' be the desired degree, either $2\lceil \Delta/2 \rceil$ or $4\lceil \Delta/4 \rceil$ (some of our algorithms later will require degree that is a multiple of 4). We construct a directed regular multigraph as follows.

Algorithm 2 Making a directed multigraph Δ'-regular

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

Every node in the resulting graph has degree $\Delta'$.
```

2.1.2 2-Factor Decomposition

It is well known that a $2k$-regular multigraph can be factored into k 2-factors. This algorithm (see Algorithm 2.1.2) 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 3 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 node in the
   graph on both sides. Add in all directed edges going from left to right. Note that
   each edge is represented in 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.
```

2.2 Migration with Space Constraints

We now turn to the problem of migration (or edge coloring) with space constraints. For this problem, we will describe the algorithm 4-*Factoring Direct* which computes a $6\lceil \Delta/4 \rceil$ stage direct migration plan with no bypass nodes. We will also describe the algorithm 4-*Factoring Indirect*, which computes a $4\lceil \Delta/4 \rceil$ stage indirect migration with $n/3$ bypass nodes. As mentioned in the introduction, 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.

2.2.1 Indirect Migration of 4-Regular Multigraphs with Space Constraints

We begin with some intuition before presenting the migration algorithms for 4-Regular graphs. The difficulty in constructing an efficient migration plan for these graphs 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 node 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] Figure 4 illustrates the five types of vertices.

Algorithm 4 presents our construction of an indirect migration plan for 4-regular multigraphs with space constraints. The following proposition ensures that Algorithm 4 does not violate space constraints.

Algorithm 4 The bypass algorithm for 4-regular multigraphs

1. Split each hard node 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 3a shows an example graph, and Fig. 3b 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 3c 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 node (each component contains one of the split hard vertices), label the two out-edges of the split node A, label the two in-edges of the split node B, and alternately A/B label the remaining portions of the two Euler tours. (Figure 3d 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 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 (or odd path) that occurs in either the A or B graph, converting all odd cycles (or odd paths) to even-length cycles (or even-length paths). Alternately color the edges in each A cycle 1 and 2, and alternately color the edges in the B cycle 3 and 4.

Proposition 1 *Let G be a* 4*-regular multigraph. Suppose that the edges of G are A/B labeled such that each hard node has two of its incident edges labeled A, and*

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

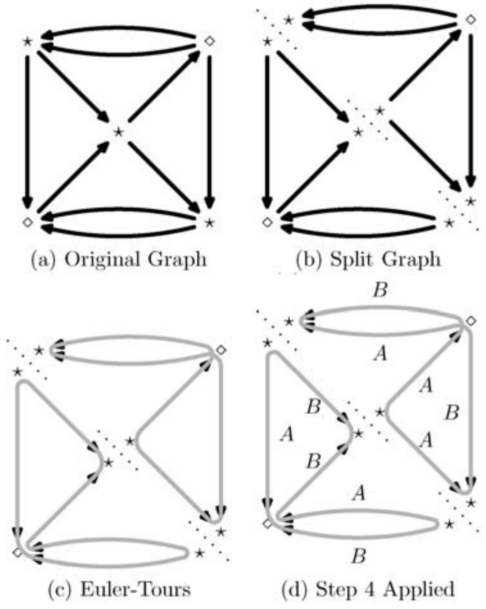


Fig. 3 Illustration of steps 1 through 4 of Algorithm 4. In step 5 of the algorithm, a single bypass node is used to get the final 4-coloring. In particular, a bypass node is added to the odd cycle induced by the A-labeled edges to ensure that it can be colored with colors 1 and 2; this same bypass node is used for the odd cycle induced by the B-labeled edges to ensure that it can be colored with colors 3 and 4

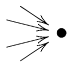
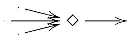
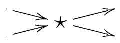
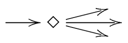
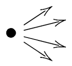
Degrees	4-in	3-in, 1-out	2-in, 2-out	1-in, 3-out	4-out
					
Initial Free Space	5	3	1	1	1
Class	easy	easy	hard	easy	easy
Parity	even	odd	even	odd	even

Fig. 4 Classification of degree 4 vertices

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.

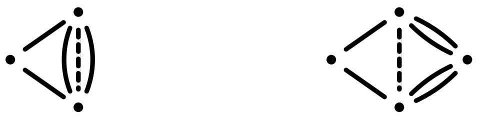
Thus, our goal is reduced to finding an A/B labeling that meets the conditions of Proposition 1, and that can be performed in as few stages as possible.

Interestingly, if there are no *odd vertices* (shown in the figures as "$\diamond$"), vertices such that the parities of their in-degree and out-degree are odd, then the problem is easy: We split each node into two with the property that each new node 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 node 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.

Procedure 1 A/B coloring odd components

There are two cases:

1. *There is only one external vertex.*
   Within this case, there are two sub-cases:
   - If there is a pair of internal vertices that are both 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.

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* node has two incident A-edges and two incident B-edges, with at least one A-labeled out-edge from each hard node. To solve the problem, we will need to bypass some of the edges in the graph.

Our algorithm starts out 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 Sect. 2.1 the incoming edge to the bypass node must be labeled A and the outgoing edge must be labeled B.

But this will not give us a good bound on bypass nodes, since there can be $2n/5$ odd components (Fig. 3d). 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 node 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 node A. In our algorithm, this happens whenever we find a pair of odd components that share representatives of the same hard node. 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 node 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 or odd paths in the A and B induced graphs. We will also need to bypass one edge in each of these odd cycles or odd paths. 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 or paths. The heart of our algorithm and analysis is judiciously choosing which edge to bypass in each odd component. With careful accounting for these bypass nodes in the analysis, we show that the total number of bypass nodes used is at most $n/3$.

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 that 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 node or both directed away from the vertex. Since adjacent odd components have been labeled (Step 4), the other two edges incident to each external node are both already labeled (one A and one B).

Figure 5 shows an example of what the resulting graph might look like. There are four unlabeled components ($C_1, \dots, C_4$). 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.

