Efficient Disk Replacement and Data Migration Algorithms for Large Disk Subsystems*

Roger Zimmermann
Computer Science Department
University of Southern California
Los Angeles, California 90089
rzimmerm@usc.edu

Beomjoo Seo
Computer Science Department
University of Southern California
Los Angeles, California 90089
bseo@usc.edu

ABSTRACT

Random data placement has recently emerged as an alternative to traditional data striping. From a performance perspective, it has been demonstrated to be an efficient and scalable approach for large-scale storage systems. In this study we address the challenge of effectively managing the physical size of large data repositories. Specifically, we define the *disk replacement problem* (DRP) as the challenge of finding a sequence of disk additions and removals for a storage system while migrating the data and respecting the following constraints: (1) the data is initially balanced across the existing configuration, (2) the data must again be balanced across the new configuration, and (3) the data migration cost (either the amount of data moved or the elapsed time) must be minimized. Removing and adding disks in a large storage system may be required when devices are approaching the end of their life span (i.e., old disks are replaced with new ones) or when applications require increased storage space or performance. In practice, migrating data from old disks to new devices is complicated by the fact that the total number of disks that can physically be connected to the storage system is often limited by a fixed number of available *slots* and not all the old and new disks can be connected at the same time. We present solutions for both cases, where the number of disk slots is either unconstrained or constrained. We introduce a cost model that allows our algorithms to either optimize for minimal data movement or shortest elapsed time. Additionally, we suggest a heuristic to minimize the time cost while reducing the computational complexity. Finally, we extensively compare and evaluate all algorithms with analytical models and the results show that the presented approach provides efficient solutions to the disk replacement problem.

1. INTRODUCTION

In recent years, many computing environments have witnessed a dramatic growth in their need for storage space. Applications that handle media-rich data are at the forefront in this development but other data intensive applications are growing superlinearly as well. This trend has recently gained national attention when the scientists Jim Gray and Gordon Bell argued at a June 2003 meeting of the National Research Council's Computer Science and Telecommunications Board that federal money should be better spent on researching data-storage systems, rather than increasing the speed of computing [2].

It has long been well known that large data repositories can be built by aggregating collections of magnetic disk drives into array structures. Not only will this increase the total storage space that can be utilized, but parallel access to multiple disk drives increases the I/O throughput as well. However, aggregating many disks together increases the potential for hardware failures and protective measures based on data mirroring or parity computations are needed. Pioneering work in this field are has been done by Patterson et al. [3].

To achieve high I/O performance with disk arrays the access load should be evenly balanced across all the devices. Most existing data placement techniques use some form of striping to decluster the data across the storage system. However, these conventional techniques lack the flexibility of easy hardware reconfiguration that is necessary with todays very large storage systems. Specifically, data placement schemes such as RAID do often not allow the efficient addition or removal of disks and hence are a hindrance to the scalability of a system. With striping almost every data item must be relocated if the number of disks is changed. Some of these problems can be somewhat alleviated with techniques such as spare disks or multiple, small RAID partitions.

However, more recently a new type of data placement based on random block allocation has been shown to be an efficient and scalable contender for large scale storage systems without some of the idiosyncrasies of striping. For example, the RIO multimedia object server demonstrated the applicability of random data placement for media-rich storage systems [1]. Next to flexibility and reliability, performance is often of paramount importance with storage systems. When data blocks are randomly distributed across all the storage devices, an efficient directory service is necessary to locate individual blocks. It has been shown that random data placement can achieve similar performance to traditional striping techniques [4].

The random distribution has great advantages when data needs to be reorganized: blocks to migrate can randomly be chosen and moved, resulting in a new random distribution after the reorganization. At the same time, only the minimal amount of data is moved. For example, to add one disk to a four disk storage system, only 20% of the data must be relocated.

*This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), and IIS-0082826, and unrestricted cash/equipment gifts from Intel and SUN.

Online scalability is also closely related to a problem refered to as *data migration*. The general problem that a data migration algorithm addresses is to re-balance a storage system after it has become un-balanced (because of usage, etc.). Data migration works in two steps to (a) compute a new, load-balanced data distribution of all the currently existing data items (or blocks) in the system, and (b) efficiently move all the data blocks from their previous placement to their new locations.

A number of studies have investigated data migration algorithms [5, 6, 7]. Finding a migration plan can be mapped to a multi-graph edge-coloring problem to compute the minimum number of colors needed. Unfortunately, finding the optimal solution is $\mathcal{NP}$-complete and hence several approximation algorithms have been proposed in the literature. The data migration problem is very general in that it allows any data item to move from any device to any other device. In this paper, we address a problem that is slightly more constrained, but still covers many practical cases and very importantly allows an optimal solution to be computed in polynomial time.

Consider the following example. A storage system consists initially of, say, ten disk drives. Each disk stores 100 GB of data, the disk bandwidth is 20 MB/s and the data is load balanced across all the devices. A disk administrator wants to replace two older disks in the current configuration and also add ten new devices to increase the total storage space. At the end, the data should be load balanced again across all twenty disk drives (i.e., 50 GB on each disk). What is the best way to accomplish these device replacements and additions?

In this study we generalize the example given above and term it the *disk replacement problem* (DRP). Informally, DRP describes the challenge of finding a sequence of disk drives removals and additions to obtain a final, target storage system while minimizing the data migration cost. The migration cost may depend on the requirements of the application and could either be the total amount of data moved during the disk replacement operation, or the total elapsed time to complete the task.

Consider the above example again. A naive disk administrator may determine the following sequence. Copy the data of the two disks to be replaced onto the other eight disk drives. Then remove the two old disks, add two new ones and restore the data. Finally, add ten disks and distribute the data evenly across all the disks. An experienced administrator may take a different approach. Add twelve disks first, distribute the data evenly over all the disks except the disks to be removed soon, and then after moving the contents from two old disks to the existing disks, remove the two disks. Needless to say that the latter is more efficient than the former. The former requires $2 \times 2 \times 100$ GB data movement to backup and restore two disks (a total of 400 GB) plus the redistribution of half of all the data (500 GB) for a total of 900 GB. The latter only requires 600 GB data movement, which will be detailed in Section 5. The result is an improvement of 33% of the data movement. In terms of total elapsed time, the naive approach requires 12.5×10^3 seconds (3.47 hours), from two 5×10^3 seconds disk copies and 2.5×10^3 seconds data distribution time. The optimized method only takes 5×10^3 seconds, or 2.08 hours. As illustrated, the savings can be substantial especially in today's environments where tera-bytes of storage are not uncommon. Hence, a more efficient disk replacement algorithm will definitely be beneficial.

The contributions of this paper are twofold. First, we provide the mathematical models for DRP to find the optimal sequence of add/delete/replac operations. Second, we investigate a variation of the basic disk replacement problem and provide algorithmic results when the number of disk slots is constrained.

The rest of this paper is organized as follows. Section 2 surveys the related work. Section 3 formally defines the disk replacement problem by using graph representations in a homogeneous disk environment and analyzing the runtime complexity of the solution. Section 4 presents a cost model to be used for both space and time minimization. Section 5 introduces two algorithms to solve the disk replacement problem with unlimited disk slots. Next, in Section 6 we describe a divide and conquer algorithm to compute the minimal space cost when the number of disk slots is limited. We further suggest four fast heuristics for finding minimal cost solution. Section 7 contains a comparison of all algorithms and discusses the experimental results. Finally, Section 8 concludes the paper and discusses some future research directions.

2. RELATED WORK

Traditional clustered storage subsystems, such as disk arrays and disk striping, achieve dramatic disk I/O performance gains by accessing disks in parallel, but they require a well-organized data layout and an efficient data look-up service. However, round robin striping, one of the prevalent strategies, exhibits significant scalability problems especially when adding new disks or removing disks.

Randomized algorithms, assigning objects to randomly chosen storage nodes, adapt well in a dynamically changing environment. Random striping [8] showed that this scheme reduces the access latency significantly and satisfies real-time requirements with negligible deadline misses for media streaming purposes. Randomized I/O (RIO) [1] using randomized striping, when compared with the conventional striping, demonstrated that there is usually no significant difference and in some cases the randomized algorithm outperforms the round robin striping. Azar et al. [9] suggested an on-line load balancing strategy to place objects to d randomly chosen nodes uniformly where the objects are dynamically inserted and removed. The above three approaches primarily focus on object allocation to balance the load at the time of placement.

The data migration algorithm proposed in [5] concentrates on finding an efficient plan to move the data from one configuration to another assuming the network is complete. Due to the hardness of finding an optimal solution, a polynomial time migration is suggested by introducing bypass nodes.

In [10] the pseudo-random hash function was extended to evenly distribute the data among the storage nodes with high probability, to move a small fraction of data blocks during node addition or removal, and to search for data location efficiently. Because accessing the look-up tables in massively distributed storage systems such as Storage Area Network (SAN) may cause bottlenecks, this technique minimizes the look-up table size and uses computational power to calculate the data lay-out. Its placement and migration algorithm combine a hash function with a *cut-and-paste* mapping function that may be used in Step 2 and 3 of our algorithm $ABBD$. Placement schemes of above two systems, however, only deal with a sequence of atomic disk additions or removals.

3. DISK REPLACEMENT PROBLEM

Symbol	Meaning
R	disk bandwidth (read or write)
S	total amount of data stored in current system
C	set of disk slots
N	set of disks in use at current configuration
A	set of disks to add
D	set of disks to remove
c	max number of disk slots (=$\|C\|$)
n	number of disks in use (=$\|N\|$)
a	number of disks to add (=$\|A\|$), $0 \leq a' \leq a$
d	number of disks to remove (=$\|D\|$), $0 \leq d' \leq d$
$r(n, d, a, c)$	concise representation of disk scaling op.: start with n disks, then add a disks and delete d disks while using no more than c disk slots
s	amount of data in each disk before scaling op.
s'	amount of data in each disk after scaling op.
A_n^a	a disk addition in a given n disks
D_n^d	d disk deletion in a given n disks
$\omega(r)$	cost of disk scaling sequence r (ω_s : space cost, ω_τ : time cost)

Table 1: Symbolic notations used.

Table 1 summarizes the symbolic notations used throughout this paper. We use set representations to describe our proposed algorithms. The sets C, N, A, D, and their corresponding cardinalities, c, n, a, d, are sometimes used interchangeably when the meaning is unambiguous.

3.1 Problem Description

DRP should satisfy the following two pre-requisites.

