An Experimental Study of Data Migration Algorithms

Eric Anderson[1], Joe Hall[2], Jason Hartline[2], Michael Hobbs[1], Anna R. Karlin[2], Jared Saia[2], Ram Swaminathan[1], and John Wilkes[1]

[1] Storage Systems Program, Hewlett-Packard Laboratories
Palo Alto, CA 94304
{anderse,mjhobbs,swaram,wilkes}@hpl.hp.com

[2] Department of Computer Science and Engineering, University of Washington
Seattle, WA 98195
{jkh,hartline,karlin,saia}@cs.washington.edu

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 of data. In this paper, we consider the case where the objects are fixed-size and the network is complete. We introduce two new data migration algorithms, one of which has provably good bounds. We empirically compare the performance of these new algorithms against similar algorithms from Hall et al. [7] which have better theoretical guarantees and find that in almost all cases, the new algorithms perform better. We also find that both the new algorithms and the ones from Hall et al. perform much better in practice than the theoretical bounds suggest.

1 Introduction

The performance of modern day large-scale storage systems (such as disk farms) can be improved by balancing the load across devices. Unfortunately, the optimal data layout is likely to change over time because of workload changes, device additions, or device failures. Thus, it may be desirable to periodically compute a new assignment of data to devices [3,5,6,11], 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 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 offline 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 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 and the size of each object.) Our goal is to find a *migration plan* that uses the existing connections between storage devices to move the data from the initial configuration to the final configuration in as few time steps as possible. We assume that the objects are all the same size, for example, fragmenting them into fixed sized extents. We also assume that any pair of devices can send to each other without impacting other pairs, i.e., we assume that the underlying network is complete. 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.

We consider two interesting variants of the fixed size migration problem. First, we consider the effect of space constraints. For *migration without space constraints* we assume that an unlimited amount of storage for data objects is available at each device in the network. At the other extreme, in *migration with space constraints*, we assume that each device has the minimum amount of extra space possible – only enough to hold one more object than the maximum of the number of objects stored on that device in the initial configuration or in the final configuration. At the end of each step of a migration plan, we ensure that the number of objects stored at a node is no more than the assumed total space at that node.

Second, we compare algorithms that migrate data directly from source to destination with those that allow indirect migration of data, through intermediate nodes. We call these intermediate nodes *bypass nodes*. Frequently, there are devices with no objects to send or receive, and we can often come up with a significantly faster migration plan if we use these devices as bypass nodes.

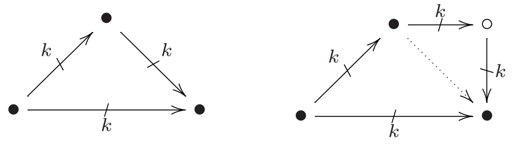
We can model the input to our problem as a directed *multigraph*[1] $G = (V, E)$ without self-loops that we call the *demand graph*. Each of the vertices in the demand graph corresponds to a storage device, and each of the directed edges (u, v) represents an object that must be moved from storage device u (in the initial configuration) to storage device v (in the final configuration). The output of our algorithms will be a positive integer label for each edge which indicates the stage at which that edge is moved. I/O constraints imply that no vertex can have two edges with the same integer label incident to it.

The labels on the edges may be viewed as colors in an edge coloring of the graph. Thus, direct migration with no space constraints is equivalent to the well known multigraph edge-coloring problem. The minimum number of colors needed to edge-color a graph is called the *chromatic index* or χ' of the graph. Computing

[1] A multigraph is a graph which can have multiple edges between any two nodes.

χ' is NP-complete but there is a $1.1\chi'(G) + .8$ approximation algorithm [9]. Δ, the maximum degree of the graph, is a trivial lower bound on the number of colors needed. It is also well known that 1.5Δ colors always suffice and that there are graphs requiring this many colors.

