1-1-2000

Trackless on-line paging and computer memory management

Edward Benjamin Mikhalkov
University of Nevada, Las Vegas

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

TRACKLESS ON-LINE PAGING AND COMPUTER MEMORY MANAGEMENT

by

Edward Benjamin Mikhalkov

Bachelor of Science (Honours)
University of British Columbia, Vancouver, Canada
1997

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2000

UNLV

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

February 16, 2000

The Thesis prepared by

Edward Benjamin Mikhalkov

Entitled

Trackless On-Line Paging and Computer Memory Management

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

PR: 1017-53/1-00

ABSTRACT

Trackless On-Line Paging and Computer Memory Management

by

Edward Benjamin Mikhalkov

Dr. Wolfgang W. Bein, Examination Committee Chair
Assistant Professor of Computer Science
University of Nevada, Las Vegas

We consider existing research methodology for dealing with competitiveness analysis of on-line algorithms as well as introduce some newer analysis techniques for testing research hypotheses. Paging is discussed in general and current algorithms are surveyed and analyzed.

We also present a new randomized on-line algorithm for the 2-page cache problem that matches the lower bound, and, therefore, is optimal. The algorithm uses fewer resources than currently known algorithms for the same problem, and is, therefore, an improvement on existing results. Experimental findings for this new algorithm are also presented and analyzed.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my thesis supervisor Dr. Wolfgang Bein whose wise guidance and helpful comments made creation of this document possible. My parents Valentina and Benjamin Mikhalkov deserve the greatest praise for supporting me all along the way, and without whom none of this would have happened nor made sense. I would also like to express very warm thanks to my significant other Anna Maria who made my life so much happier during these long months and whose company encouraged my advancement.

In addition, my thanks go to the National Supercomputing Center for Energy and the Environment and the U.S. Department of Energy for allowing me to use their facilities.

CHAPTER 1

PRELIMINARIES

The problem we are going to investigate deals with paging, and in particular with paging in caches of size 2. Moreover, we are going to address this question from the trackless randomized on-line algorithms' point of view. Paging in general is a very interesting problem and has been studied fairly extensively. The problem that paging deals with exists in many other environments where there is a limited amount of a certain resource. One has to design a fair strategy that achieves maximum efficiency under the restrictions forced onto the system by the amount of the limited resource. We are going to describe paging in the first section of this chapter. In the next section we shall introduce the notion of an on-line algorithm, and how it deals effectively with paging. Finally, in the third section of this chapter we shall describe the idea of tracklessness, and how it is applicable to the paging problem. Then we shall proceed to tackle the randomized 2-page cache problem using the techniques described.

Paging

There are many situations when one has to achieve as good a performance as possible and yet one does not have all the resources provided. In case of paging this problem has the following nature. For the CPU to effectively process information the informaton must be readily available. However, most of the information resides at places where it is possible to store large amounts of data, but the speed of access from such locations is not as high as one would desire. It may be that the data have to

be fetched from computer's main memory depository which is RAM which is capable of storing large amounts of data but is rather slow to access. It can also be that the data are to be fetched from a hard drive and be brought into the memory. The speed of access to the hard drive is even slower than to RAM. Also, it may be that the files, or some portions of files have to be brought from across a network to a particular computer, or even from the Internet. In all these cases we obviously would like to have the data we are going to need to be the closest to the CPU while we know that amount of such data is limited.

This problem can be modeled by using the following paradigm. We introduce the notion of a *cache*, which is very fast memory, but quite expensive, and therefore has limited size. We also introduce the notion of *slow memory*, which requires more time to access, but is relatively cheap, so one can store large amounts of information in it.

Consider a computer system with the memory structure as described above. Both types of memory will be further subdivided into fixed-sized units, called pages. Let the number of such units in the fast memory be k, and the number of such units in the slow memory be N. As determined by the description of sizes of both types above, it is clear that $k < N$. Initially, when the fast memory is empty, the system fetches the information from the slow memory as pages are needed. If certain pages are found in the cache, the system reads them from the cache instead of the main memory, thus improving performance. When the cache becomes full, and the new pages become needed, one is faced with the choice of which pages to evict from the cache. Clearly, one would want the pages that are going to be needed the most to always be in the cache, and those that will be needed the least to be evicted with higher probability. In fact, Belady [4] proved that the algorithm that evicts the page, which is going to be needed in the future the latest, has optimal performance. However, since the

future sequence of requests is usually not known, we must make a decision without the knowledge of what a pattern of requests is going to be.