PROPERTY 1. *After every atomic operation the data is balanced across the devices involved, i.e., the space is uniformly utilized.*

PROPERTY 2. *After every atomic operation all the block locations are well randomized, i.e., the workload imposed on every device is approximately equal.*

Properties 1 and 2 guarantee that the final configuration is load balanced and well randomized with high probability. Furthermore, since every intermediate configuration observes the same rules and creates no hotspots, it is possible to run the reorganization in parallel with the normal workload of the storage system, albeit at a slower pace.

We now formalize the disk replacement problem with a graph representation. Consider a weighted and directed graph $G_c = (V, E)$, where each vertex v in V represents a tuple of four non-negative integer variables (n, d, a, c). The meaning of the variables is as follows. The current number of disks in use is n and the next operation will remove d old disks and add a new disks. In any real storage system a constraint exists on how many disks can be attached. For example, one SCSI channel can typically support 15 storage devices. We refer to this maximum as the number of *slots* and denote it with c. For an unconstrained graph $c = \infty$, we omit the forth parameter in our notation, e.g., $(n, d, a, \infty) \equiv (n, d, a)$. The edge $(v_i(n_i, d_i, a_i, c),\ v_j(n_j, d_j, a_j, c),\ v_i, v_j \in V$, represents the replacement operation that removes $(d_i - d_j)$ old disks and adds $(a_i - a_j)$ new disks. It must satisfy the following nine graph constraints to guarantee that the graph G_c is **directed and acyclic**:

$$n \leq c \quad (1)$$
$$n - d + a \leq c \quad (2)$$
$$d \leq n \quad (3)$$
$$n + a \leq c \quad (4)$$
$$d_i \geq d_j \quad (5)$$
$$a_i \geq a_j \quad (6)$$
$$(d_i \neq d_j) \vee (a_i \neq a_j) \quad (7)$$
$$n_i - d_i + a_i = n_j - d_j + a_j \quad (8)$$
$$n_i + a_i - a_j \leq c \quad (9)$$

If a vertex, termed *disk scaling request (state, condition)*, satisfies the edge constraints 1 through 3, it is said to be *valid* ; otherwise, it is considered *invalid*. If a valid vertex satisfies the edge constraint 4, it is said to be *unbounded*; otherwise, it is *bounded*. Similarly, if an edge, termed a *disk scaling operation (transition)* which is incident to a valid pair of vertices satisfies edge constraints 5 through 8, it is said to be *valid*; otherwise, it is *invalid*. If a valid edge satisfies constraint 9, it is *unbounded*; otherwise it is *bounded*. A DRP graph G_c consisting of valid vertices and valid edges is directed and acyclic.

Each edge $e(v_i, v_j)$ is labelled with a non-negative weight $\omega(e)$, $\omega : E \rightarrow R^+$ that represents the data migration cost from v_i to v_j. The cost model will be discussed in detail in Section 4. The weight of a complete path $p = \langle v_0, v_1, \cdots, v_k \rangle$ is the sum of the weights of its constituent edges

$$\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

and the *shortest-path weight* from from u to v is denoted by

$$\delta(u, v) = \begin{cases} min\{\omega(p) : u \overset{p}{\leadsto} v\} & : \text{path exists from } u \text{ to } v \\ \infty & : \text{otherwise} \end{cases}$$

In other words, the shortest path from vertex u to vertex v is defined as any path p with weight $\omega(p) = \delta(u, v)$ [11]. Consequently, the disk replacement problem can be formalized as finding the shortest path from the initial configuration $v_0(n_0, d_0, a_0, c)$ to the final configuration $v_t(n_t = n_0 - d_0 + a_0, d_t = 0, a_t = 0, c)$, possibly with a given constraint c that denotes the maximum number of disk slots which cannot be exceeded at any step of the algorithm.

To facilitate an analytical proof, we separate the graph G_c into two graphs; one is the *unconstrained DRP graph* G_∞ where $c = \infty$ and all vertices and edges are unbounded. And the other is the *constrained DRP graph* G_c where all vertices and edges are valid.

Example: Figure 1 illustrates all the possible paths for adding three disks and deleting two, assuming disks are added or removed one at a time. Each vertex represents the state of an intermediate configuration, while the edges show the specific disk scaling operation (add or delete) with a weight that corresponds to the amount of data moved from the previous state to the current one. Note that states on the same horizontal level use the same number of disks. For example, states `Init`, `D`, and `J` all use the same number of disks n, while states `A`, `G`, and `End` use one more, i.e., $n + 1$. Depending

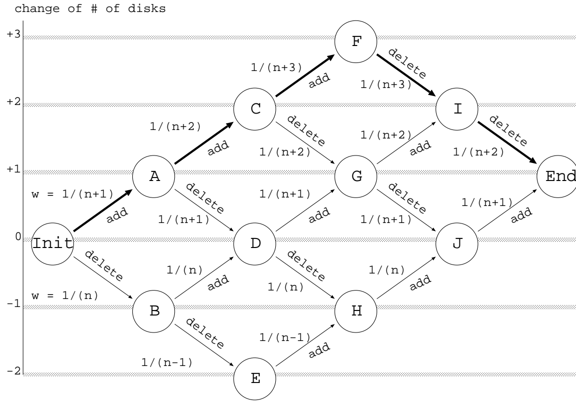

Figure 1: Example DRP graph for adding three disks while deleting two.

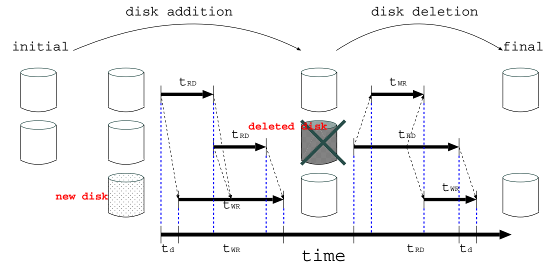

Figure 2: Illustration of the time cost model.

on which path is chosen, more or less slots are temporarily occupied. In the example graph, the path `Init-A-C-F-I-End` is the shortest path that minimizes the total space cost.

3.2 Complexity Analysis

The running time of finding the single-pair shortest path in a given Direct Acyclic Graph(DAG) – based on topological sort – is known to be $O(V+E)$, see [11]. Given the initial source $v_0(n_0, d_0, a_0, c)$, the maximum number of possible vertices is $O((a_0+1)(d_0+1)) = O(a_0 d_0)$. Thus, the running time is bounded by $O({a_0}^2 {d_0}^2)$ because $O(V+E) = O(V+V^2) = O(V^2)$. Hence, constrained or unconstrained DRP can be solved in polynomial time.

The complexity of building a DAG from a given source and destination is also dominated by $O(V^2) = O(E)$, which has the same order of magnitude of time complexity as the solution algorithm.

Still, the above running time bounds show that computing the shortest path can take a long time for large input data sets.

3.3 Aggregation of Disk Scaling Operations

To simplify our further discussions we introduce the notation of an aggregation of a sequence of operations, $r(n, d, a, c)$. The aggregation r represents a complete sequence of d disk deletions and a disk additions starting with the current n disks:

$$r(n,d,a,c) \stackrel{def}{=} e(v_0(n,d_0,a_0,c), v_1(n-d+a, d_0-d, a_0-a,c)) \text{ where } \exists\omega(e(v_0,v_1)) < \infty \text{ in } G_c$$

We further introduce a simplified notation for single disk additions: $A_n^1 = A_n \equiv r(n,0,1,c)$. Consequently, we represent multiple *simultaneous* disk additions as A_n^a. Note however that A_n^a is not equivalent to $r(n,0,a,c)$ because $r(n,0,a,c)$ can be achieved through a number of different paths: $\langle A_n^1\ A_{n+1}^1\ A_{n+2}^1\ A_{n+3}^1 \cdots A_{n+a-1}^1\rangle$ or $\langle A_n^2 A_{n+2}^1 \cdots A_{n+a-3}^3\rangle$, etc., or finally $\langle A_n^a\rangle$. A_n^a strictly represents the simultaneous addition of a disks.

Analogous, we define a single disk deletion as $D_n \equiv r(n,1,0,c)$ and multiple disks deletions at a time as D_n^d.

4. COST MODELS

Recall that the weight of each edge in a DRP graph represents the cost of performing either a disk addition or removal. We consider two types of data migration costs. The first one is based on the total amount of data being moved during the disk scaling operation and termed *space cost*, while the second one computes the total elapsed time and is correspondingly called *time cost*. We will start our discussion with the former, simpler model and then continue with the latter, more practical one. Both cost models are based on the observation that the total amount of data S stored in the initial disk configuration is unchanged after the disk scaling operation.

$$S = ns = (n-d+a)s' \tag{10}$$

where s and s' represent the amount of data on each disk before and after the scaling operation, respectively, and S denotes the total amount of data.

4.1 Space Cost

We will first assume that all the disk scaling operations are either simple disk additions or disk deletions. The space cost, ω_s, is the amount of data moved during a disk scaling operation.

$$\omega_s(A_n^a) \models \frac{a}{n+a} S \tag{11}$$

Equation 11 quantifies the space cost of simultaneously adding a disk to the current n-disk configuration. The amount of data moved between the initial and the final state is as'. Substituting from Equation 10 with $d=0$, we observe that the total amount of data moved is $as' = a(\frac{1}{n+a})S$. Analogous, Equation 12 quantifies the space cost of d simultaneous disk deletions from an n disk configuration.

$$\omega_s(D_n^d) \models \frac{d}{n} S \tag{12}$$

All data of D disks is moved to $(C-D)$ disks, resulting in ds data being moved, i.e., $ds = d(\frac{S}{n})$.

4.2 Time Cost

Even though minimizing the space cost will generally result in a fast migration operation, it does not guarantee to minimize the time spent. Occasionally moving more data but making use of parallel data transfers can reduce the overall migration time. The total elapsed time can be expressed with three independent time variables t_{RD}, t_d, and t_{WR}. t_{RD} is the read time to load data from storage devices into the memory. Similarly, t_{WR} denotes the time of writing data from memory into the storage devices. Finally, t_d denotes the delay of moving data through a network. Hence, the

edge weights are expressed as follows:

$$\omega_\tau(A_n^a) \text{ or } \omega_\tau(D_n^d) \models max(t_{RD} + t_d, t_{WR} + t_d) \quad (13)$$

Figure 2 illustrates how we derive the time cost of each disk scaling operation. In case of disk additions, the new disks are installed into empty disk slots and the data is moved from existing disks to the newly installed devices. Consequently, $(s - s')$ amount of data will be retrieved from N during t_{RD}. As soon as it is available in the system memory, the data will be transmitted over the network, requiring t_d amount of time. At the receiver side, the transferred data s' will be stored into the newly added disks. Therefore, the total elapsed time is the maximum of either the retrieval or the storage time. Similarly, in case of disk deletions, data is read from D disks and written to $(N - D)$ disks.