Analysis We now turn to the analysis of the algorithm. By construction, edges are A/B labeled so that the conditions of Proposition 1 are met, and hence we have:

Lemma 2 *Algorithm 4 computes a migration plan that respects space constraints.*

Proof The correctness of the algorithm is immediate—we have only to verify that space constraints are satisfied, and this is easy. Because of the way hard nodes were split, and the fact that every Euler tour is alternately A/B labeled, unless step 4 was performed, the two copies of each hard node each have one A and one B incident to them. Hence, in the first two steps (stage A), an out-edge is sent. If, on the other

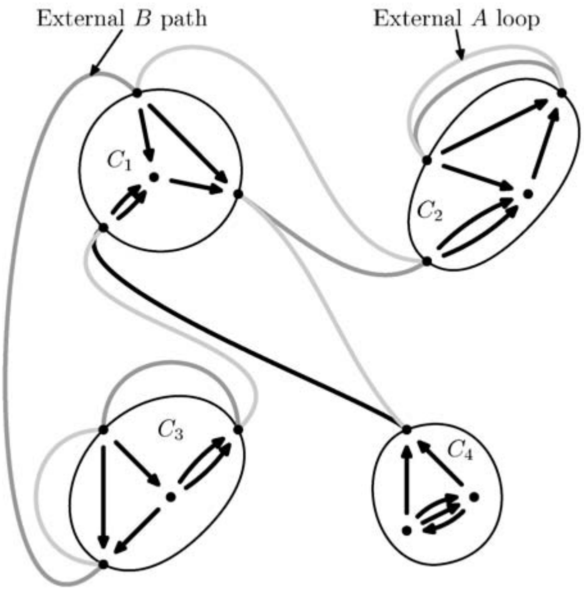


Fig. 5 An example of what the graph might look like after Step 4

hand, step 4 was performed with respect to some particular hard node shared by two odd components, then both out-edges from that hard node are labeled A, and hence are sent before any in-edge is sent. □

Our main theorem is the following:

Theorem 3 *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 and 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. Claims 1 and 2 below complete the rest of the proof.

Claim 1 *Each bypass node that is created when an odd component is* A/B *labeled can be credited with* 3 *vertices.*

Proof 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 vertex. When we pick an edge inside the component to bypass, and A/B label the component, every internal node (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 vertex either terminates at a bypass node, in which case we call it an *end path*, or it terminates at another external vertex, 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 node 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 vertex 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:

Case 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.

Case 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. □

Claim 2 *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.*

Proof 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 node cannot have an A loop. Thus, both this node 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 Fig. 6b). 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 to the Euler tour does not cause an inscribed 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 inscribed 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

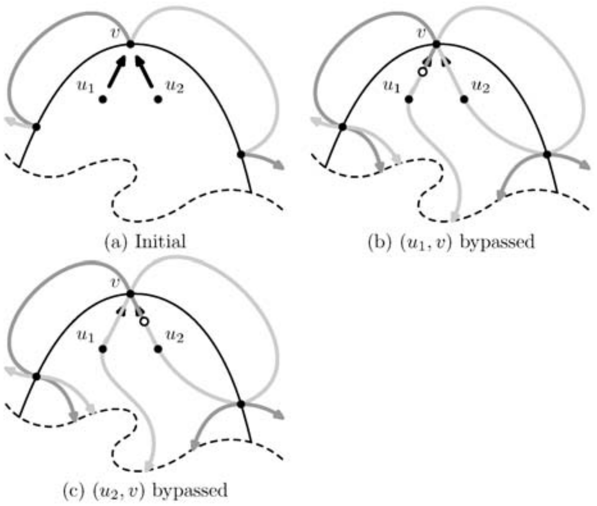


Fig. 6 Choosing which edge to bypass

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

labeled A and the edge (b, v) is labeled B (see Fig. 6c). 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. □

It is easy to see that from Claims 1 and 2, the theorem follows. □

2.2.2 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 4. Procedure 2 replaces Steps 5 and 6 of Algorithm 4, 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 (Fig. 5). We make the following additional observations about the graph:

- Each odd component contains at least one odd node. (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 2 Final steps in 4-regular migration without bypass nodes

```
5′. $A/B$ label each remaining odd component by starting with an odd node, $v$, and
    the label $B$ and following the Euler tour of the component labeling edges alter-
    nately $A$ 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 node. 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 in Sect. 2.2.2).
        iii. The remaining graph is just paths and even cycles. Color them with col-
             ors 4 and 5.
```

The algorithm can be easily seen to meet the conditions of Proposition 1. We can thus obtain the following theorem.

Theorem 4 *The above algorithm computes a proper six coloring of the graph that respects the space constraints of the hard vertices.*

Proof First, we argue that the coloring is proper. For the A induced subgraph, this is immediate. For the B induced subgraph, the argument is not as straight forward. First, the path distance between any two degree three vertices in the B subgraph is at least three. This is because they are odd nodes from different odd components that are not adjacent. This means that they do not share neighbors so they must have two vertices between them (and thus three edges). Since this is the case, Step $6'$(b)(i) does not color two edges 6 that are incident on the same node. Also because of the distance between degree three vertices in the remaining odd cycles there must be an edge that is not incident on a degree three node. As such Step $6'$(b)(i) does not color and edge 6 that is incident to a node that already has a 6 edge. It is easy to see that colors 4 and 5 are proper.

Now we argue that space constraints are respected. Since all edges in the A subgraph are sent before all edges in the B subgraph, all we have to show is that each hard node has an A colored out-edge. In Step $5'$ we colored alternating edges A and B in the Euler tour of each odd component. Since hard nodes are split into two representatives such that each representative has either two in-edges or two out-edges, the alternating Euler tour will color one in-edge and one out-edge A and likewise for B. Having consecutive Bs on an odd node in the tour does not affect the hard vertices. As such each hard node in the remaining components has an A colored out-edge. By the correctness of the previous steps of the algorithm, all vertices previously A/B colored have this property as well. □

2.2.3 The Reduction

We now show how to reduce the general migration problem with space constraints to the problem of migration with space constraints on 4-regular multigraphs. As is evident from the solution to the migration problem without space constraints, if the in-degree is the same as the out-degree for every node in the graph then our problem is easy. A bipartite matching with representatives for every node on both sides of the graph, and all edges traversing the cut in the same direction, partitions the graph into a set of 2-factors. These 2-factors have the useful property that the edges in each cycle go in the same direction. Therefore, if each node has one free space initially, then after sending one out-edge and one in-edge from each node, each node will still have at least one free space. Thus sending the edges in the 2-factor in any order maintains the invariant that there is always one free space at each node.