For indirect migration, we want to get as close as possible to the theoretically minimum-length migration plan of Δ while minimizing the number of bypass nodes needed. The following example shows how a bypass node can be used to reduce the number of stages. 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 $\circ$.)



It is easy to see that $n/3$ is a *worst case lower bound* on the number of bypass nodes needed to perform a migration in Δ stages – consider the example demand graph consisting of k disjoint copies of the 3-cycle ($n = 3k$).

In this paper, we introduce two new migration algorithms. The primary focus of our work is on the empirical evaluation of these algorithms, and the migration algorithms introduced in [7].

For the case where there are no space constraints, we evaluate two algorithms which use indirection. We introduce the first of these in this paper; it 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* [7] which finds a migration plan taking $2\lceil\Delta/2\rceil$ steps by using no more than $n/3$ bypass nodes [7]. 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 very close to Δ time steps and never took more than $3\Delta/2$ time steps. This compares favorably with *4-factoring direct* [7] which also never uses bypass nodes but which always takes essentially $3\Delta/2$ time steps.

The paper is organized as follows. In Section 2, we describe the algorithms we have evaluated for indirect migration without space constraints. In Section 3, we describe the algorithms we have evaluated for migration with space constraints. Section 4 describes how we create the demand graphs on which we test the migration algorithms while Sections 5 and 6 describe our experimental results. Section 7 gives an analysis and discussion of these results and Section 8 summarizes and gives directions for future research.

2 Indirect Migration without Space Constraints

We begin with a new algorithm, *Max-Degree-Matching* 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 vertex 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 vertex up with a bypass node. Finally we use the general matching algorithm [8] to expand this matching to a maximal matching and then send every edge in this new expanded matching. The full algorithm is given in Appendix A.1; a proof of the following theorem is given in [1].

Theorem 1. Max-Degree-Matching *computes a correct Δ-stage migration plan using at most $2n/3$ bypass nodes.*

We compare *Max-Degree-Matching* with *2-factoring* from Hall et al. which also computes an indirect migration plan without space constraints. Hall et al., show 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 Hall et al., 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 Migration with Space Constraints

The *Greedy Matching* algorithm (Algorithm 1) is a new and straightforward direct migration algorithm which obeys space constraints. This algorithm eventually sends all of the objects [1] but the worst case number of stages is unknown.