To simplify our model we assume that the disks are *fully connected and the network transmission time is negligible*, i.e., $t_d = 0$. Furthermore, we assume that the network and system bus bandwidth do not present a bottleneck and that the reading and writing data rates of the disk devices is the same, i.e., $R = R_{RD} = R_{WR}$.

By adding a disks to the current disk configuration n, Equation 13 simplifies to $max(\frac{s-s'}{R}, \frac{s'}{R}) = max(\frac{aS}{n(n+a)R}, \frac{S}{(n+a)R})$. Thus, Equation 13 can be rewritten as follows.

$$\begin{aligned}\omega_\tau(A_n^a) &= \begin{cases} \frac{S}{(n+a)R} = \frac{1}{aR}\omega(A_n^a) & : \text{ if } n \geq a \\ \frac{aS}{n(n+a)R} = \frac{1}{nR}\omega(A_n^a) & : \text{ otherwise} \end{cases} \\ &= \frac{1}{min(n,a)R}\omega(A_n^a) \quad (14)\end{aligned}$$

Intuitively the elapsed time is primarily affected by the current number of disks or by the number of disk additions with the same space cost. The cost model for disk deletions is derived analogous. Data is read from D and transmitted to $(N - D)$.

$$\begin{aligned}\omega_\tau(D_n^d) &= \begin{cases} \frac{S}{nR} = \frac{1}{dR}\omega(D_n^d) & : \text{ if } n \geq 2d \\ \frac{dS}{n(n-d)R} = \frac{1}{(n-d)R}\omega(D_n^d) & : \text{ otherwise} \end{cases} \\ &= \frac{1}{(n+d-max(n,2d))R}\omega(D_n^d) \quad (15)\end{aligned}$$

5. UNCONSTRAINED DRP

We first consider a variation of the disk replacement problem where the number of disk slots in the system is not bounded. More formally stated, in an unconstrained DRP environment every vertex and edge is unbounded, i.e., $r(n, d, a, \infty)$.

5.1 Non-Overlapping Approach - ABBD (Add–Balance–Balance–Delete)

As mentioned in Section 1, if a disk scaling sequence consists of two non-overlapping disk scaling operations, disk additions and disk removals, the optimal sequence for minimizing both the space and the time cost is to add all the new disks first (including re-balancing the data) and then remove all the old disks to be deleted (including first re-balancing the data). We present the Lemmas 1 to 8 to outline our argument.(see the proofs in Appendix)

LEMMA 1. $\omega_s(A_n^{i+j}) \leq \omega_s(A_n^i A_{n+i}^j)$.

LEMMA 2. $\omega_s(D_n^{i+j}) \leq \omega_s(D_n^i D_{n-i}^j)$.

LEMMA 3. $\omega_s(A_n^i D_{n+i}^j) \leq \omega_s(D_n^j A_{n-j}^i)$.

LEMMA 4. *The shortest path of* $\omega_s(r(n, d, a))$ *is* $A_n^a D_{n+a}^d$.

LEMMA 5. $\omega_\tau(A_n^{i+j}) \leq \omega_\tau(A_n^i A_{n+i}^j)$.

LEMMA 6. $\omega_\tau(D_n^{i+j}) \leq \omega_\tau(D_n^i D_{n-i}^j)$.

LEMMA 7. $\omega_\tau(A_n^i D_{n+i}^j) \leq \omega_\tau(D_n^j A_{n-j}^i)$.

LEMMA 8. *The shortest path of* $\omega_\tau(r(n, d, a))$ *is* $A_n^a D_{n+a}^d$.

These lemmas suggest two additional findings, which we will use in several heuristics described in later sections:

1. *Disk additions are preferable to disk deletions.*
2. *Add as many disks as possible.*

The above observations result in algorithm $ABBD$ shown below, which is the algorithmic implementation of Lemmas 4 and 8.

Algorithm 1 $ABBD(N, D, A)$

Require: $D \subset N, A \not\subset N$

1: add A
2: distribute the data evenly from N to $(N + A)$
3: distribute all data evenly from D to $(N - D + A)$
4: remove D from N

5.2 Merged Approach - ABD (Add–Balance–Delete)

Lines 2 and 3 of the $ABBD$ algorithm describe two non-overlapping data distribution operations, which result in wasteful data movement because some data stored on disk set D are moved to $(N - D)$ via A. Algorithm ABD is an enhancement of $ABBD$ and merges the two re-balance operations into one. Lines 3 through 7 of algorithm ABD disseminate the data probabilistically. Hence, ABD provides the optimal space cost for the unconstrained DRP. Lemma 9 shows the time cost of ABD.

Algorithm 2 $ABD(N, D, A)$

Require: $D \subset N, A \not\subset N$

1: $a \leftarrow |A|, d \leftarrow |D|, n \leftarrow |N|$
2: install a disks
3: **if** $(a \geq d)$ **then**
4: move all data in D to A uniformly, and move $\frac{a-d}{n-d+a}$ faction of data from each disk in $(N - D)$ to A
5: **else**
6: move $\frac{(n-d)(d-a)}{d(n-d+a)}$ amount of data to $(N - D)$, and move $\frac{an}{d(n-d+a)}$ amount of data to A from each disk in D
7: **end if**
8: remove d disks

LEMMA 9. *If* $a > d$*, then the time cost of algorithm* ABD *for* $r(n, d, a)$ *is* $\frac{1}{n}\frac{S}{R}$*, otherwise it is* $\frac{1}{n-d+a}\frac{S}{R}$.

Proof: If $a \geq d$, then a fraction of data read from $(N-D)$ is copied to A and all the content read from D is migrated to A. Therefore, the total time cost is the maximum of the following elapsed times:

- t_1 : time to read the data from $(N-D)$ is $\frac{s-s'}{R}$.
- t_2 : time to write the data to A is $\frac{s'}{R}$.
- t_3 : time to read the data from D is $\frac{s}{R}$.

$\Rightarrow max(t_1, t_2, t_3) = max(\frac{a-d}{n-d+a}\frac{S}{nR}, \frac{n}{n-d+a}\frac{S}{nR}, \frac{S}{nR}) = \frac{1}{n}\frac{S}{R}$.

If $a = d$, then all data read from D moves to A. Therefore, the algorithm only requires disk read or write time: $\frac{s}{R} = \frac{1}{n}\frac{S}{R}$. Finally, if $a < d$, then the data read from D is copied to either $(N-D)$ or A. Therefore, the total time cost is the maximum of the reading time from D and the writing time to $(N-D)$ and A. $max(\frac{s}{R}, \frac{(n-a)}{(n-d+a)}\frac{s}{R}, \frac{n}{(n-d+a)}\frac{s}{R}) = \frac{n}{n-d+a}\frac{S}{nR} = \frac{1}{n-d+a}\frac{S}{R}$. ∎

5.3 Comparison of $ABBD$ and ABD

Table 2 shows the computed space and time cost of two unconstrained DRP algorithms ($ABBD$ and ABD). Analogously, Figure 3 plots the cost ratio of $\frac{ABBD}{ABD}$ when fixing the number of currently used disks. It reveals that the ABD algorithm reduces the space and time cost of $ABBD$ by half, but the performance gain is gradually reduced when more disks are added. We conclude that it would be two times faster to perform the merged disk scaling operations rather than a sequence of atomic disk additions or deletions in the unconstrained disk environment.

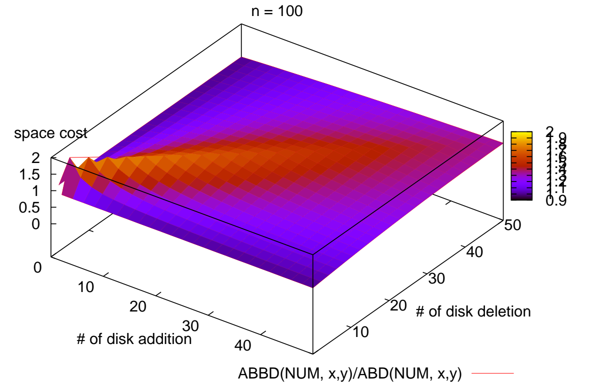

Figure 3.a: space cost ratio of $\frac{ABBD}{ABD}$

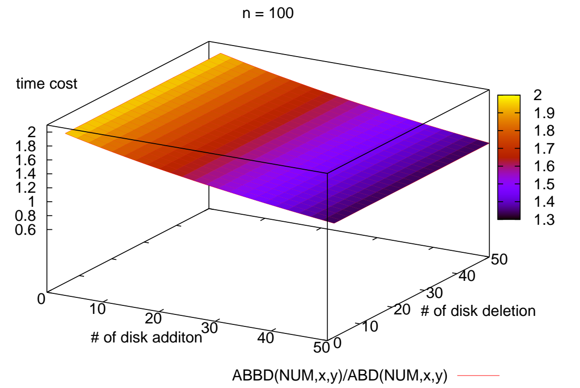

Figure 3.b: time cost ratio of $\frac{ABBD}{ABD}$

Figure 3: Computed space and time cost of $ABBD$ and ABD unconstrained algorithms as a function of the number of disk additions and the number of disk deletions with a fixed number of currently installed disks $(n = 100)$.

6. CONSTRAINED DRP

In the previous section, we presented the ABD algorithm for situations where we may temporarily add as many disks to the system as necessary. Next, we will address a more realistic disk replacement problem, $r(n, d, a, c)$, where the number of disk slots in the system is limited to c while the current number of disks plus the new disks $(n + a)$ exceed c.

The algorithms that we present here are based on the following observation. If the number of current disks plus the new disks to be added exceed the maximum number of available disk slots, $(n + a) > c$, then we may break the problem into two sub-parts: adding up to $a' \leq (c - n)$ disks first and then considering the remaining problem of adding $(a - a')$ disks. Hence, we introduce a *divide and conquer* algorithm to find optimal path sequence.