This problem has been addressed, and there are several algorithms that deal with this issue. We briefly describe them here, and then present the empirical results on their performance determined experimentally by Young [7].

- LRU (LEAST RECENTLY USED). This method evicts a page, which has been last used the earliest.
- CLOCK (CLOCK-REPLACEMENT). Similar to LRU, but instead of the actual time of reference, the algorithm only maintains one bit of information on how recently a page was used.
- FIFO (FIRST-IN/FIRST-OUT). This algorithm simply evicts a page that has been fetched the earliest from all the pages currently in the fast memory.
- LIFO (LAST-IN/FIRST-OUT). Evicts a page that has been fetched last.
- LFU (LEAST FREQUENTLY USED). Replaces the page whose number of requests has been the lowest.
- LFD (LONGEST FORWARD DISTANCE). This will replace the page whose request will come the latest in the sequence of future requests. It is an off-line algorithm as the knowledge of future requests is necessary.

The graph below shows relative performances of the algorithms above for different cache sizes. They have been first obtained by Young [7].

Now we prove that the LFD algorithm is optimal, and no other algorithm can do better than LFD with any sequence of requests. The original proof is due to Belady [4]. There are a number of modified proofs currently known, and here we present one such proof. This version is due to Borodin and El-Yaniv [5].

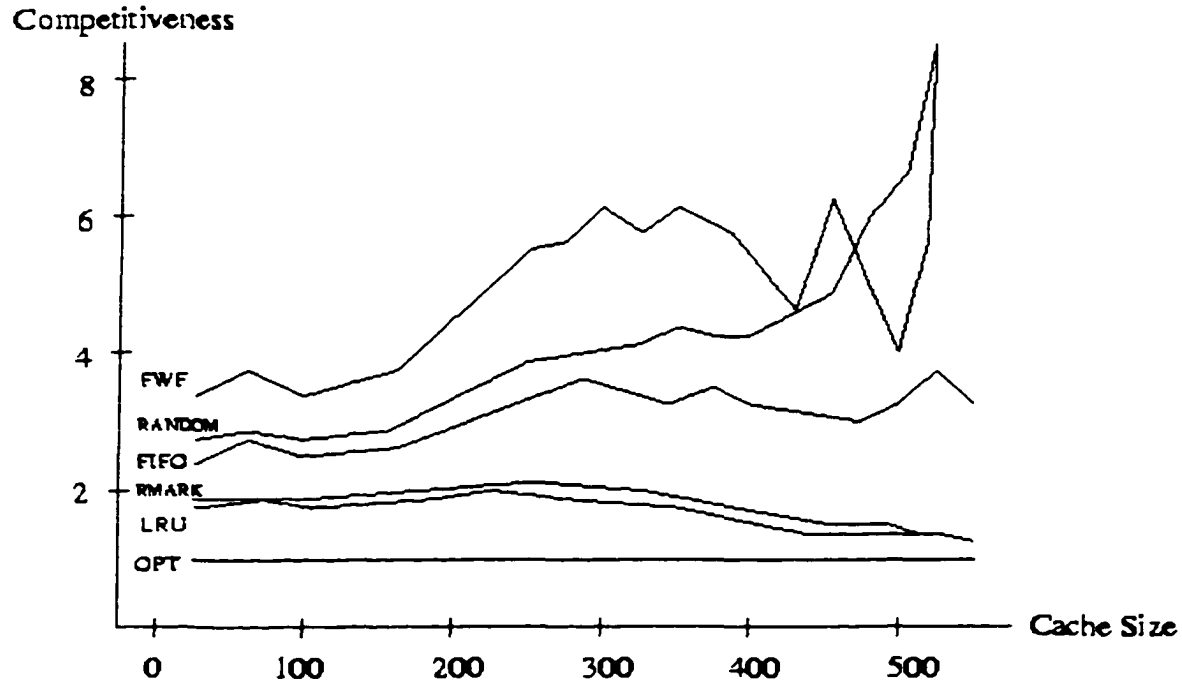


Figure 1.1: Competitive ratios of different algorithms with respect to the optimal off-line algorithm as a function of cache size.