We note that if the in-degree of a node is different from its out-degree then, in a sense, this node is less constrained then it would be with equal in and out-degree. Either it has more that one free space initially or it will have more at the end of the migration. This observation motivates the following strategy to transforming our original graph G into a new graph G', for which we can easily find factors that allow us to preserve space constraints. To create G', we split each node v in G into two representative nodes, v_{in} and v_{out}. We assign exactly half of the edge of v to each of these nodes as follows. If v has more in-edges than out-edges then v_{in} will have only incident in-edges and v_{out} will have all of the out-edges incident to it as well as the remaining in-edges, and vice versa. Now note that if we pick two arbitrary edges, one from each of v's representatives, that we will be able to send these two edges in either order and when we are done, v will still have one free space. Consider the case where v has $d_{\text{in}} > d_{\text{out}}$. In this case, either we will pick an in-edge and an out-edge or we will pick two in-edges. In the former case, there is clearly a free space remaining. In the latter, we only have a free space remaining if there was initially at least two extra free spaces at v. Note, however, that the number of in-edges incident to v_{out} is $(d_{\text{in}} - d_{\text{out}})/2$ so there must be two extra free spaces per in-edge that is incident to v_{out}. We formalize this argument in Lemma 5.

Like in the migration without space constraints, we will be adding dummy self loops and edges to vertices to make the graph regular. These self loops and dummy edges must be treated specially when dividing v's edges between v_{in} and v_{out}. Dummy self loops must always go from v_{out} to v_{in}. Dummy edges, regardless of their orientation, must be placed incident to the node with the smaller number of edges. For example, if $d_{\text{in}} > d_{\text{out}}$ then the dummy edge goes to v_{out}. Recall that there is at most one dummy edge but possibly many dummy self loops.

What we are thus able to do is the graph into 4-factors such that after each 4-factor is sent, there is always at least one free space at each node. We do this by (1) creating a new graph G' by splitting each vertex v in G into v_{in} and v_{out}, and dividing up v's edges between v_{in} and v_{out} as described above; (2) computing a 2-factoring of G' (with G' viewed as undirected); and (3) for each original node v in G, merging v_{in} and v_{out} to transform each 2-factor of G' into a 4-factor of G. Algorithm 5 gives the precise details and the correctness is formalized in the following lemma.

Lemma 5 *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 1: v_{out} has only incident out-edges.
Then since one of the edges chosen in any two-factor is an out-edge there will be at least one out-edge sent for every in-edge sent in the two-factor (in-edges might be chosen from v_{in}). Thus, the free space after the two edges are sent is at least what it was before.

Case 2: 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 5 The reduction to 4-regular graphs

```
1. Make $G$ regular with degree a multiple of four ($4k$) using the procedure in Sect. 2.1.1.
2. Split each node $v$ into $v_{\text{in}}$ and $v_{\text{out}}$ assigning $v$'s edges to either $v_{\text{in}}$ or $v_{\text{out}}$ to get $G'$:
   (a) Assign any dummy self loops, $(v, v)$, to both $v_{\text{in}}$ and $v_{\text{out}}$ as $(v_{\text{out}}, v_{\text{in}})$.
   (b) Assign the remaining in-edges to $v_{\text{in}}$ and the remaining out-edges to $v_{\text{out}}$ (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 $v_{\text{in}}$ and $v_{\text{out}}$ 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 node representatives back together. That is, $v_{\text{in}}$ and $v_{\text{out}}$ 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 6 *Algorithm 5 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.*

2.2.4 *The Algorithms* 4-Factoring Direct *and* 4-Factoring Indirect

We can now define the algorithms 4-*Factoring Direct* and 4-*Factoring Indirect*.

The algorithm 4-*Factoring Direct* is defined as follows: Use Algorithm 5 to reduce the graph into 4-factors and then use the algorithm from Theorem 4 (in Sect. 2.2.2) to migrate each of these in 6 steps.

The algorithm 4-*Factoring Indirect* is defined as follows: Use Algorithm 5 to reduce the graph into 4-factors and then use the algorithm 4 to migrate each of these in 4 steps using no more than $n/3$ bypass nodes.

Combining Theorem 6 with Theorems 4 and 3 gives us the following corollaries:

Corollary 7 4-Factoring Direct *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.*

Corollary 8 4-Factoring Indirect *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.*

3 Empirical Results

The primary focus of this section is on the empirical evaluation of the algorithms from the last section along with two new heuristic algorithms. The section is organized as follows. In Sect. 3.1, we describe the new algorithms we have evaluated for indirect migration without space constraints. In Sect. 3.2, we describe the new algorithms we have evaluated for migration with space constraints. Section 3.3 describes how we create the demand graphs on which we test the migration algorithms while Sects. 3.4 and 3.5 describe our experimental results. Section 3.6 gives an analysis and discussion of these results.

3.1 Max-Degree Matching for Migration without Space Constraints

We present an algorithm, *Max-Degree-Matching* (see Algorithm 6), which uses at most $2n/3$ bypass nodes and always attains an optimal Δ step migration plan without space constraints. *Max-Degree-Matching* works by sending, in each stage, one object from each node in the demand graph that has maximum degree. To do this, we first find a matching which matches all maximum-degree vertices with no out-edges. Next, we match each unmatched maximum-degree node up with a bypass node. Finally we use the general matching algorithm [16] to expand this matching to a maximal matching and then send every edge in this new expanded matching.

We now give a high level description of why *Max-Degree-Matching* takes Δ steps and uses $2n/3$ bypass nodes; the formal proof of this fact is given in Theorem 9 below. To complete the migration in Δ steps, it is enough to make sure that at each stage every node in the graph that has maximum degree either sends out an object or receives an object. Our solution is optimal in that it achieves a Δ step migration plan. At a high level the algorithm constructs and solves a bipartite matching problem on a bipartite graph between the maximum degree vertices (on the left) and their neighbors (on the right). The solution to the matching problem induces paths and cycles in the demand graph with the property that every maximum degree node is in a path or cycle. Sending every other edge in the paths and cycles and bypassing one edge in odd cycles will guarantee that each maximum degree node has its degree reduced by one in every stage. To reduce the number of bypass nodes used, we make the following observation: any maximum degree node that has a bypass node waiting to send it an object can receive the object in the current round and so we can leave these maximum degree vertices out of the left hand side of the bipartite matching problem and just assume that they are receiving objects from the bypass nodes.

Algorithm 6 *Max-Degree-Matching* (*demand graph* G)