6.1 Divide-and-Conquer Approach (D&C)

A bounded disk scaling request can be divided into single unbounded disk scaling transition – using ABD for its solution – and its remaining disk scaling operation, because generally there is no direct transition from the bounded disk scaling request to the final termination state. This leads to a recursive algorithm where the subsequent disk scaling requests can be either bounded or unbounded. If a subsequent disk scaling request is unbounded, it has a direct transition sequence to reach the final disk configuration (using ABD). Otherwise, the problem is further subdivided until it resolves to an unbounded state.

Consider the following example. A bounded state $(5, 1, 2, 6)$ with two empty disk slots has three possible unbounded transitions: $ABD(5, 0, 1)$, $ABD(5, 1, 0)$, and $ABD(5, 1, 1)$. The sub-states after a single transition will be $(6, 1, 1, 6)$, $(4, 0, 2, 6)$, and $(5, 0, 1, 6)$, respectively. Sub-states $(4, 0, 2, 6)$ and $(5, 0, 1, 6)$ are unbounded and can be solved directly. However, the first sub-state, $(6, 1, 1, 6)$, is bounded. After the transition $ABD(6, 1, 0)$ the sub-substate is $(5, 0, 1, 6)$, which is unbounded and can be solved without further recursion.

Equation 16 formalizes the $D\&C$ algorithm. It can be viewed as a constrained DRP graph. Thus, its search cost is same as that of the shortest path algorithm. Note that both a' and d' in the equation cannot be zero at the same time. This solution finds both the optimal time and space cost with a constraint.

$$\omega(r(n, d, a, c)) =$$

$$\begin{cases} \omega(ABD(n, d, a)) & : c \geq (n + a) \\ \underset{\substack{0 \leq d' \leq d, \\ 0 \leq a' \leq (c-n)}}{\text{MIN}} \begin{pmatrix} \omega(ABD(n, d', a')) + \\ \omega(r(n - d' + a', d - d', a - a', c)) \end{pmatrix} & : \text{if bounded} \end{cases} \quad (16)$$

To obtain the complexity of the $D\&C$ algorithm we refer to Lemma 10.

Algorithm		Space cost ω ($\times S$)	Time cost ω_τ ($\times \frac{S}{R}$)
$ABBD$		$\frac{d+a}{n+a}$	$\frac{a}{min(n,a)(n+a)} + \frac{d}{(n+a+d-max(n+a,2d))(n+a)}$
ABD	$a \geq d$	$\frac{a}{n-d+a}$	$\frac{1}{n}$
	$a < d$	$\frac{d}{n}$	$\frac{1}{(n-d+a)}$

Table 2: Performance comparison of unbounded DRP algorithms $ABBD$ and ABD.

LEMMA 10. *A bounded state, (n, d, a, c), has $((c-n+1)(a-c+n)-1)$ possible subsequent bounded states after single unbounded disk scaling transition.*

Lemma 10 implies that probability to transition from one bounded state to another bounded state is approximately $\frac{a-c+n}{d+1)}$. As a result, when $(a-c+n)$ is smaller than $(d+1)$, the complexity of $D\&C$ algorithm approaches $((c-n+1)(d+1)-1)$. In other words, the lower bound of its complexity is $o(n \times d)$.

Proof: Given both on n and c, let the unbounded transition of the bounded scaling request $r(n, d, a, c)$ be $ABD(n, d', a', c)$ where $0 \leq d' \leq d$ and $0 \leq a' \leq (c-n)$. Thus, the total number of subsequent states after transitioning is $((c-n+1)(d+1)-1)$, because the special case, $d' = 0$ and $a' = 0$, must be excluded. After the transition, the resulting scaling request is $(n+a'-d', d-d', a-a', c)$. If this request requires more than $(c-n-a'+d')$ disk additions – that is, $(n+a'-d'+a-a') > c$ – it will be bounded again. Thus, the choice of d' determines whether the resulting state is bounded or unbounded. If $0 \leq d' < (a-c+n)$, then the new state will be bounded. As a result, the total number of subsequent bounded states of $r(n, d, a, c)$ is $((c-n+1)(a-c+n)-1)$. ∎

6.1.1 Memoization Technique

The $D\&C$ algorithm may suffer from increased execution time because it recomputes the same subproblems multiple times. The well-known solution to avoid redundant computations is the memoization technique which stores intermediate results for future use [11]. Intermediate stored results include n, d, a, c, ω (or ω_τ), d', and a' for any bounded request $r(n, d, a, c)$. Unbounded states need not be stored because their solution can be directly obtained from a simple calculation. The maximum number of intermediate results for any bounded request $r(n, d, a, c)$ is $(d+1)(a+1)-1$. For example, if we assign two bytes for each n, d, a, c, d', a' and four bytes for the computed cost result, then the system will require $16 \times ((d+1)(a+1)-1)$ bytes of memory. This is quite small – even for $c = 1000$, the memoization technique uses no more than 16 MB of memory.

In our experiments, all the presented algorithms use this memoization technique whenever beneficial.

6.2 Space Cost Minimization

Space cost minimization may be useful when the amount of data moved is crucial. We presently assume that the network or storage bus transmission time is negligible compared with the disk read and write times. If this assumption does not hold for a specific application (i.e., the data must be moved through a slow network), then minimizing the amount of data moved is desirable. We first suggest a slightly enhanced $D\&C$ algorithm termed EB and then introduce a linear solution with no memoization overhead.

Algorithm 3 $EB(N, D, A, C)$

```
Require: D ⊂ N, A ⊄ N
 1: D'' ← D, A'' ← A
 2: if |N| < |C| then
 3:   if |A| ≥ |D| then
 4:     repeat
 5:       D' ← select min(|D''|, |C| − |N|) number of disks
          from D''
 6:       A' ← select |D'| number of disks from A''
 7:       install A' to empty disk slots
 8:       move all data from D' to A' respectively
 9:       remove D' from disk slots
10:       D'' ← D'' − D'
11:       A'' ← A'' − A'
12:     until D'' = ∅
13:     ABD(N, ∅, A'')
14:   else
15:     repeat
16:       A' ← select min(|A''|, |C| − |N|) number of disks
          from A''
17:       D' ← select |A'| number of disks from D''
18:       install A' to empty disk slots
19:       move all the data from D' to A' respectively
20:       remove D' from disk slots
21:       D'' ← D'' − D'
22:       A'' ← A'' − A'
23:     until A'' = ∅
24:     ABD(N, D'', ∅)
25:   end if
26: end if
```

6.2.1 Exchange–Balance (EB)

The EB algorithm solves a bounded scaling request directly, which results in a smaller search space than with $D\&C$. Line 2 of algorithm EB examines whether the current disk scaling request includes at least one empty disk slot, regardless of its bounded or unbounded condition. If $a > d$, it adds as many disks A' as possible into the empty disk slots, selects $|A'|$ disks from the set D, copies their data to A', and removes the $|A'|$ selected disks. The complete operation is repeated until all D disks are removed. After the iterative exchange procedure, it installs the remaining new disks. When $a \leq d$ the procedure resembles that of the $a > d$ case. As shown in Table 3, when $a \leq d$ the space cost computed by EB is exactly same as that of ABD. However, this algorithm is not applicable for minimizing the time cost.

As an extension, the algorithm $D\&C + EB$ uses the EB procedure only when the system has at least one empty slot and the number of new disks is smaller than or equal to the number of disks to be deleted. Otherwise, $D\&C + EB$ executes equivalently to $D\&C$.

6.2.2 Linear Heuristic Algorithm (SPACE)

	space cost ω $(\times S)$	extra disk slots required
$a > d$	$\left(\frac{d}{n} + \frac{a-d}{n-d+a}\right)$	$a - d$
$a \leq d$	$\frac{d}{n}$	$c - n$

Table 3: Space cost and extra disk slots required during the execution of the EB Algorithm.

Algorithm 4 $SPACE(N, D, A, C)$

Require: $D \subset N, A \not\subset N$

1: $n \leftarrow |N|, d \leftarrow |D|, a \leftarrow |A|, e \leftarrow |C| - |N|$
2: **if** $(n - d + a) > |C|$ **then**
3: return ∞
4: **else if** $e \geq a$ **then**
5: return $\omega_s(ABD(N, D, A))$
6: **end if**
7: **if** $e = 0 \wedge d \geq a$ **then**
8: $D' \leftarrow$ select a disk from D
9: return $\omega_s(ABBD(N, D', \emptyset))$ + $\omega_s(EB(N - D', D - D', A, C))$
10: **else if** $d \geq a$ **then**
11: return $\omega_s(EB(N, D, A, C))$
12: **end if**
13: **if** $e > 0 \wedge a > d$ **then**
14: **if** $(a - d) = e \wedge a \neq d + 1$ **then**
15: $D' \leftarrow \emptyset$
16: $A' \leftarrow$ select $(a - d - 1)$ disks from A
17: **else if** $(a - d) < e$ **then**
18: $D' \leftarrow \emptyset$
19: $A' \leftarrow$ select $(a - d)$ disks from A
20: **end if**
21: **else**
22: $D' \leftarrow$ select a disk from D
23: $A' \leftarrow$ select e disks from A
24: **end if**
25: return $ABD(N, D', A')$ + $SPACE(N - D' + A', D - D', A - A', C)$

The $SPACE$ algorithm is a linear solution for finding the minimum space cost for bounded disk scaling requests. Its computation time depends only on the number of disks to be removed, regardless of the constraint and current disk configuration. The $SPACE$ heuristic has its origins in the findings of Section 5.1: to add as many disks as possible to the remaining disk slots. This is illustrated in lines 16, 19, and 23 of the algorithm. Additionally, $SPACE$ attempts to perform disk additions before deletions. The complexity of the $SPACE$ algorithm primarily depends on the last line, which is executed at most D times. Therefore, the complexity of $SPACE(N, D, A, C)$ is $O(|D|)$.

6.3 Time Cost Minimization

The $D\&C$ algorithm may also be used to find an optimal time cost solution for bounded disk scaling requests. Its complexity unchanged and equal to the $D\&C$ space cost algorithm.

6.3.1 Simple Heuristic Algorithm (TIME)

The $TIME$ algorithm selects a subset A' of the disks to be added A when attempting to divide a disk scaling request into an unbounded transition and its remainder. The set A' is chosen to be equal to the number of empty disk slots. As a result, the search space is halved. Furthermore, all possible state transitions are searched by varying the number of old disks to delete from 0 up to d disks. We call this search variable *pivot*.

The complexity of $TIME$ is determined by the number of times line 9 is executed. The memoization technique can be easily applied by checking for stored results before line 9 and memorizing sub-solutions after line 14.