Theorem 1 *Longest Forward Distance (LFD) is an optimal off-line algorithm.*

Proof: We shall show that any optimal off-line algorithm and LFD can be modified to behave equivalently without decreasing the performance of the optimal algorithm. To show that we shall need to use the following lemma.

Lemma 1 *Let ALG be any paging algorithm. Let ρ be any request sequence. It is possible to construct for any $i, i = 1, 2, \ldots, |\rho|$ an off-line algorithm ALG_i so that the following three properties will hold: (i) ALG_i processes the first $i-1$ requests exactly like ALG does; (ii) if the i^{th} request results in a page fault, then ALG_i will evict the page from the cache that has the longest forward distance, and if constructed in this way, then it will also hold that (iii) $ALG_i(\rho) \leq ALG(\rho)$.*

Proof: Suppose that X is the set of $k-1$ pages that have been fetched from the slow memory during $k-1$ steps by both ALG and ALG_i. In other words, the $k-1$ pages

are common to both caches. Since the algorithms are not necessarily identical, there will be a step at which different pages will be evicted to make room for the requested page, and after that the algorithms behave the same again. Let the page kept by ALG be v, and the page kept by ALG_i be u. If v is requested on subsequent input, then ALG_i will incur a page fault, but ALG will not. However, we know what since ALG_i evicts pages with the longest forward distance, at the time when v is requested, there must have been a request for u, at which point ALG would have had a page fault. Therefore, the number of page faults would be equal for both algorithms. Furthermore, when v is requested, it is fetched into both the cache serviced by ALG and ALG_i, so the two algorithms identify. $\diamond$

Now that we have the proof of the lemma, we can easily prove the theorem. By considering any input sequence ρ as a number of subsequences at the end of which the algorithms identify, it is easy to see that if the end of ρ falls on the end of one of such subsequences, the algorithms will be exactly equivalent in efficiency. If the end of ρ happened anywhere between the ends of subsequences, then the number of page faults incurred by ALG_i is strictly less than those incurred by ALG. $\diamond$

Having proved the theorem, we can use this result for the analysis of the problem from the point of view of on-line algorithms. In the next section we briefly describe what on-line algorithms are, and how we can use them to analyze the behavior of the 2-page cache.

On-Line Algorithms

On-line algorithms deal with situations in which the knowledge of future input is unknown, yet the algorithm must come up with the most efficient method of acting upon the input that may occur. Naturally, the question arises: how can one compare

the performance of two such algorithms if until the very end of the execution it is not clear what the next input item will be, and therefore each algorithm has potentially the chance of outperforming the other one. In their paper on analyzing lists, Sleator and Tarjan [6] have introduced the concept of *competitiveness*, which represents a measure of how a particular algorithm compares to an optimal algorithm for the same problem. More precisely, competitiveness is defined as follows.

Definition 1 *Let $\mathcal{A}$ be any algorithm and σ be a sequence of requests that $\mathcal{A}$ services. Let OPT be an optimal algorithm for a given problem. Then algorithm $\mathcal{A}$ is C-competitive if the following inequality holds:*

$$\mathcal{A}(\sigma) \leq C \cdot OPT(\sigma) + \mathcal{K},$$

where $\mathcal{K}$ is some constant. If $\mathcal{K} = 0$ then we say that the algorithm is strictly competitive.

There are two large classes of on-line algorithms. Algorithms in the first class specify a concrete set of steps that will be taken when dealing with making decisions on which pages to evict. They are called *deterministic*. Algorithms in the second class, however, do not fix a rigid strategy, but rather give a probability that a certain decision will be made concerning a specific page. Such algorithms are called *randomized*. The ways of specifying how and when such decisions are made vary from one method to another, but the concept of being able to make different choices in the same situation given a certain probability remains.