```
1. Set up a bipartite matching problem as follows: the left hand side of the graph is all maximum degree vertices not adjacent to degree one vertices in $G$, the right hand side is all their neighbors in $G$, and the edges are all edges between maximum degree vertices and their neighbors in $G$.
2. Find the maximum bipartite matching. The solution induces cycles and paths in the demand graph. All cycles contain only maximum degree vertices, all paths have one endpoint that is not a maximum degree node.
3. Mark every other edge in the cycles and paths. For odd length cycles, one node will be left with no marked edges. Make sure that this is a node with an outgoing edge (and thus can be bypassed if needed). Each node has at most one edge marked. Mark every edge between a maximum degree node and a degree one node.
4. Let $V'$ be the set of vertices incident to a marked edge. Compute a maximum matching in $G$ that matches all vertices in $V'$. (This can be done by seeding the general matching algorithm [16] with the matching that consists of marked edges.) Define $S$ to be all edges in the matching.
5. For each edge node $u$ of maximum degree with no incident edge in $S$, let $(u, v)$ be some out-edge from $u$. Add $(u, b)$ to $S$, where $b$ is an unused bypass node, and add $(b, v)$ to the demand graph $G$.
6. Schedule all edges in $S$ to be sent in the next stage and remove these edges from the demand graph.
7. If there are still edges in the demand graph, go back to step 1.
```

Theorem 9 Max-Degree-Matching *computes a correct* Δ*-stage migration plan using at most* $2n/3$ *bypass nodes.*

Proof First we show that the algorithm uses no more than Δ stages. Hall's theorem can be used to show that the matching problem constructed in step 1 of the algorithm always has a solution in which all the maximum degree vertices are matched. Thus at each stage the degree of each maximum degree node is decreased by one. After Δ stages we have no edges left and are done. The constraint that each node has only one edge sent or received per stage is maintained because we only send the edges in a general matching solution and edges out of vertices not matched by the general matching.

Next we show that the algorithm uses no more than $2n/3$ bypass nodes. We first show that there are no more than $n/2$ active bypass nodes after each stage. Then we show that in the turnover as some bypass nodes are used and some are created within a stage, that the number of bypass nodes does not exceed $2n/3$. Let k be the number of paths and cycles induced in G by the matching. After this stage each component can only have one bypass node associated with it. For cycles this is because originally the cycle had no bypass nodes, but if it is an odd cycle then after this stage it has one bypass node. For paths this is because any odd length path could end in a node with a bypass node. If this is the case, since the path is odd length, the bypass node does not get used. Each of these path or cycle has at least two vertices, so $k \leq n/2$. Any bypass node that is not adjacent to a path or cycle will be unused in this stage. Thus, there are at most $n/2$ bypass nodes after any stage. Note that this says nothing about the number of different bypass nodes used in a stage. For example, the bypass nodes at the end of the stage might not be the same as the ones at the start of the stage. If one node is used in a stage and one node is created, two nodes must exist because the node cannot both send (to be used) and receive (to be created) in the same stage.

To get the bound on the number of nodes in use during a given stage of less than $2n/3$, we note that we need to bound the sum of the number of bypass nodes enlisted in this stage and the number that were left from the previous stage. Assume the number of bypass nodes left over from the previous stage is n_b, and that the number of maximum degree vertices without bypass nodes is n_m. The number of bypass nodes created in this stage is bounded by $n_m/3$ because bypass nodes are only created from odd length cycles in the matching. These cycles must have at least 3 vertices in them and only vertices not adjacent to bypass nodes (ones put on the left hand side of the matching) can be in cycles in the matching. Here, n_m is at most $n - n_b$ because each bypass node is adjacent to exactly one maximum degree node. So to maximize the number of bypass nodes we must maximize: $n_m/3 + n_b \le (n - n_b)/3 + n_b = n/3 + 2n_b/3$. We proved above that $n_b \le n/2$ so the most bypass nodes that can be in use during a given stage is $2n/3$. Bypass nodes can be recycled so that only $2n/3$ are required for the entire bypass algorithm. □

In what follows, we compare *Max-Degree-Matching* with 2*-factoring*, which also computes an indirect migration plan without space constraints. We have shown in the last section that 2*-factoring* takes $2\lceil \Delta/2 \rceil$ time steps while using no more than $n/3$ bypass nodes.

We note that in a particular stage of 2*-factoring* as described in the last section, there may be some nodes which only have dummy edges incident to them. A heuristic for reducing the number of bypass nodes needed is to use these nodes as bypass nodes when available to decrease the need for "external" bypass nodes. Our implementation of 2*-factoring* uses this heuristic.

3.2 Greedy Matching for Migration with Space Constraints

The *Greedy Matching* algorithm (Algorithm 7) is a straightforward direct migration algorithm which obeys space constraints. This algorithm eventually sends all of the objects (see Lemma 10) but the worst case number of stages is unknown.

Algorithm 7 *Greedy Matching*