Algorithm 5 $TIME(N, D, A, C)$

Require: $D \subset N, A \not\subset N$

1: $n \leftarrow |N|, d \leftarrow |D|, a \leftarrow |A|, e \leftarrow |C| - n$
2: **if** $(n - d + a) > |C|$ **then**
3: return ∞
4: **else if** $e \geq a$ **then**
5: return $\omega_\tau(ABD(N, D, A))$
6: **end if**
7: $min \leftarrow \infty$
8: $A' \leftarrow$ select e disks from A
9: **for** $\forall_{(D' \subset D) \wedge (D' \neq \emptyset \wedge A' \neq \emptyset)}$ **do**
10: $cost \leftarrow \omega_\tau(ABD(N, D', A'))+$ $TIME(N - D' + A', D - D', A - A', C)$
11: **if** $cost < min$ **then**
12: $min \leftarrow cost$
13: **end if**
14: **end for**
15: return min

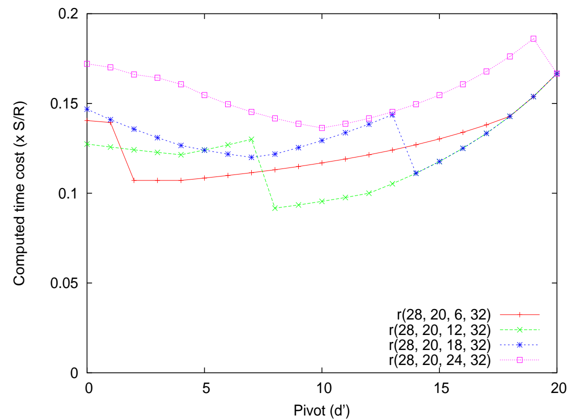

Figure 4: Multiple bowl-shaped time cost curves as a function of the pivot.

6.3.2 Local Minima Algorithm (LMIN)

The $TIME$ algorithm examines all the possible result states after an unbounded transition by varying the pivot value ranging from 0 to d. The time cost function of this pivot value may show multiple bowl shapes as illustrated in Figure 4. In many cases, the optimal pivot value is its median value. The reason is that if many disks are involved in an unbounded transition, there are benefits from disk I/O parallelism. The optimal pivot never lies at either end of the pivot range, because the resulting states may also require as many disks for their transition. For this reason, we intuitively acknowledge that selecting the median value of the pivot range will be a promising start to find a near optimal solution. However, as shown in Figure 4, this intuition can be wrong.

Algorithm 6 $LMIN(N, D, A, C)$

Require: $D \subset N, A \not\subset N$
1: $n \leftarrow |N|, d \leftarrow |D|, a \leftarrow |A|, e \leftarrow |C| - n$
2: **if** $(n - d + a) > |C|$ **then**
3: return ∞
4: **else if** $e \geq a$ **then**
5: return $\omega_\tau(ABD(N, D, A))$
6: **end if**
7: $A' \leftarrow$ select e disks from A
8: $pivot \leftarrow$ BINARY_SEARCH(N, D, A, C)
9: $D' \leftarrow$ select $pivot$ number of disks from D
10: return $\omega_\tau(ABD(N, D', A'))+$ $LMIN(N - D' + A', D - D', A - A', C)$

The $LMIN$ algorithm extends the intuition by using a binary search to locate a local minima. Even though it is not guaranteed to find the global minima of the time cost curve, it may find a local minima. This decreases the search space significantly by the factor of $\log D$. If necessary, the memoization technique can be applied before line 8 and after line 10.

7. EXPERIMENTAL EVALUATION

To evaluate the performance of the proposed DRP algorithms we implemented them on a test system and compared their results. The set of algorithms are ABD for unconstrained situations, $D\&C$, $D\&C + EB$, and $SPACE$ for constrained space cost optimizations and $D\&C$, $TIME$, and $LMIN$ for constrained time cost optimizations.

For the constrained experiments, we varied the disk slot constraint from 2 to 70. For all bounded disk scaling requests in a given constraint, we collected the computed cost, the elapsed time, and the number of items stored in memory. We did not store any intermediate results between test runs. We also computed the cost of the unbounded algorithm for all bounded test cases to have a baseline in order to examine how badly the system works by the bounded situation. Simulation programs were written in C and all experiments were executed on machine running Redhat 8 machine with two 2.0 GHz Intel Xeon CPUs and 1 GB memory.

All the algorithms use the memoization technique to reduce the computation time. We use a splay tree data structure, which is a self-adjusting binary search tree and minimizes the number of access operations over a period of time [13]. The binary search algorithm in $LMIN$ was implemented as follows. First, compute the costs of the two end points and the mid point. Discard the one point out of the three with the highest cost and use the other two as the end points of the next interval. Continue to subdivide the new intervals until a single point is reached.

7.1 Comparison of the Space Cost Minimization Algorithms

Figure 5 shows the ratio of the space costs incurred by different algorithms as a function of the disk slot constraint c. Both the average and the maximum ratio are illustrated. The results show that $D\&C$, $D\&C + EB$, and $SPACE$ all compute the same space cost (average and maximum). The average ratio of $SPACE$ to ABD converges to one as the disk constraint c increases, and the maximum ratio does not exceed a factor of two. We conclude that the amount of data moved in a constrained environment is never more than twice the minimum data moved in the unconstrained case.

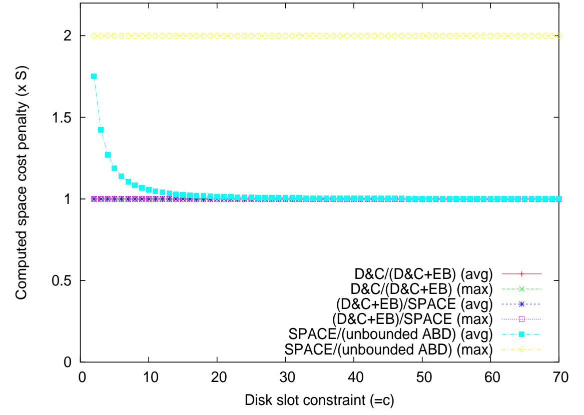

Figure 5: Computed space cost ratio of six algorithms as a function of the disk slot constraint.

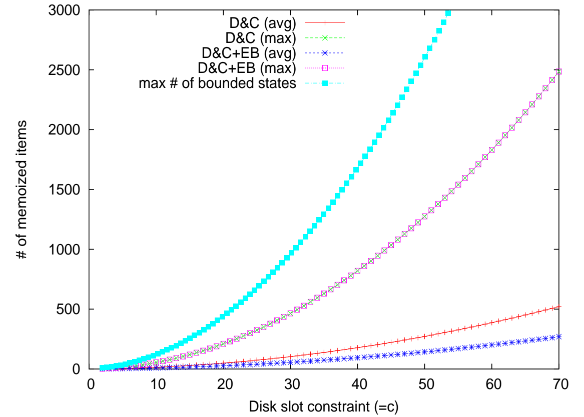

Figure 6: Number of stored intermediate results in space cost domain as a function of the disk slot constraint.

Figure 6 illustrates the average and maximum number of memoized intermediate results for each bounded DRP algorithm. The maximum number of bounded conditions per cluster is the upper bound of memoization size per constraint. However, every algorithms solving $r(n, d, a, c)$ stores only up to $(c - n + 1)(a - c + n) - 1$ bounded states. On average, the $D\&C$ with EB algorithm accesses a smaller number of subsequent bounded states than $D\&C$. For the maximum, there is no difference of the memoization size between $D\&C$ and $D\&C$ with EB. The $SPACE$ algorithm is not shown in this figure because it does not need to store any intermediate results.

Lastly, Figure 7 compares the average and maximum computation time measured for different algorithms. Note that the time scale uses milli-seconds and is logarithmic. The average measured time of the $SPACE$ algorithm is constant for all constraints, but the maximum time increases slowly for higher values of c. This is because the computational complexity of the $SPACE$ algorithm depends only on the number of disks to delete, which increases with a growing constraint c. As expected, the $D\&C$ and $D\&C$ with EB algorithms perform worse than $SPACE$. Even though $D\&C$ with EB shows a faster response time than that of $D\&C$ in the average case, there is no significant advantage of using $D\&C$ with EB over $D\&C$ alone. The former performs worse than the latter in

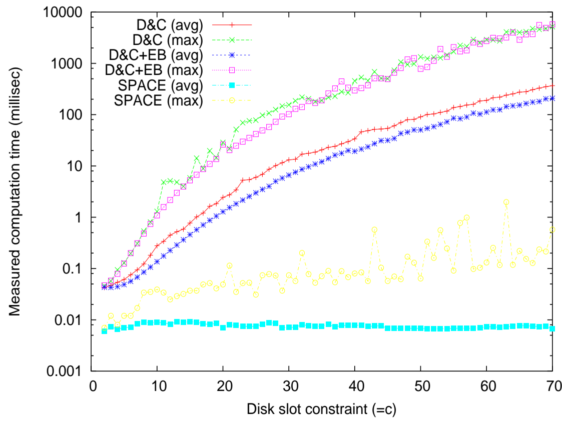

Figure 7: Measured computation time to find optimal cost solution as a function of the disk slot constraint.

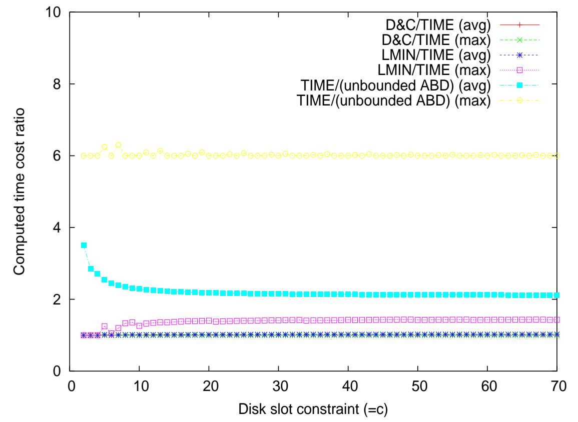

Figure 8: Computed time cost ratio of algorithms as a function of the disk slot constraint.

some cases, which is caused by the fact that different insertion and access sequences affects splay tree organization.

To summarize, the $SPACE$ achieves both minimal computation time and finds the optimal space cost in linear time. Even when $D\&C$ and $D\&C$ with EB algorithms use efficient memoization, the $SPACE$ algorithm outperforms them significantly. The $D\&C$ with EB generally performs better than $D\&C$ alone, but under some conditions the two algorithms are the same.