For the analysis of on-line algorithms it is often helpful to represent the execution of a particular algorithm as a game between two players. One player is referred to as the *algorithm* and the other is called the *adversary*. The adversary knows the strategy of an algorithm and can choose the input sequence in such a way that the competitive ratio C is maximized. We shall cover this approach in more detail in the chapter on proof techniques.

For the algorithms in the first class it has been proved that the upper bound is $\frac{k}{k-h+1}$, where k is the size of the algorithm cache, and h is the size of the optimal cache. Since in this case both k and h are equal to 2, the upper bound reduces to 2. For details of the proof refer to Borodin and El-Yaniv [5]. The same source presents a proof that the lower bound for deterministic algorithms is k. That means that no deterministic algorithm for a cache of size 2 can achieve better performance than $2 \cdot OPT(\sigma)$ for any σ. Therefore, we can conclude that all existing optimal deterministic algorithms for 2-page caches are 2-competitive.

However, if we analyze randomized algorithms, we notice that the performance can be further improved even though it does not guarantee that such performance can be achieved at all runs of the algorithm. Borodin and El-Yaniv [5] give a lower bound of $\mathcal{H}_k$ on the performance of randomized algorithms, where $\mathcal{H}_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$ is the k^{th} harmonic number. Therefore, if we can produce an algorithm that will be $\mathcal{H}_k$-competitive, that will mean that the algorithm is optimal for the randomized model of computation and for the cache size of 2.

The Trackless Approach

As we have discussed, an on-line algorithm cannot base its performance on the future input because it is unknown to it. However, there has been nothing said about what can be inferred about the knowledge that has been accumulated in the past. For some algorithms we shall allow for such a possibility, and indeed it is often beneficial to have access to the past requests, responses as well as the eventual outcome of a certain strategy. In many environments, however, it is impractical, if not impossible, to keep track of all the actions taken by an algorithm and/or adversary in the past. For example, one may be able to maintain information about the status of the pages in the cache, or the sequence of the last k requests, but it is clear that if one were to keep track of everything that has occurred, or the status of all the

pages in main memory, for example, then such a task would be quite demanding in terms of time and space. For example, given the highly volumous nature of the slow memory, which for practical purposes can be even considered unbounded, it is clear that trying to maintain the information about pages in slow memory would be either totally impossible or simply not feasible.

Therefore, it makes sense to introduce the concept of *tracklessness*. It was first introduced by Bein and Larmore [3] and later used in a number of other research projects. In the trackless paradigm of algorithm's behavior we have the following participants. Just like in other cases of on-line algorithm analysis, we have the adversary that tries to come up with the worst possible input scenario to maximize the competitiveness ratio C. We also have the algorithm itself whose task is to keep the competitiveness ratio as low as possible. And finally, we have a so-called *referee* that uses some function $\mathcal{F}_t(x_1, \ldots, x_t, y_1, \ldots, y_t)$ of all available inputs $x_1, \ldots, x_t$ and all outputs $y_1, \ldots, y_t$ produced so far to generate the new input α_t, which is then passed on to the algorithm at each time step t. The relationship among the algorithm, adversary and the referee is represented by the diagram in Figure 1.2.