```
1. Let $G'$ be the graph induced by the sendable edges in the demand graph. An edge
   is sendable if there is free space at its destination.
2. Compute a maximum general matching on $G'$.
3. Schedule all edges in matching to be sent in this stage.
4. Remove these edges from the demand graph.
5. Repeat until the demand graph has no more edges.
```

For a node v, let $d_{\text{in}}(v)$ be the in-degree of the node and let $d_{\text{out}}(v)$ be the out degree.

Lemma 10 *Given initial free space of* $f_i \geq 1 + \max(0, d_{\text{in}}(v_i) - d_{\text{out}}(v_i))$, *at any stage of a direct migration* (*after sending any number of objects*) *at least one unsent object is sendable.*

Proof At any stage in the migration, the graph G only has edges left in it for objects that have not yet been sent to their destination. Since we assume that $f_i \geq 1 + \max(0, d_{\text{in}}(v_i) - d_{\text{out}}(v_i))$ for all i initially, all we have to show now is that there exists a node $v_* \in V$ that has $d_{\text{in}}(v_*) - d_{\text{out}}(v_*) \geq 0$ with $d_{\text{in}}(v_*) > 0$. This would imply that v_* has free space and an incoming edge and hence the object corresponding to that incoming edge is sendable.

Let V' be the subset of V that has only vertices that have sendable edges (that is, $d_{\text{in}} + d_{\text{out}} > 0$). Then $\sum_{v \in V'} (d_{\text{in}}(v) - d_{\text{out}}(v)) = 0$. This is because each edge contributes to exactly one node's in-degree and one node's out-degree. Since the average over all $v \in V'$ of $(d_{\text{in}}(v) - d_{\text{out}}(v)) = 0$, there must be a node in V', which we call v_*, with $d_{\text{in}}(v_*) - d_{\text{out}}(v_*) \geq 0$. Since we also have $d_{\text{out}}(v_*) \geq 0$, it must be that $d_{\text{in}}(v_*) > 0$. □

In what follows, we compare *Greedy-Matching* with the two provably good algorithms for migration with space constraints from the last section: 4-*factoring direct* and 4-*factoring indirect*. We have shown that 4-*factoring direct* finds a $6\lceil \Delta/4 \rceil$ stage migration without bypass nodes and that *4-factoring indirect* finds a $4\lceil \Delta/4 \rceil$ stage migration plan using at most $n/3$ bypass nodes.

In our implementation of 4-*factoring indirect*, we again use the heuristic of using nodes with only dummy edges in a particular stage as bypass nodes for that stage.

3.3 Experimental Setup

Table 1 summarizes the theoretical results known for each algorithm on which we have run experiments.[4] We tested all of these algorithms on four types of multigraphs:[5]

1. *Load-Balancing Graphs*. These graphs represent real-world migrations. A detailed description of how they were created is given in the next subsection.
2. *General Graphs* (n, m). A graph in this class contains n nodes and m edges. The edges are chosen uniformly at random from among all possible edges disallowing self-loops (but allowing parallel edges).
3. *Regular Graphs* (n, d). These graphs represent real-world migrations especially for load balancing. Graphs in this class are chosen uniformly at random from among all regular graphs with n nodes, where each node has total degree d (where d is even). We generated these graphs by taking the edge-union of $d/2$ randomly generated 2-regular graphs over n vertices.

[4]For each algorithm, time to find a migration plan is negligible compared to time to implement the plan.

[5]Java code implementing these algorithms along with input files for all the graphs tested is available at www.cs.unm.edu/homes/saia/migration.

4. *Zipf Graphs* $(n, d_{\min})$. These graphs are chosen uniformly at random from all graphs with n nodes and minimum degree $d_{\min}$ that have Zipf degree distribution: i.e., the number of nodes of degree d is proportional to $1/d$. Our technique for creating random Zipf graphs is given in detail in Sect. 3.3.2.

3.3.1 Creation of Load-balancing Graphs

A migration problem can be generated from any two configurations of a set of objects on a set of nodes in a network. To generate the *Load-Balancing* graphs, we used two different methods of generating sequences of configurations of objects which might occur in a real world system. For each sequence of say l configurations, $C_1, \ldots, C_l$, for each i, $1 \le i \le l-1$, we generate a demand graph using C_i as the initial configuration and C_{i+1} as the final.

For the first method, we used the Hippodrome system on two variants of a retail data warehousing workload [2]. Hippodrome adapts a storage system to support a given workload by generating a series of object configurations, and possibly increasing the node count. Each configuration is better at balancing the workload of user queries for data objects across the nodes in the network than the previous configuration. We ran the Hippodrome loop for 8 iterations (enough to stabilize the node count) and so got two sequences of 8 configurations. For the second method, we took the 17 queries to a relational database in the TPC-D benchmark [20] and for each query generated a configuration of objects to devices which balanced the load across the devices effectively. This method gives us a sequence of 17 configurations.

Different devices have different performance properties and hence induce different configurations. When generating our configurations, we assumed all nodes in the network were the same type of storage device. For both methods, we generated configurations based on two different types of devices. Thus for the Hippodrome method, we generated 4 sequences of 8 configurations (7 graphs) and for the TPC-D method, we generated 2 sequences of 17 configurations (16 graphs) for a total of 60 demand graphs.