Algorithm 1 *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.
```

We compare *Greedy-Matching* with two provably good algorithms for migration with space constraints from Hall et al.. We refer to these algorithms as

4-factoring direct and *4-factoring indirect*. Hall et al. show 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.

4 Experimental Setup

The following table summarizes the theoretical results known for each algorithm[2].

Algorithm	Type	Space Constraints	Plan Length	Worst Case Max. Bypass Nodes
2-factoring [7]	indirect	No	$2\lceil \Delta/2 \rceil$	$n/3$
Max-Degree-Matching	indirect	No	Δ	$2n/3$
Greedy-Matching	direct	Yes	unknown	0
4-factoring direct [7]	direct	Yes	$6\lceil \Delta/4 \rceil$	0
4-factoring indirect [7]	indirect	Yes	$4\lceil \Delta/4 \rceil$	$n/3$

We tested these algorithms on four types of multigraphs[3]:

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). 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. *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 [1].

4.1 Creation of Load-Balancing Graphs

A migration problem can be generated from any pair of configurations of objects on nodes in a network. To generate the *Load-Balancing* graphs, we used two different methods of generating sequences of configurations of objects which might

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

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

occur in a real world system. For each sequence of say l configurations, $C_1, \ldots C_l$, for each i, $1 \leq i \leq l-1$, 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 7 iterations (enough to stabilize the node count) and so got two sequences of 7 configurations. For the second method, we took the 17 queries to a relational database in the TPC-D benchmark [4] 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 device. For both methods, we generated configurations based on two different types of devices. Thus for the Hippodrome method, we generated 4 sequences of 7 configurations (6 graphs) and for the TPC-D method, we generated 2 sequences of 17 configurations (16 graphs) for a total of 56 demand graphs.

5 Results on the *Load-Balancing Graphs*

5.1 Graph Characteristics

Detailed plots on the characteristics of the *load-balancing* graphs are given in [1] and are summarized here briefly. 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 density 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 density around 140.

5.2 Performance

Figure 1 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 (2) *Greedy-Matching* always takes fewer stages than 4-*factoring direct*.

6 Results on General, Regular and Zipf Graphs

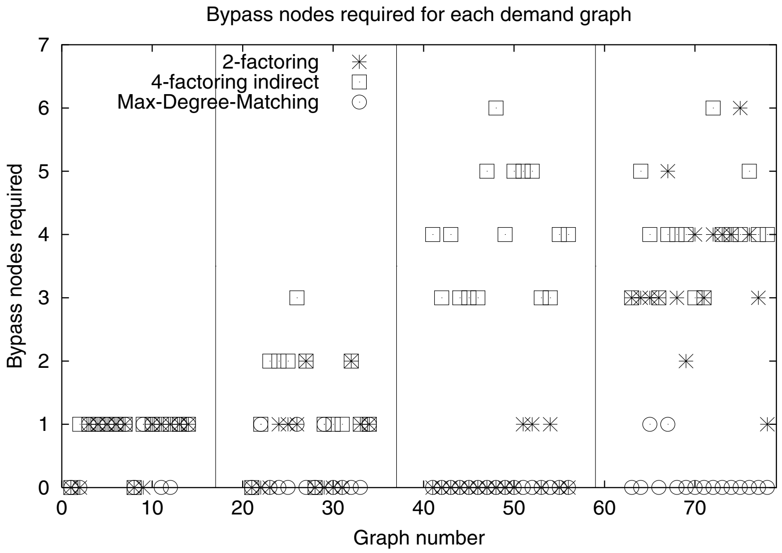
6.1 Bypass Nodes Needed

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*.

Varying Number of Nodes The three plots in the left column of Figure 2 give results for random graphs where the edge density 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 density fixed at 10 as the number of nodes increases from 100 to 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 very similar, starting at 3 and growing very 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*