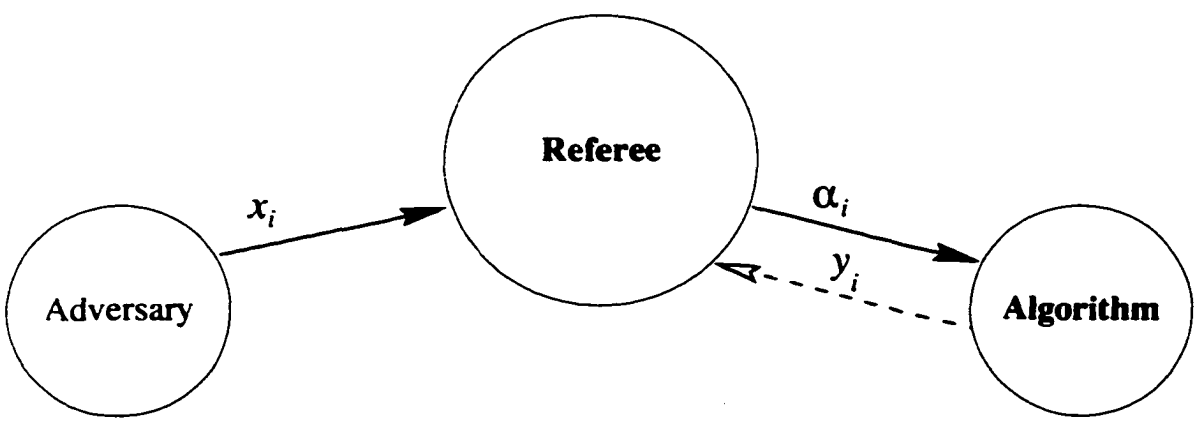


Figure 1.2: Illustration of the Trackless Model of Computation.

CHAPTER 2

PROOF TECHNIQUES

Game Graphs

We are going to introduce some proof techniques that will be used in proving the upper bound of our $\frac{3}{2}$-competitive algorithm. Most of these concepts deal with Graph Theory and Game Theory results and enable us to perform analysis of our algorithm by methods that other techniques do not provide.

First we define a *game graph*. A *game graph* is a weighted directed graph $\mathcal{G}$ whose nodes are partitioned into 2 sets. The nodes in one set are called the *red* nodes, and the nodes in the other set are called the *black* nodes.

The edges of the graph $\mathcal{G}$ are also divided into two sets. There are *red* edges and *black* edges. The color of the edge is determined by the node that is its source. So for an edge (x, y) the color is black only if the node x is black. Otherwise the edge is red.

The game on such a graph proceeds as a sequence of moves of two players that we shall refer to as Red and Black. The current position is specified by a "pebble" that is at some node at any point in time. If the pebble is at a Red node, then it is the turn of the Red player to make a move, and if it is at a Black node, then the Black player is to make a move next.

Each move has a cost associated with it. If the cost is positive, the Red player "pays" the Black player the amount specified as the weight of the edge. If the cost is

negative that means that the Black player pays the Red player the amount equal to the absolute value of the weight of the edge.

We say that the Black player wins the game if Black can force Red to pay an unbounded amount. That is, if, for any constant $\mathcal{K}$, Black can force Red to pay more than $\mathcal{K}$, then we say that Black wins. Red wins if Black cannot force it to pay more than $\mathcal{K}$ for some $\mathcal{K}$. We say that Black has a winning strategy if Black can play in such a way that Black always wins, and Red has a winning strategy if Red can play in such a way that Red always wins.

Sometimes whether or not Red can win may depend on the starting node. In these cases we specify if the start node makes a difference. We say that *Red can win* if Red can win regardless of which start node was selected. We say that *Red can win with start node x* if the start node is specified.

Next we define game potentials and show how they are helpful in proving competitiveness of algorithms.

Definition 2 *Let $\mathcal{G}$ be a game graph. A game potential $\Phi(\mathcal{G})$ for a graph $\mathcal{G}$ is a real-valued function Φ defined for all nodes of $\mathcal{G}$ that satisfies the following properties:*

- *If x is a black node and (x, y) is an edge, then $\Phi(x) \geq \Phi(y) + cost(x, y)$.*
- *If x is a red node, then there exists a node y and an edge (x, y) such that $\Phi(x) \geq \Phi(y) + cost(x, y)$.*

Furthermore, we define a *critical cycle* to be a cycle of the graph that can be forced by the Black player, and for which the inequalities above hold as strict equalities.

By results from game theory, we know that Red can win if and only if the graph $\mathcal{G}$ has a potential. For details refer, for example, to Borodin and El-Yaniv [5] or to Sleator and Tarjan [6]. Next we show how we can use these results.

Competitiveness Proofs Using Game Graphs

Now, suppose we are given a game graph $\mathcal{G}$ in the Figure 2.1. The Red player will represent possible algorithm moves, while the Black player will represent the adversary moves. The solid circles and solid arrows labeled b_1, b_2 represent black nodes and black edges, while the dashed circles and arrows labeled r_1, r_2 represent red nodes and edges respectively.