7.2 Comparison of Time Cost Minimization Algorithms

Figures 8 through 10 illustrate the same performance metrics as were used in the previous section. Here we compare the following four algorithms: $D\&C$, $TIME$, $LMIN$, and the unconstrained DRP algorithm, ABD.

In Figure 8 we see that the $D\&C$ and the $TIME$ algorithm perform the same within our test range and that the $LMIN$ solution produces a slightly worse result than $TIME$ solution does. The time costs computed by the constrained algorithms converge to a factor of two worse than that of the unconstrained algorithm on average. In the worst case, the time cost difference between the constrained and unconstrained algorithms are significant. This figure also shows that the worst time cost of the $LMIN$ algorithm increases slightly as the disk constraint increases. However, we expect that the gap between $LMIN$ and $TIME$ can be reduced if we adopt a more comprehensive search algorithm for $LMIN$ than the one currently implemented, with some expense to the search cost.

Figure 9 shows that $LMIN$ algorithm has smaller memoization size than either $D\&C$ or $TIME$, both on average and maximum. For the worst case, the difference between $LMIN$ and $TIME$ is more evident.

Figure 10 illustrates that the computation time of $LMIN$ increases more gradually than that of $TIME$. We also see that the average measured time of the $TIME$ algorithm is much longer than that of maximum measured time of $LMIN$. Furthermore, the average execution time of the $D\&C$ algorithm is much longer than the maximum time of $TIME$.

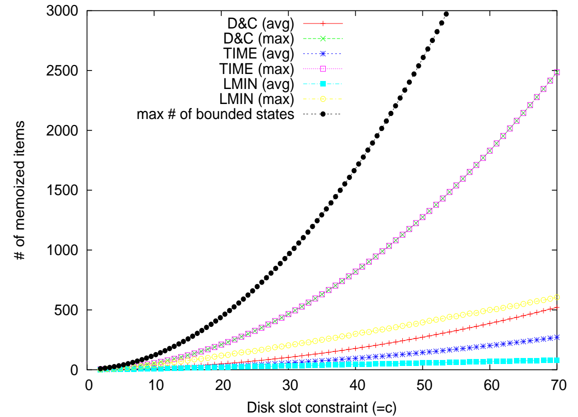

Figure 9: Number of stored intermediate results in time cost domain as a function of the disk slot constraint.

Overall, $TIME$ computes the same optimal time cost as the $D\&C$ algorithm, but with a shorter execution time. In terms of memoization size, $TIME$ usually requires less intermediate items than $D\&C$ but occasionally the same number of items in worst case. $LMIN$ has an obvious advantage in terms of both memoization size and computation time. But it would require a more comprehensive local minima detection algorithm to reduce the time cost penalty.

7.3 Numeric Example

So far we have analyzed the different algorithms without presenting the benefits of their application. Let us now examine an example of a bounded disk scaling request, $r(80, 50, 60, 100)$, i.e., in a storage system with 80 current disks we would like to add 60 new disks and remove 50 old ones with 20 available empty disk slots. Figure 11 and 12 show the space and time cost distribution when varying a' and d' values by using the $D\&C$ algorithm given in Equation 16. The $D\&C$ algorithm computes eleven different optimal sequences for minimizing the space cost

$$ABD(80, 0, 10) \rightarrow EB(90, 50, 50, 100)$$
$$ABD(80, 1, 11) \rightarrow EB(90, 49, 49, 100)$$
$$\cdots$$
$$ABD(80, 10, 20) \rightarrow EB(90, 40, 40, 100)$$

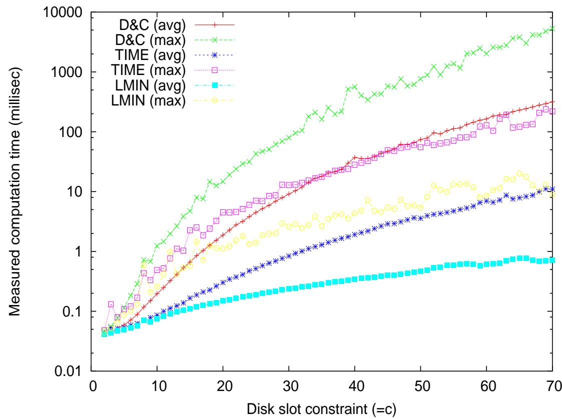

Figure 10: Measured computation time to find the time cost solution as a function of the disk slot constraint.

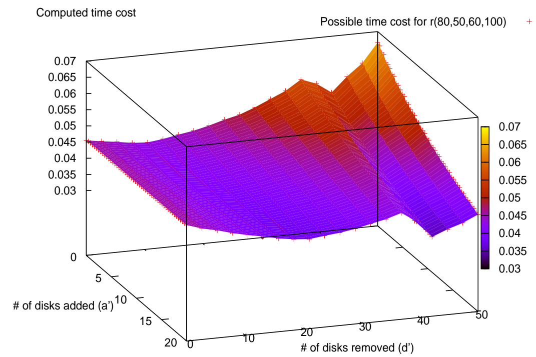

Figure 12: Time cost distribution of $r(80, 50, 60, 100)$ while varying a' disks and d' disks.

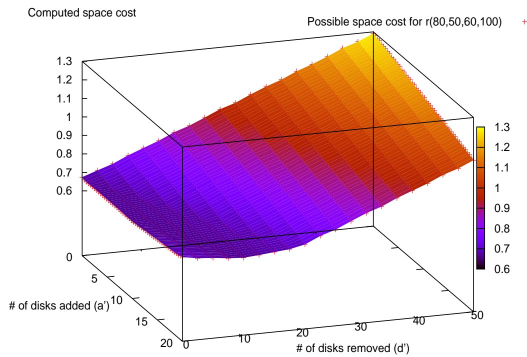

Figure 11: Space cost distribution of $r(80, 50, 60, 100)$ while varying a' disks and d' disks.

with a computed space costs of approximately $0.67S$. However, the $SPACE$ algorithm finds only one optimal disk scaling sequence: $ABD(80, 0, 10) \rightarrow EB(90, 50, 50, 100)$. If the wrong scaling sequence is chosen, for example deleting 50 disks first and adding 60 disks later – $ABD(80, 50, 0) \rightarrow ABD(30, 0, 60)$ – we suffer a 93% higher data movement than is optimal. Hence, assuming each disk stores 100 GB of data, the amount of data moved varies from 4,960 GB to 10,320 GB for different sequences.

Figure 12 shows the exhaustive time cost results obtained from the $D\&C$ algorithm for a $r(80, 50, 60, 100)$ operation varying from $0.033\frac{S}{R}$ to $0.067\frac{S}{R}$. The $TIME$ algorithm computes the optimal sequence with a time cost of $0.033\frac{S}{R}$ (add 20 and delete 40 disks first and then add 40 and delete 10 disks: $ABD(80, 40, 20) \rightarrow ABD(60, 10, 40)$). The worst case scaling sequence, $ABD(80, 50, 0) \rightarrow ABD(30, 0, 60)$, takes two times longer than the optimal sequence. Consequently, if each disk stores 100 GB and the disk bandwidth is 20 MB/s, then the optimal sequence takes 3 hours and 40 minutes, while the worst sequence takes 7 hours and 20 minutes.

8. CONCLUSIONS AND FUTURE WORK

We presented the disk replacement problem, DRP, of finding a sequence of disk additions and removals (also called a disk scaling operation) for a storage system while migrating the data and respecting a constraint on the total number of available disk slots. If the system has enough empty disk slots to add all new disks the operation is termed *unbounded*. Otherwise, it will be *bounded*. We analytically modelled DRP and proved it to be a variation of the single-source shortest path problem. Thus, the upper bound of the complexity of finding the optimal sequence is polynomial.

For unbounded disk scaling requests, we proposed the $ABBD$ and ABD algorithms. $ABBD$ is the naive implementation of scaling sequences with two atomic operations - disk additions and disk removals. The shortest sequence to minimize both the space and time cost is to add all new disks first and then to remove all disks to be deleted. The ABD algorithm improves upon $ABBD$ by combining two data re-balancing steps into one.

For bounded disk scaling requests, we proposed an exhaustive search algorithm based on a divide-and-conquer approach and termed $D\&C$, minimizing both the time and space costs. To reduce the its search space size, we introduced two space cost minimization heuristics, $D\&C + EB$ and $SPACE$, and two time cost minimization heuristics, $TIME$ and $LMIN$. The $D\&C + EB$ algorithm uses the EB component only when has system has at least one empty disk slot and the number of new disks is smaller than or equal to the number of disks to be removed. The $SPACE$ algorithm finds the optimal data moving sequence in linear time. $TIME$ reduces the $D\&C$ search space by filling all empty disk slots with new disks at once and then running the $D\&C$ algorithm. Finally, scaling sequences found by $LMIN$ are not guaranteed to be optimal but the search space is again reduced compared with $TIME$. Table 4 summarizes the cost and complexity results of all presented algorithms.

There are several directions in which we plan to extend our work. First, our algorithms may also be extended to hash based media streaming applications. For example, for Content Distribution Networks (CDN) that are based on Distributed Hash Tables (DHT) we may distribute the media content by segmenting it into successive data blocks and placing them on distributed disks (or nodes) randomly. Due to the dynamic nature of a CDN where nodes may newly join or leave, the data layout must be reorganized after such an operation. Otherwise, the average response time to locate the data will increase, and thus the real-time delivery is not guaranteed. Our algorithms can rebalance the data layout in a dynamically changing environment, and hence guarantee the load balance of the data accesses. Moreover, we have proved that multiple node additions or departures are preferred over a single node addition and de-

		Computed Cost (ω)		Complexity (avg.)	
		Space (ω_s)	Time (ω_τ)	Space	Time
Unbounded	$ABBD$	non-optimal	non-optimal	–	constant
	ABD	optimal[a]	optimal[a]	–	constant
Bounded	$D\&C$	optimal[b]	optimal[b]	high	high
	$D\&C+EB$	optimal[b]	–	medium	medium
	$SPACE$	optimal[b]	–	–	extremely low
	$TIME$	–	optimal[b]	medium	medium
	$LMIN$	–	near optimal[b]	low	low

[a]Optimal for unbounded cases.
[b]Optimal for bounded cases.

Table 4: Summary of all proposed DRP algorithms.

parture. This means that it would be advantageous to perform data reorganization periodically after "batching" enough node joins and leaves together, rather than on an individual basis.