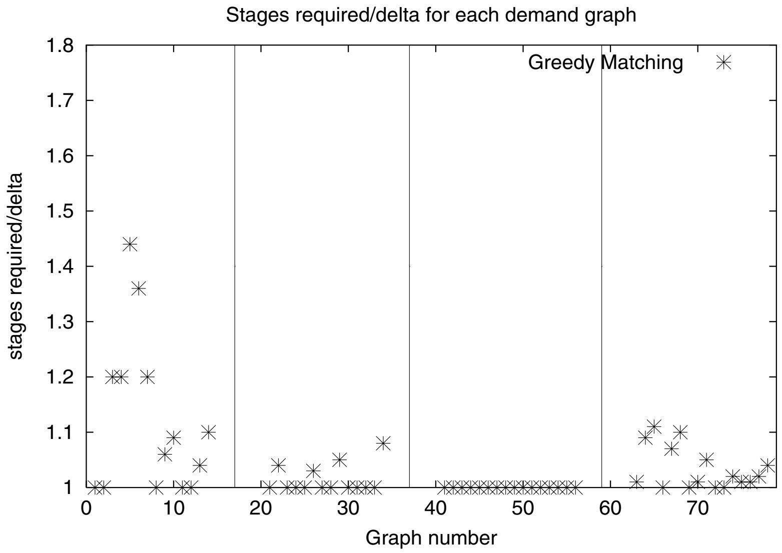


Fig. 1. The top plot gives the number of bypass nodes required for the algorithms *2-factoring*, *4-factoring indirect* and *Max-Degree-Matching* on each of the *Load-Balancing Graphs*. The bottom plot gives the ratio of time steps required to Δ for *Greedy-Matching* on each of the *Load-Balancing Graphs*. The three solid lines in both plots divide the four sets of *Load-Balancing Graphs*

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 Figure 2 show the number of bypass nodes used for graphs with a fixed number of nodes as the edge density 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 density 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 very 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.

6.2 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 Figure 3, 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 density Zipf graphs are the one weakness we found for *Greedy-Matching*.

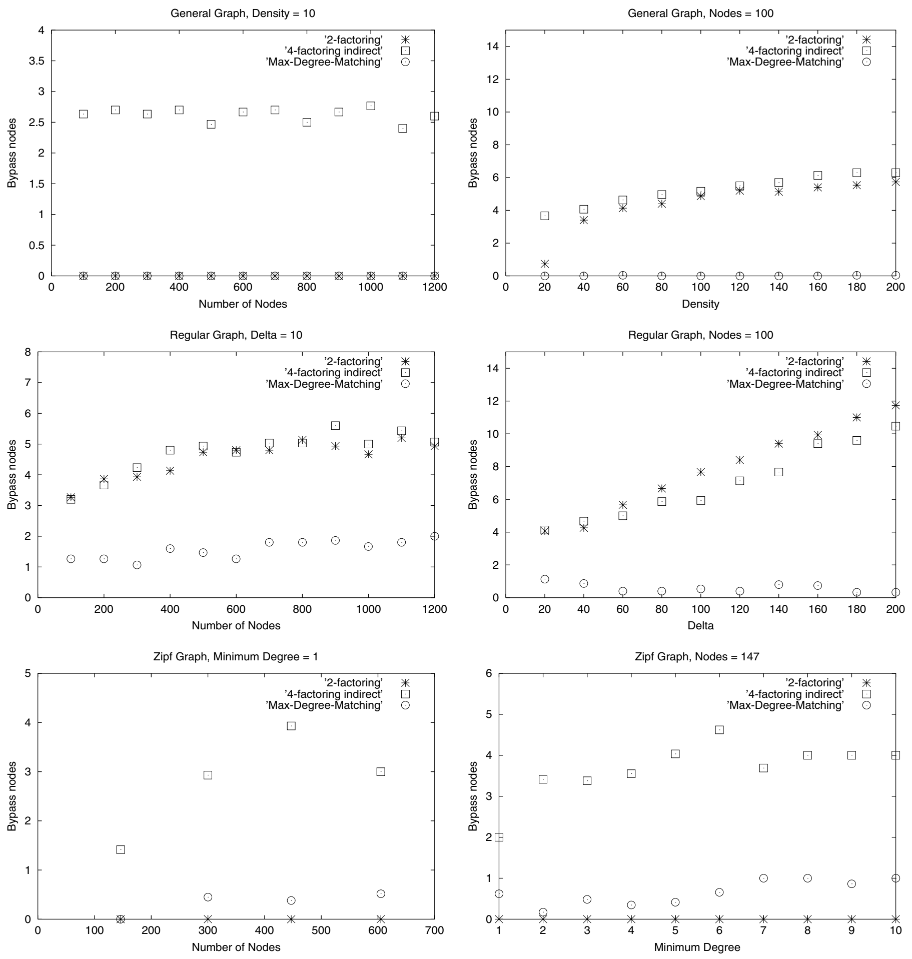


Fig. 2. These six plots give the number of bypass nodes needed for *2-factoring*, *4-factoring direct* and *Max-Degree-Matching* for the *General*, *Regular* and *Zipf Graphs*. The three plots in the left column give the number of bypass nodes needed as the number of *nodes* in the random graphs increase. The three plots in the right column give the number of bypass nodes needed as the *density* 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*

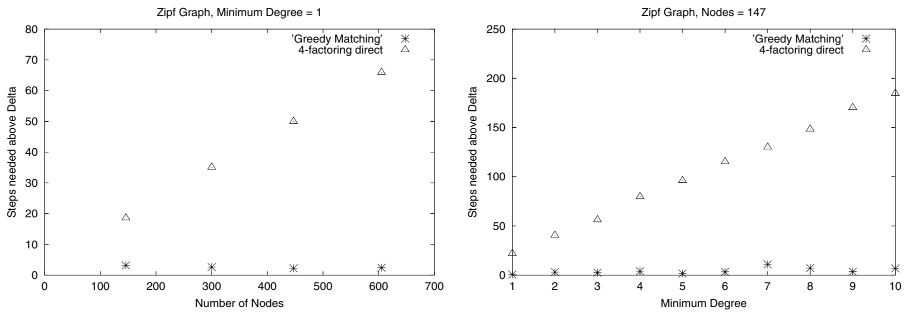


Fig. 3. Number of steps above *Delta* needed for *Greedy-Matching* on *Zipf Graphs*

7 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 very 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.

7.1 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 density 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 so much better than *2-factoring* and *4-factoring indirect* despite its worse theoretical bound.

8 Conclusion

We have introduced two new data migration algorithms and have empirically evaluated their performance compared with two algorithms from [7]. The metrics we used to evaluate the algorithms 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 found on several types of random and load-balancing graphs that the new algorithms outperform the algorithms from [7] on these two metrics despite the fact that the theoretical bounds for the new algorithms are not as good. Not surprisingly, we have also found that for all the algorithms tested, the theoretical bounds are overly pessimistic. We conclude that many of the algorithms described in this paper are both practical and effective for data migration.

There are several directions for future work. Real world devices have different I/O speeds. For example, one device might be able handle sending or receiving 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 very sparse topology which is more closely related to a tree than to a complete graph. We want good approximation algorithms for commonly occuring topologies (such as trees) and in general for arbitrary topologies. Saia [10] gives some preliminary approximation algorithms for migration with variable device speeds and different network topologies.