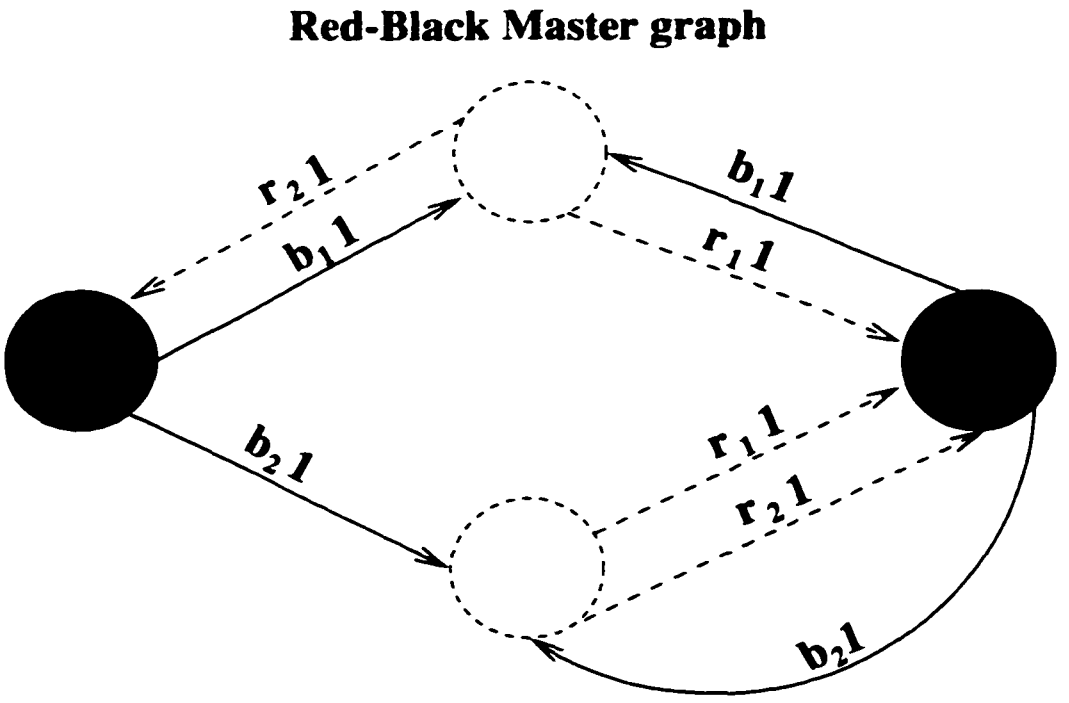


Figure 2.1: A Red-Black Master graph for an algorithm-adversary game.

Suppose we consider a particular algorithm, which we shall call $\mathcal{A}$ that follows some consistent strategy. For example an algorithm that always responds with r_2. Then we postulate a certain competitiveness constant that we suspect is the right number for this algorithm. We then try to prove it by checking if the inequalities above hold. If they do, but no cycle gives a strict equality, that means that the value we suspected to be the competitiveness constant is simply an upper bound, which can be possibly improved further. Then we decrease the suspected value and check

the inequalities again. In this case we shall show that the algorithm $\mathcal{A}$ is no worse than 3-competitive.

First of all, since the responses of the algorithm are fixed for every move of the adversary, the edges labeled anything else other than r_2 are useless since according to the strategy of the algorithm they will never be followed. Therefore, the transitions described by these edges are not reachable. So it is now possible to construct a graph that describes fully the behavior of this particular algorithm and adversary moves, since the algorithm is known to follow a certain strategy. We construct such a reduced Red-Black and it is shown in the Figure 2.2. Note that there are no transitions labeled r_1 for the algorithm, but the adversary kept all its transitions. This is because we can restrict algorithm moves by assigning a certain strategy to it while adversary moves can never be restricted.

In this example we shall assume that the cost of each transition is 1. Normally, the actual costs are given in the statement of the problem when a particular algorithm is considered.