3.3.2 Creation of Zipf Graphs

The Zipf graphs for minimum degree $d_{\min}$ are generated as follows: we first create sets $S_1, \ldots, S_k$ each containing $k!$ vertices and let S be the union of all the sets. We then find d_{min} random perfect matchings from S to S so that now every node in S has degree $d_{\min}$. We next, for all i, $1 \le i \le k$, partition S_i into $k!/i$ subsets each with i nodes and then merge each of these subsets into one node. By doing this, we get $k!/i$ new vertices each with degree $d_{\min} \times i$. We let these new vertices be the vertices of the Zipf graph.

The total number of vertices in this Zipf graph is $k!H_k$ where H_k is the k-th Harmonic number. We note that the maximum degree of the graph is $d_{\min} \times k$ while the minimum degree is 1. We further note that for all i, $1 \le i \le k$, the number of nodes with degree i is $d_{\min} \times k!/i$.

3.4 Results on the *Load-Balancing Graphs*

3.4.1 Graph Characteristics

In this section, we describe the *load-balancing* graphs used in our experiments. We refer to the sets of graphs generated by Hippodrome on the first and second device type as the first and second set respectively and the sets of graphs generated with the TPC-D method for the first and second device types as the third and fourth sets.

The number of nodes in each graph is less than 50 for the graphs in all sets except the third in which most graphs have around 300 nodes. The edge degree for each graph varies from about 5 for most graphs in the third set to around 65 for most graphs in the fourth set. The Δ value for each graph varies from about 15 to about 140, with almost all graphs in the fourth set having degree around 140.

3.4.2 Performance

Figure 7 shows the performance of the algorithms on the load-balancing graphs in terms of the number of bypass nodes used and the time steps taken. The x-axis in each plot gives the index of the graph which is consistent across both plots. The indices of the graphs are clustered according to the sets the graphs are from with the first, second, third and fourth sets appearing left to right, separated by solid lines.

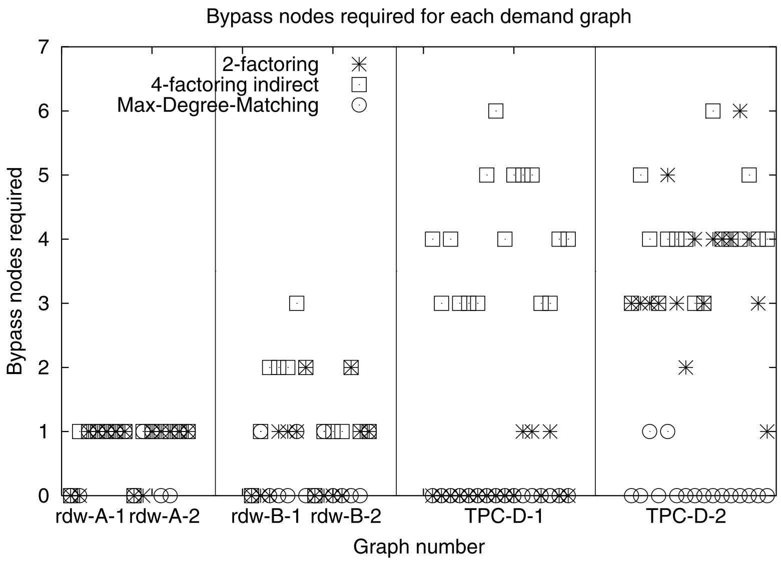
The first plot shows the number of bypass nodes used by 2-*factoring*, 4-*factoring indirect* and *Max-Degree-Matching*. We see that *Max-Degree-Matching* uses 0 bypass nodes on most of the graphs and never uses more than 1. The number of bypass nodes used by 2-*factoring* and 4-*factoring indirect* are always between 0 and 6, even for the graphs with about 300 nodes. The second plot shows the number of stages required divided by Δ for *Greedy-Matching*. Recall that this ratio for 2-*factoring*, *Max-Degree-Matching* and 4-*factoring indirect* is essentially 1 while the ratio for 4-*factoring direct* is essentially 1.5. In the graphs in the second and third set, *Greedy-Matching* almost always has a ratio near 1. However in the first set, *Greedy-Matching* has a ratio exceeding 1.2 on several of the graphs and a ratio of more than 1.4 on one of them. In all cases, *Greedy-Matching* has a ratio less than 4-*factoring direct*.

We note the following important points: (1) On all of the graphs, the number of bypass nodes needed is less than 6 while the theoretical upper bounds are significantly higher. In fact, *Max-Degree-Matching* used *no bypass nodes* for the majority of the graphs; and (2) *Greedy-Matching* always takes fewer stages than 4-*factoring direct*.

3.5 Results on General, Regular and Zipf Graphs

3.5.1 Bypass Nodes Needed

For *General*, *Regular* and *Zipf Graphs*, for each set of graph parameters tested, we generated 30 random graphs and took the average performance of each algorithm over all 30 graphs. For this reason, the data points in the plots are not at integral values. *Greedy-Matching* never uses any bypass nodes so in this section, we include results only for *Max-Degree-Matching*, 4-*factoring indirect* and 2-*factoring*.



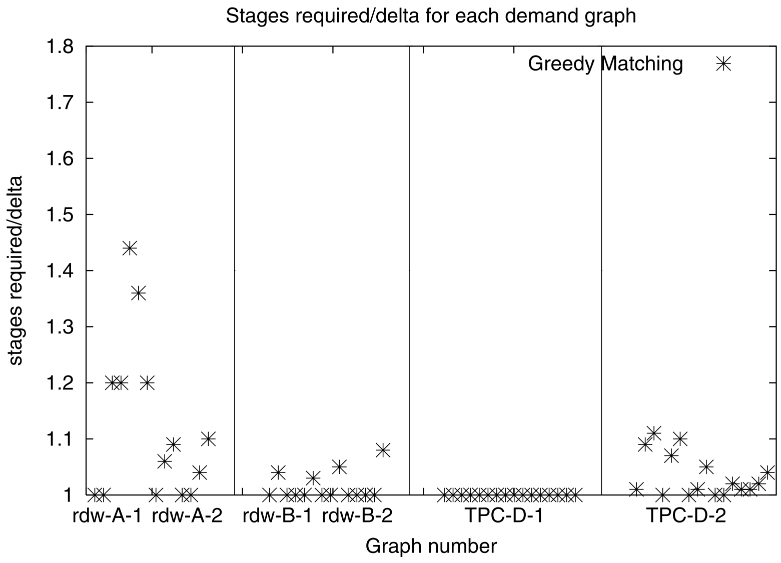


Fig. 7 Bypass nodes and time steps needed for the algorithms for the three workloads (rdw-A, rdw-B, TPC-D), and the two arrays ($-1, -2$). The *top plot* gives the number of bypass nodes required for the algorithms 2-*factoring*, 4-*factoring indirect* and *Max-Degree-Matching* on each of the *Load-Balancing Graphs*. The *bottom plot* gives the ratio of time steps required to Δ for *Greedy-Matching* on each of the *Load-Balancing Graphs*. The three *solid lines* in both plots divide the four sets of *Load-Balancing Graphs*

Varying Number of Nodes The three plots in the left column of Fig. 8 give results for random graphs where the edge degree is fixed and the number of nodes varies. The first plot in this column shows the number of bypass nodes used for *General Graphs* with edge degree fixed at 10 as the number of nodes increases from 100 to

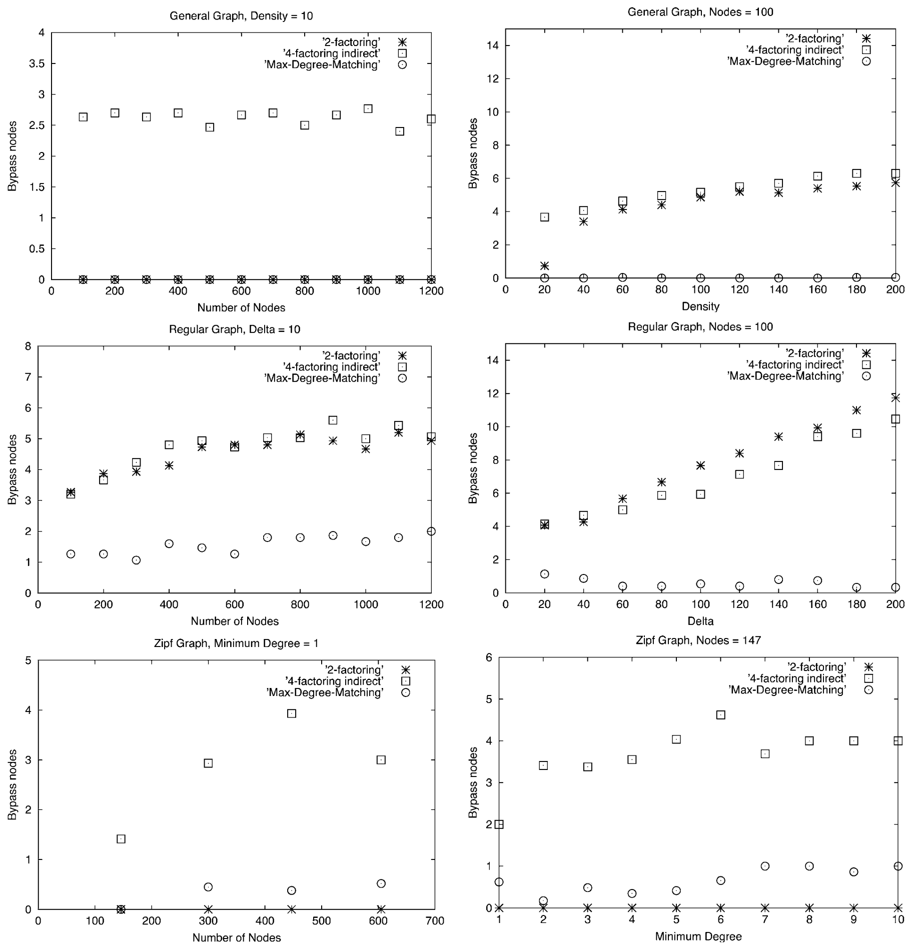


Fig. 8 Six plots giving the number of bypass nodes needed for 2-*factoring*, 4-*factoring direct* and *Max-Degree-Matching* for the *General*, *Regular* and *Zipf Graphs*. The three plots in the *left column* give the number of bypass nodes needed as the number of *nodes* in the random graphs increase. The three plots in the *right column* give the number of bypass nodes needed as the *degree* of the random graphs increase. The plots in the *first row* are for *General Graphs*, plots in the *second row* are for *Regular Graphs* and plots in the *third row* are for *Zipf Graphs*