One aspect of data migration that we have not addressed in this paper is the limited storage of each disk. Hence, in some cases the presented algorithms may exceed the physically available storage on some devices temporarily. Such an overcommitment will happen only when the disk scaling operation involves excessive disk removals. These situations can be detected by keeping track of the disk storage capacities during computations and avoiding such cases. Another extension of this work will be to apply the proposed solutions to heterogeneous storage environments.

9. REFERENCES

[1] R. Muntz, J. Santos, and S. Berson, "RIO: A Real-time Multimedia Object Server," in *ACM Sigmetrics Performance Evaluation Review*, September 1997, vol. 25.

[2] John Markoff, "In Computing, Weighing Sheer Power Against Vast Pools of Data," *The New York Times*, June 2, 2003, Section: Technology.

[3] David A. Patterson, Garth Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1988, pp. 109–116.

[4] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto, "Comparing Random Data Allocation and Data Striping in Multimedia Servers," in *Proceedings of the SIGMETRICS Conference*, Santa Clara, California, June 17-21 2000.

[5] Joseph Hall, Jason Hartline, Anna R. Karlin, Jared Saia, and John Wilkes, "On Algorithms for Efficient Data Migration," in *12th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Washington DC, January 7-9, 2001.

[6] Eric Anderson, Joe Hall, Jason Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes, "An Experimental Study of Data Migration Algorithms," in *WAE: 5th International Workshop of Algorithm Engineering*, Aarhus, Denmark, August 28-30, 2001, LNCS.

[7] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence Mustafa Uysal, and Alistair Veitch, "Hippodrome: Running Circles around Storage Administration," in *Conference on File and Storage Technologies (FAST'02)*, USENIX, Berkeley, CA, January 28-30, 2002, pp. 175–188.

[8] J. Alemany and J. S. Thathacher, "Random Striping News on Demand Servers," Tech. Rep. TR-97-02-02, University of Washington, 1997.

[9] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal, "Balanced allocations," *SIAM Journal on Computing*, vol. 29, no. 1, pp. 180–200, 2000.

[10] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Efficient, Distributed Data Placement Strategies for Storage Area Networks," in *ACM Symposium on Parallel Algorithms and Architecture*, 2000, pp. 119–128.

[11] T.H. Cormen et al., *Introduction to Algorithms*, The MIT Press, second edition, 2001.

[12] Roger Zimmermann and Beomjoo Seo, "Efficient Disk Replacement and Data Migration Algorithms," Technical report, University of Southern California, 2003.

[13] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, 1985.

APPENDIX

Throughout this appendix, we omit constant part of each cost like S, and $\frac{S}{R}$ for simpler proofs. Due to the limitation of paper space requirement, we omit proofs for lemma 5, 6 7. Although they have lots of branch conditions to be proved, their proofs are straight forward.

Lemma 1 $\omega(\langle A_n^{i+j}\rangle)$ is less than or equal to $\omega(\langle A_n^i A_{n+i}^j\rangle)$.

Proof: $\omega(\langle A_n^i A_{n+i}^j\rangle) - \omega(\langle A_n^{i+j}\rangle) = \frac{i}{n+i} + \frac{j}{n+i+j} - \frac{i+j}{n+i+j} = \frac{ij}{(n+i)(n+i+j)} \geq 0$. Therefore, lemma 1 is correct. ∎

Lemma 2 $\omega(\langle D_n^{i+j}\rangle)$ is less than or equal to $\omega(\langle D_n^i D_{n-i}^j\rangle)$.

Proof: $\omega(\langle D_n^i D_{n-i}^j\rangle) - \omega(\langle D_n^{i+j}\rangle) = \frac{i}{n} + \frac{j}{n-i} - \frac{i+j}{n} = \frac{ij}{n(n-i)} \geq 0$.

Therefore, lemma 2 is correct. ∎

Lemma 3 $\omega(\langle A_n^i D_{n+i}^j\rangle)$ is less than or equal to $\omega(\langle D_n^j A_{n-j}^i\rangle)$.

Proof: $\omega(\langle A_n^i D_{n+i}^j\rangle) = \frac{i}{n+i}S + \frac{j}{n+i}S, \omega(\langle D_n^j A_{n-j}^i\rangle) = \frac{j}{n}S + \frac{i}{n-j+i}S$. Let f be the numerator of $\{\omega(\langle D_n^j A_{n-j}^i\rangle) - \omega(\langle A_n^i D_{n+i}^j\rangle)\}$.

$$\begin{aligned}
f &= in(n+i) + j(n+i)(n-j+i) - (i+j)n(n-j+i) \\
&\geq in(n-j+i) + j(n+i)(n-j+i) - (i+j)n(n-j+i) \\
&= (n-j+i)\{in + j(n+i) - (i+j)n\} \\
&= (n-j+i)ij \\
&\geq 0
\end{aligned}$$

Therefore, lemma 3 is correct. ∎

Lemma 4 The shortest path of $\omega(r(n,d,a))$ is $\langle A_n^a D_{n+a}^d\rangle$.

Proof: We prove it by contradiction. Let's assume that the shortest path of minimizing the space cost of $r(n,d,a)$ not be $\langle A_n^a D_{n+a}^d\rangle$. It implies that the space cost of the other operation sequence is less than that of $\langle A_n^a D_{n+a}^d\rangle$.

And shortest path of $r(n,a,d)$ belongs to the following sequence; $\langle A_n^{\alpha_1} D_{n+\alpha_1}^{\beta_1} A_{n+\alpha_1-\beta_1}^{\alpha_2} D_{\cdots}^{\beta_2} \cdots A_{\cdots}^{\alpha_I} D_{\cdots}^{\beta_I}\rangle$ where $\sum_{i=1}^{I}\alpha_i = a, \sum_{i=1}^{I}\beta_j = d$ and $\alpha_i \in \{0,1,2,\cdots,a\}, \beta_j \in \{0,1,2,\cdots,d\}$ except $\langle A_n^a D_{n+a}^d\rangle$.

Afterwards we omit the subscript of each disk scaling operation for simplicity. And

$$\begin{aligned}
\omega(A^{\alpha_1}D^{\beta_1}A^{\alpha_2}D^{\beta_2}\cdots A^{\alpha_I}D^{\beta_I}) &= \omega(A^{\alpha_1}) + \omega(D^{\beta_1}) + \omega(A^{\alpha_2}) + \omega(D^{\beta_2})\cdots\omega(A^{\alpha_I}) + \omega(D^{\beta_I})) \\
&\geq \omega(A^{\alpha_1}) + \omega(A^{\alpha_2}) + \omega(D^{\beta_1}) + \omega(D^{\beta_2})\cdots\omega(A^{\alpha_I}) + \omega(D^{\beta_I})) \quad (17) \\
&= \omega(A^{\alpha_1}A^{\alpha_2}D^{\beta_1}D^{\beta_2}A^{\alpha_3}\cdots A^{\alpha_I}D^{\beta_I}) \\
&\geq \omega(A^{\alpha_1}A^{\alpha_2}D^{\beta_1}A^{\alpha_3}D^{\beta_2}\cdots A^{\alpha_I}D^{\beta_I}) \\
&\geq \omega(A^{\alpha_1}A^{\alpha_2}A^{\alpha_3}D^{\beta_1}D^{\beta_2}\cdots A^{\alpha_I}D^{\beta_I}) \\
&\cdots \\
&= \omega(A^{\alpha_1}A^{\alpha_2}\cdots A^{\alpha_I}D^{\beta_1}D^{\beta_2}\cdots D^{\beta_I}) \\
&\geq \omega(A^{\alpha_1+\alpha_2+\cdots+\alpha_I}D^{\beta_1}D^{\beta_2}\cdots D^{\beta_I}) \quad (18) \\
&\geq \omega(A^{\alpha_1+\alpha_2+\cdots+\alpha_I}D^{\beta_1+\beta_2+\cdots+\beta_I}) \quad (19) \\
&= \omega(A^a D^d)
\end{aligned}$$

Equation 17 is derived from lemma 3, equation 18 from lemma 1, equation 19 from lemma 2. This result contradicts our initial assumption. Therefore, lemma 4 is correct. ∎

Lemma 5 $\omega_\tau(\langle A_n^{i+j}\rangle)$ is less than or equal to $\omega_\tau(\langle A_n^i A_{n+i}^j\rangle)$.

Proof:

$\omega_\tau(\langle A_n^i A_{n+i}^j\rangle) - \omega_\tau(\langle A_n^{i+j}\rangle) = \frac{i}{min(n,i)(n+i)} + \frac{j}{min(n+i,j)(n+i+j)} - \frac{i+j}{min(n,i+j)(n+i+j)}$.

Let f be the numerator of $\{\omega_\tau(\langle A_n^i A_{n+i}^j\rangle) - \omega_\tau(\langle A_n^{i+j}\rangle)\}$.

$f = (i*min(n+i,j)*min(n,i+j)*(n+i+j)) + (j*min(n,i)*min(n,i+j)*(n+i)) - ((i+j)*min(n,i)*min(n+i,j)*(n+i))$

If $(n+i) \geq j, n \geq (i+j), n \geq j$, then $f = (j(i+j)ij) + (ij(i+j)(n+i)) \geq 0$.

If $(n+i) \geq j, n \geq (i+j), n < i$, then $f = (ij(i+j)(n+i+j)) + (jn(i+j)(n+i)) - ((i+j)nj(n+i)) = ij(i+j)(n+i+j) \geq 0$.