Since every transition has a cost associated with it, and we have a value that we suspect to be the competitiveness, it is now possible to label the edges. The cost of each transition of the algorithm is 1, so we put 1 as the value along each red edge. Now we recall that the postulated competitiveness of this algorithm is 3, therefore, the algorithm is allowed to be within a factor of 3 away from the optimal algorithm. So if the adversary issues a certain request, and the optimal algorithm pays 1, then for the game graph to accurately represent this situation the corresponding edge must be labeled $-3 \cdot 1$. The minus sign represents the fact that the adversary pays the algorithm. In general, if the conjectured competitiveness of an algorithm is C and the cost of serving the request to the optimal algorithm is $cost_{OPT}(r)$, then the corresponding black edge representing such a transition is labeled $-C \cdot cost_{OPT}(r)$.

The costs to the algorithm and to the adversary are shown as labels on the edges in the Figure 2.2.

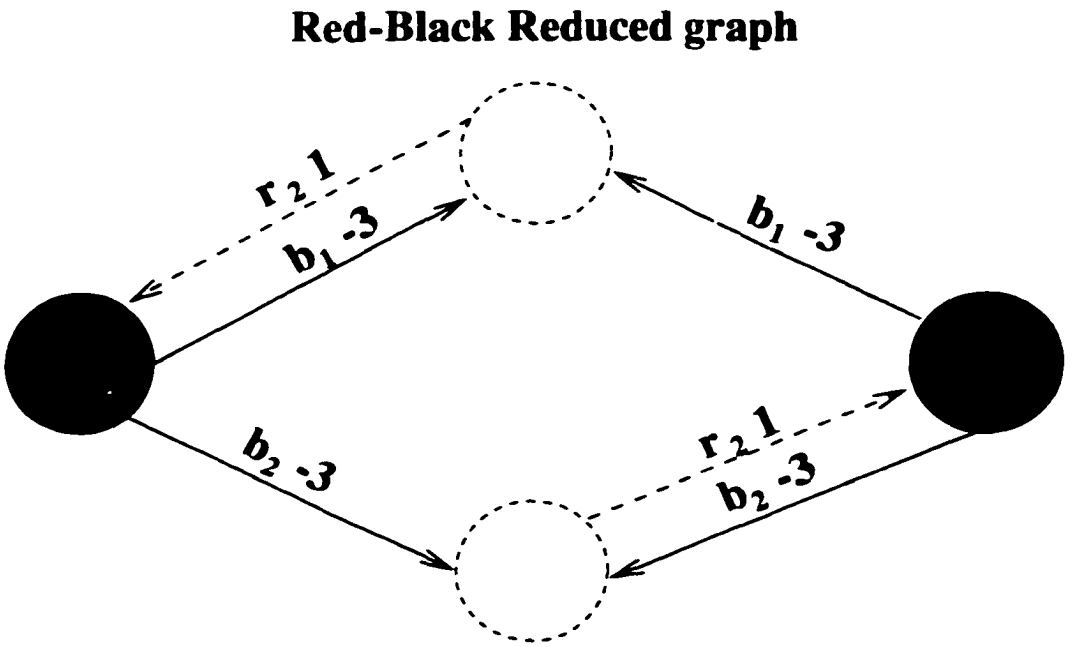


Figure 2.2: Reduced Red-Black graph for the algorithm $\mathcal{A}$.

Next, we further observe that once a black edge has been selected, the moves of the game are determined entirely until the next black node is reached. This is what we expect because the algorithm can do only one thing at every step, namely what its strategy specifies. Therefore, all moves are now completely determined by the adversary, and the game proceeds by moving among the black nodes. The intermediate red nodes do not contribute anything to the description of the game, since they are completely fixed, and each pair of edges to and from the red nodes can be regarded as one edge from one black node to another. It is also possible to calculate the cost of such an edge by adding the costs of the black and the red edges.

Following this method, we convert the Red-Black graph in Figure 2.2 to a completely Black graph shown in the Figure 2.3. It is important to remember that this graph is still a Red-Black graph, however, and that we simply chose to omit the

red transitions as they are completely pre-determined. We shall show how results obtained for such Black graphs are also valid for the original Red-Black graphs that they were derived from.