1200. We see that *Max-Degree-Matching* and 2-*factoring* consistently use no bypass nodes. 4-*factoring indirect* uses between 2 and 3 bypass nodes and surprisingly this number does not increase as the number of nodes in the graph increases.

The second plot shows the number of bypass nodes used for *Regular Graphs* with $\Delta = 10$ as the number of nodes increases from 100 to 1200. We see that the number of bypass nodes needed by *Max-Degree-Matching* stays relatively constant at 1 as the number of nodes increases. The number of bypass nodes used by 2-*factoring* and 4-*factoring indirect* are similar, starting at 3 and growing slowly to 6, approximately linearly with a slope of $1/900$.

The third plot shows the number of bypass nodes used on *Zipf Graphs* with minimum degree 1 as the number of nodes increases. In this graph, 2*-factoring* is consistently at 0, *Max-Degree-Matching* varies between $1/4$ and $1/2$ and 4*-factoring indirect* varies between 1 and 4.

Varying Edge Density The three plots in the right column of Fig. 8 show the number of bypass nodes used for graphs with a fixed number of nodes as the edge degree varies. The first plot in the column shows the number of bypass nodes used on *General Graphs*, when the number of nodes is fixed at 100, and edge degree is varied from 20 to 200. We see that the number of bypass nodes used by *Max-Degree-Matching* is always 0. The number of bypass nodes used by 2 and 4*-factoring indirect* increases slowly, approximately linearly with a slope of about $1/60$. Specifically, the number used by 2-factoring increases from $1/2$ to 6 while the number used by 4*-factoring indirect* increases from 4 to 6.

The second plot shows the number of bypass nodes used on *Regular Graphs*, when the number of nodes is fixed at 100 and Δ is varied from 20 to 200. The number of bypass nodes used by *Max-Degree-Matching* stays relatively flat varying slightly between $1/2$ and 1. The number of bypass nodes used by 2*-factoring* and 4*-factoring indirect* increases near linearly with a larger slope of $1/30$, increasing from 4 to 12 for 2*-factoring* and from 4 to 10 for 4*-factoring indirect*.

The third plot shows the number of bypass nodes used on *Zipf Graphs*, when the number of nodes is fixed at 146 and the minimum degree is varied from 1 to 10. 2*-factoring* here again always uses 0 bypass nodes. The *Max-Degree-Matching* curve again stays relatively flat varying between $1/4$ and 1. 4*-factoring indirect* varies slightly, from 2 to 4, again near linearly with a slope of $1/5$.

We suspect that our heuristic of using nodes with only dummy edges as bypass nodes in a stage helps 2*-factoring* significantly on *Zipf Graphs* since there are so many nodes with small degree and hence many dummy self-loops.

3.5.2 Time Steps Needed

For *General* and *Regular Graphs*, the migration plans *Greedy-Matching* found never took more than $\Delta + 1$ time steps. Since the other algorithms we tested are guaranteed to have plans taking less than $\Delta + 3$, we present no plots of the number of time steps required for these algorithms on *General* and *Regular Graphs*.

As shown in Fig. 9, the number of stages used by *Greedy-Matching* for *Zipf Graphs* is significantly worse than for the other types of random graphs. We note however that it always performs better than 4*-factoring direct*. The first plot shows that the number of extra stages used by *Greedy-Matching* for *Zipf Graphs* with minimum degree 1 varies from 2 to 4 as the number of nodes varies from 100 to 800. The second plot shows that the number of extra stages used by *Greedy-Matching* for *Zipf Graphs* with 146 nodes varies from 1 to 11 as the minimum degree of the graphs varies from 1 to 10. High degree Zipf graphs are the one weakness we found for *Greedy-Matching*.

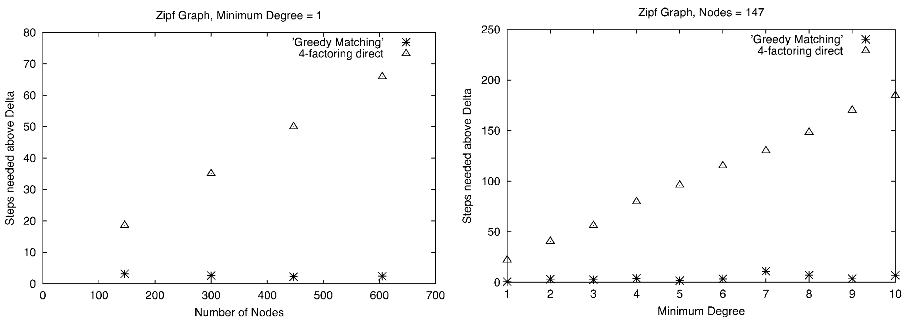


Fig. 9 Number of steps above *Delta* needed for *Greedy-Matching* on *Zipf Graphs*

3.6 Analysis

Our major empirical conclusions for the graphs tested are:

- *Max-Degree-Matching* almost always uses less bypass nodes than 2-*factoring*.
- *Greedy-Matching* always takes less time steps than 4-*factoring direct*.
- For all algorithms using indirection, the number of bypass nodes required is almost always no more than $n/30$.

For migration without space constraints, *Max-Degree-Matching* performs well in practice, often using significantly fewer bypass nodes than 2-*factoring*. Its good performance and good theoretical properties make it an attractive choice for real world migration problems without space constraints.

For migration with space constraints, *Greedy-Matching* always outperforms 4-*factoring direct*. It also frequently finds migration plans within some small constant of Δ. However there are many graphs for which it takes much more than Δ time steps and for this reason we recommend 4-*factoring indirect* when there are bypass nodes available.

3.6.1 Theory Versus Practice

In our experiments, we have found that not only are the number of bypass nodes required for the types of graphs we tested much less than the theoretical bounds suggest but that in addition, the *rate* of growth in the number of bypass nodes versus the number of demand graph nodes is much less than the theoretical bounds. The worst case bounds are that $n/3$ bypass nodes are required for 2-*factoring* and 4-*factoring indirect* and $2n/3$ for *Max-Degree-Matching* but in most graphs, for all the algorithms, we never required more than about $n/30$ bypass nodes.