Another direction for future work is designing algorithms which work well for the online migration problem. In this paper, we have ignored loads imposed by user requests in devising a migration plan. A better technique for creating a migration plan would be to migrate the objects in such a way that we interfere as little as possible with the ability of the devices to satisfy user requests and at the same time improve the load balancing behavior of the network as quickly as possible. This may require adaptive algorithms since user requests are unpredictable.

A final direction for future work is designing algorithms which make use of free nodes when available but do not *require* them to perform well. 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.

References

1. E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An Experimental Study of Data Migration Algorithms. Technical report, University of Washington, 2001. 148, 149, 150
2. E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. Submitted to Symposium on Operating System Principles, 2001. 145, 150
3. E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *5th Intl. Workshop on Quality of Service*, Columbia Univ., New York, June 1997. 145
4. Transaction Processing Performance Council. TPC Benchmark D (Decision Support) Standard Specification Revision 2.1. 1996. 150
5. B. Gavish and O. R. Liu Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33:177–189, 1990. 145
6. 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. 145
7. J. Hall, J. Hartline, A. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *12th annual ACM-SIAM Symposium on Discrete Algorithms*, 2001. 145, 147, 149, 156
8. S. Micali and V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding a maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science*, 1980. 148, 158
9. T. Nishizeki and K. Kashiwagi. On the 1.1 edge-coloring of multigraphs. In *SIAM Journal on Discrete Mathematics*, volume 3, pages 391–410, 1990. 147
10. J. Saia. Data Migration with Edge Capacities and Machine Speeds. Technical report, University of Washington, 2001. 156
11. 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. 145

A Appendix

A.1 *Max-Degree-Matching*

Algorithm 2 *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 vertex.
3. Mark every other edge in the cycles and paths. For odd length cycles, one vertex will be left with no marked edges. Make sure that this is a vertex with an outgoing edge (and thus can be bypassed if needed). Each vertex has at most one edge marked. Mark every edge between a maximum degree vertex and a degree one vertex.
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 [8] with the matching that consists of marked edges.) Define S to be all edges in the matching.
5. For each edge vertex 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.