If $(n+i) \geq j, n < (i+j), n \geq i$, then

$$\begin{aligned} f &= (ijn(n+i+j)) + (ijn(n+i)) - ((i+j)ij(n+i)) = ij((n(n+i+j)) + (n(n+i)) - ((i+j)(n+i))) \\ &= ij(2n^2 + ni - i^2 - ij) = ij(n(2n+i) - i(i+j)) > ij(n(2n+i) - n(i+j)) = ijn(2n-j) > 0 \\ &\because n \geq (j-i) > (j-n) \Rightarrow 2n > j \end{aligned}$$

If $(n+i) \geq j, n < (i+j), n < i$, then

$$\begin{aligned} f &= (ijn(n+i+j)) + (jnn(n+i)) - (nj(i+j)(n+i)) = (ijn(n+i+j)) + (jnn(n+i)) - (nj(i+j)(n+i)) \\ &= jn((n+i+j)i) + (n(n+i)) - ((i+j)(n+i))) = jn(ni + n^2 - nj) = jn(n^2 + n(i-j)) = jn(ni - n(j-n)) \\ &> ni - ni = 0 \end{aligned}$$

If $(n+i) < j, n \geq (i+j), n \geq i$, then

$$f = (i(n+i)(i+j)(n+i+j)) + (ij(i+j)(n+i)) - (i(i+j)(n+i)^2) = i(n+i)(i+j)2j \geq 0$$

If $(n+i) < j, n \geq (i+j), n < i$, then

$$f = (i(n+i)(i+j)(n+i+j)) + (jn(i+j)(n+i)) - ((i+j)n(n+i)^2) = (n+i)(i+j)((i(n+i+j)) + (nj) - (n(n+i)))$$

Let $f' = (i(n+i+j)) + (nj) - (n(n+i)) = i^2 - n^2 + ij + nj = (i-n)(i+n) + ij + nj > 0 \therefore f \geq 0$

If $(n+i) < j, n < (i+j), n \geq i$, then

$$\begin{aligned} f &= (in(n+i)(n+i+j)) + (ijn(n+i)) - (i(i+j)(n+i)^2) = i(n+i)((n(n+i+2j)) - ((i+j)(n+i))) \\ &= i(n+i)(n^2 + ni + 2nj - ni - i^2 - nj - ij) = i(n+i)(n^2 - i^2 + 2nj - ij) = i(n+i)((n-i)(n+i) + j(2n-i)) > 0 \end{aligned}$$

If $(n+i) < j, n < (i+j), n < i$, then

$$\begin{aligned} f &= (in(n+i)(n+i+j)) + (jnn(n+i)) - ((i+j)n(n+i)^2) = n(n+i)((i(n+i+j)) + (jn) - ((i+j)(n+i))) \\ &= n(n+i)(ij - ij) = 0 \end{aligned}$$

Therefore, lemma 5 is correct. ∎

Lemma 6 $\omega_\tau(D_n^{i+j}) \leq \omega_\tau(D_n^i D_{n-i}^j)$.

Proof:

$\omega_\tau(\langle D_n^i D_{n-i}^j\rangle) - \omega_\tau(\langle D_n^{i+j}\rangle) = \frac{i}{n(n+i-max(n,2i))} + \frac{j}{(n-i)(n-i+j-max(n-i,2j)))} - \frac{i+j}{n(n+i+j-max(n,2(i+j))}$.

Let f be the numerator of $\{\omega_\tau(\langle D_n^i D_{n-i}^j\rangle) - \omega_\tau(\langle D_n^{i+j}\rangle)\}$.

$$\begin{aligned} f = &(i(n-i)(n+i+j-max(n,2i+2j))(n-i+j-max(n-i,2j))) \\ &+(jn(n+i+j-max(n,2i+2j))(n+i-max(n,2i))) \\ &-((i+j)(n-i)(n+i-max(n,2i))(n-i+j-max(n-i,2j))) \end{aligned}$$

If $n \geq 2(i+j), (n-i) \geq 2j, n \geq 2i$, then

$$f = (i(i+j)j(n-i)) + (j(i+j)in) - ((i+j)ij(n-i)) = ij(i+j)(n-i+n-n+i) = nij(i+j) \geq 0$$

If $n \geq 2(i+j), (n-i) \geq 2j, n < 2i$, then

$$\begin{aligned} f &= (i(i+j)j(n-i)) + (j(i+j)(n-i)n) - ((i+j)(n-i)j(n-i)) = (i+j)(n-j)(ij+jn-jn+ij) \\ &= (i+j)(n-j)(2ij) \geq 0 \end{aligned}$$

If $n \geq 2(i+j), (n-i) < 2j, n \geq 2i$, then

$$f = (i(i+j)(n-i-j)(n-i)) + (j(i+j)in) - ((i+j)i(n-i-j)(n-i)) = ijn(i+j) \geq 0$$

If $n \geq 2(i+j), (n-i) < 2j, n < 2i$, then

$$\begin{aligned} f &= (i(i+j)(n-i-j)(n-i)) + (j(i+j)(n-i)n) - ((i+j)(n-i)(n-i-j))(n-i)) \\ &= (i+j)(n-i)((i(n-i-j)) + jn - ((n-i)(n-i-j))) = (i+j)(n-i)^2 n \geq 0 \end{aligned}$$

If $n < 2(i+j), (n-i) \geq 2j, n \geq 2i$, then

$$\begin{aligned} f &= (i(n-i-j)j(n-i)) + (j(n-i-j)in) - ((i+j)ij(n-i)) \\ &= ij(2n^2 - 4in + 2i^2 - 3nj + 2ij) = ij(2(n-i)^2 + (j(2i-3n))) \end{aligned}$$

Let's assume $f' = 2(n-i)^2 + (j(2i-3n))$.

If $i \geq 2j$, then $2i \leq n < 2(i+j)$. Let n be $(2i+k)$ where $(0 \leq k < 2j)$. Therefore, $f'(k) = k^2 - (3j-2i)k + 2i(i-2j)$. Because $3j < 2i$, f' is greater than $f'(0) = 2i(i-2j) \geq 0$. If $i < 2j$, then $(i+2j) \leq n < (2i+2j)$. Let n be $(i+2j+k)$ where $(0 \leq k < i)$. Therefore, $f'(k) = 2k^2 + 5jk + j(2j-i)$. Similarly, $f'(0) > 0$. Consequently, $f > 0$.

If $n < 2(i+j), (n-i) \geq 2j, n < 2i$, then

$$\begin{aligned} f &= (i(n-i)(n-i-j)j) + (jn(n-i-j)(n-i)) - ((i+j)(n-i)(n-i)j) \\ &= j(n-i)(n^2 - ni - 2nj) = j(n-i)n(n-i-2j) \geq 0 \end{aligned}$$

If $n < 2(i+j), (n-i) < 2j, n \geq 2i$, then

$$\begin{aligned} f &= (i(n-i)(n-i-j)(n-i-j)) + (jn(n-i-j)i) - ((i+j)(n-i)i(n-i-j)) \\ &= i(n-i-j)(((n-i)(n-i-j)) + jn - (i+j)(n-i)) = i(n-i-j)(n^2 - 3ni - nj + 2i^2 + 2ij) \\ &= i(n-i-j)(n-2i)(n-i-j) = i(n-i-j)^2(n-2i) \geq 0 \end{aligned}$$

If $n < 2(i+j), (n-i) < 2j, n < 2i$, then

$$\begin{aligned} f &= (i(n-i)(n-i-j)(n-i-j)) + (jn(n-i-j)(n-i)) - ((i+j)(n-i)(n-i(n-i-j)) \\ &= (n-i)(n-i-j)(i(n-i-j) + jn - (i+j)(n-i)) = 0 \end{aligned}$$

Therefore, lemma 6 is correct. ∎

Lemma 7 $\omega_\tau(A^i_n D^j_{n+i}) \leq \omega_\tau(D^j_n A^i_{n-j})$.

Proof:

$\omega_\tau(A^i_n D^j_{n+i}) = \omega_\tau(A^i_n) + \omega_\tau(D^j_{n+i}), \omega_\tau(D^j_n A^i_{n-j}) = \omega_\tau(D^j_n) + \omega_\tau(A^i_{n-j})$

If $\omega_\tau(A^i_n) \leq \omega_\tau(A^i_{n-j})$ and $\omega_\tau(D^j_{n+i}) \leq \omega_\tau(D^j_n)$, lemma 7 would be correct.

Let $f = \omega_\tau(D^j_n) - \omega_\tau(D^j_{n+i}) = \frac{j}{n(n+j-max(n,2j))} - \frac{j}{(n+i)(n+i+j-max(n+i,2j))}$

If $n \geq 2j, (n+i) \geq 2j$, then $f = \frac{1}{n} - \frac{1}{n+i} = \frac{i}{n(n+i)} \geq 0$

If $n \geq 2j, (n+i) < 2j$, then $f = \frac{1}{n} - \frac{j}{(n+i-j)(n+i)}$. Let $f'(= ((n+i-j)(n+i)) - nj)$ be the numerator of f.

$$f' > ((n+i-j)2j) - nj = j(2n+2i-2j-n) = j(n+2i-2j) > j(2j+2i-2j) = j(2i) \geq 0$$

If $n < 2j, (n+i) \geq 2j$, then $f = \frac{j}{(n-j)n} - \frac{j}{j(n+i)}$. Let f' be the numerator of f.

$$f' = j(n+i) - (n-j)n = jn + ij - n^2 + jn = 2jn - n^2 + ij > n^2 - n^2 + ij \geq 0$$

If $n < 2j, (n+i) < 2j$, then $f = \frac{j}{(n-j)n} - \frac{j}{(n+i-j)(n+i)}$. Let f' be the numerator of f.

$$f' = (n+i)(n+i-j) - n(n-j) = ni + (n+i-j)i > 0$$

$$\therefore \omega_\tau(D_n^j) \geq \omega_\tau(D_{n+i}^j) \tag{20}$$

Similarly, let $f = \omega_\tau(A_{n-j}^i) - \omega_\tau(A_n^i) = \frac{i}{min(n-j,i)(n+i-j)} - \frac{i}{min(n,i)(n+i)}$

If $i \geq n \Rightarrow i > (n-j)$, then $f = \frac{i}{(n-j)(n-j+i)} - \frac{i}{n(n+i)}$. Let f' be the numerator of f.

$$f' = (n(n+i)) - ((n-j)(n-j+i)) = nj + (j(n+i-j)) > 0$$

If $i < n, i \geq (n-j)$, then $f = \frac{i}{(n-j)(n-j+i)} - \frac{i}{i(n+i)}$. Let f' be the numerator of f.

$$f' = (i(n+i)) - ((n-j)(n-j+i)) = i^2 + ij - (n-j)^2 > i^2 + ij - i^2 = ij \geq 0$$

If $i < n, i < (n-j)$, then

$$f = \frac{1}{n-j+i} - \frac{1}{n+i} = \frac{(n+i-(n-j+i))}{(n-j+i)(n+i)} \geq 0$$

$$\therefore \omega_\tau(A_{n-j}^i) \geq \omega_\tau(A_n^i) \tag{21}$$

From Equation 20 and 21, lemma 7 is correct. ▮

Lemma 8 The shortest path of $\omega_\tau(r(n,d,a))$ is $\langle A_n^a D_{n+a}^d \rangle$.

Proof: Proof is the same as lemma 4 except the representation of cost symbol. ▮