The only exception to this trend is regular graphs with high degree for which 2-*factoring* and 4-*factoring indirect* required up to $n/10$ bypass nodes. A surprising result for these graphs was the fact that *Max-Degree-Matching* performed much better than 2-*factoring* and 4-*factoring indirect* despite its worse theoretical bound.

4 Conclusion

We have presented several algorithms for the data migration problem and have theoretically and empirically evaluated their performance. The metrics we have used are: (1) the number of time steps required to perform the migration, and (2) the number of bypass nodes used as intermediate storage devices. We have presented algorithms that provably perform well with respect to these two metrics. Moreover, we have empirically tested our algorithms and have shown that there are some that perform quite well in practice despite not having optimal theoretical guarantees. We conclude that many of the algorithms described in this paper are both practical and effective for data migration. Numerous open problems remain including the following:

What is the trade-off between the number of bypass nodes available and the number of stages required? In particular, we want a good approximation algorithm for migration with indirection when no external bypass nodes are available. To the best of our knowledge, no algorithm with an approximation ratio better than $3/2$ for this problem is known at this time.

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

Finally, can we design data migration algorithms that work well for sparse, heterogeneous networks? Real world devices have different I/O speeds. For example, one device might be able to send or receive twice as many objects per stage as another device. We want good approximation algorithms for migration with different device speeds. Also in some important cases, a complete graph is a poor approximation to the network topology. For example, a wide area network typically has a sparse topology and data center networks are more closely related to a tree than to a complete graph. We want good approximation algorithms for commonly occurring topologies (such as trees) and in general for arbitrary topologies.

References

1. Anderson, E., Hall, J., Hartline, J., Hobbes, M., Karlin, A., Saia, J., Swaminithan, R., Wilkes, J.: An experimental study of data migration algorithms. In: Proceedings of the Workshop on Algorithm Engineering (WAE), pp. 145–158 (2001)
2. Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., Veitch, A.: Hippodrome: running circles around storage administration. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pp. 175–188 (2002)
3. Borowsky, E., Golding, R., Merchant, A., Schreier, L., Shriver, E., Spasojevic, M., Wilkes, J.: Using attribute-managed storage to achieve QoS. In: Proceedings of the International Workshop on Quality of Service (IWQoS), pp. 199–202. Columbia University, New York, June 1997
4. Coffman, E., Garey, M., Johnson, D., Lapaugh, A.: Scheduling file transfers. SIAM J. Comput. **14**(3), 744–780 (1985)
5. Gavish, B., Sheng, O.: Dynamic file migration in distributed computer systems. Commun. ACM **33**(2), 177–189 (1990)
6. Golubchik, I., Khuller, S., Khanna, S., Thurimella, R., Zhu, A.: Approximation algorithms for data placement on parallel disks. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 223–232 (2000)

7. Golubchik, L., Khuller, S., Kim, Y., Shargorodskaya, S., Wan, Y.: Data migration on parallel disks. Algorithmica **45**(1), 137–158 (2006)
8. Grammatikakis, M., Hsu, D., Kraetzl, M., Sibeyn, J.: Packet routing in fixed-connection networks: A survey. J. Parallel Distrib. Process. **54**(2), 77–132 (1998)
9. Hall, J., Hartline, J., Karlin, A., Saia, J., Wilkes, J.: On algorithms for efficient data migration. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 620–629 (2001)
10. Kashyap, S., Khuller, S., Wan, Y., Golubehik, L.: Fast reconfiguration of data placement in parallel disks. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 95–107 (2006)
11. Khuller, S., Kim, Y., Wan, Y.: Algorithms for data migration with cloning. SIAM J. Comput. **33**(2), 448–461 (2004)
12. Khuller, S., Kim, Y., Malekian, A.: Improved algorithms for data migration. In: Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX), pp. 164–175 (2006)
13. Kim, Y.: Data migration to minimize the average completion time. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 97–98 (2003)
14. Kim, Y.: Data migration to minimize the average completion time. J. Algorithms **55**(1), 42–57 (2005)
15. Kubale, M.: Preemptive versus nonpreemptive scheduling of biprocessor tasks on dedicated processors. Eur. J. Oper. Res. **94**(2), 242–251 (1996)
16. Micali, S., Vazirani, V.: An $o(\sqrt{|V|}|e|)$ algorithm for finding a maximum matching in general graphs. In: Proceedings of the Symposium on Foundations of Computer Science (FOCS), pp. 17–27 (1980)
17. Nishizeki, T., Kashiwagi, K.: On the 1.1 edge-coloring of multigraphs. SIAM J. Discrete Math. **3**(3), 391–410 (1990)
18. Sanders, P., Solis-Oba, R.: How helpers hasten h-relations. In: Proceedings of the European Symposium on Algorithms (ESA), pp. 392–402 (2000)
19. Shannon, C.: A theorem on colouring lines of a network. J. Math. Phys. **28**, 148–151 (1949)
20. Transaction Processing Performance Council. TPC Benchmark D (Decision Support) Standard Specification Revision 2.1. Transaction Processing Performance Council (1996)
21. Vizing, V.: On an estimate of the chromatic class of a p-graph. Discrete Anal. **3**, 23–30 (1964) (in Russian)
22. Wolf, J.: 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, pp. 1–10 (1989)