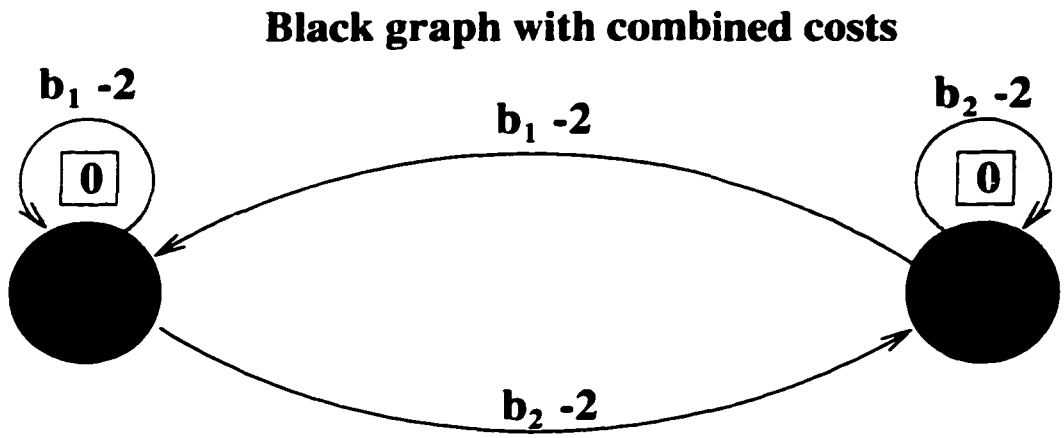


Figure 2.3: Black graph for the algorithm $\mathcal{A}$ with potentials.

Now that we have a graph that is minimal and also carries all the information about the algorithm, we are ready to conduct the competitiveness analysis. The costs of all edges are given, and it remains to determine whether there exist such potential values that make the inequalities hold. By starting with a value 0 for the potential of the first black node, we quickly convince ourselves that the value of 0 for the second node satisfies both inequalities. The values of the potential function are shown in squares in the Figure 2.3.

Since the inequalities are satisfied for all transitions in the Black graph, we are tempted to conclude that the algorithm is indeed no worse than 3-competitive. This turns out to be the case, but there is more analysis required to prove this. The difficulty is in the fact, that the upper bound is confirmed if the potential function $\Phi(\mathcal{G})$ is determined for all the nodes of the Red-Black graph $\mathcal{G}$. In this case we have only obtained the potential function for the nodes of the Black graph. We quickly

show. however. that this is only a slight difficulty and can be easily overcome with a following argument.

Suppose we are given a transition between two black nodes with a total cost of $cost_t$. This cost was obtained as the sum of the costs of the two transitions from the black to an intermediate red node, and from the red to the final black node in the original Red-Black graph. Therefore. $cost_t = cost_b + cost_r$. where $cost_b$ is the cost of the original black-red transition and $cost_r$ is the cost of the original red-black transition. We claim that is the inequalities hold for all the nodes of the graph. that is if the potential function $\Phi(\mathcal{G})$ is defined for all the nodes of the Black graph. then it is also defined for all the nodes of the original Red-Black graph.

Consider a black-black transition shown in Figure 2.4. Let the potential value at the first node be α and the potential value at the second node be β. We also know that the inequality $\alpha \geq \beta + cost_t$ holds for all the transitions in the graph. We are to show that it is always possible to find such a value γ that the potential value for the red node. for which both inequalities will also hold.

Let γ be the potential of the intermediate red node. Set $\gamma = \beta + cost_r$. Then for the red-black transition we trivially have $\gamma \geq \beta + cost_r$ satisfied. Next we need to show that for the black-red transition $\alpha \geq \gamma + cost_b$ is also true. Since $\gamma = \beta + cost_r$ we substitute its value into the inequality obtaining $\alpha \geq \gamma + cost_b = \beta + cost_r + cost_b = \beta + cost_t$ which is certainly true because it has been already shown for all the nodes in the black graph.

Therefore. by proving that a potential function exists just for the black graph such a construction confirms that the conditions for the potential function are satisfied for the original Red-Black graph. which in turns means that the postulated upper bound of the algorithm is valid. The graph for our algorithm with potentials for both red and black nodes is shown in the Figure 2.5.

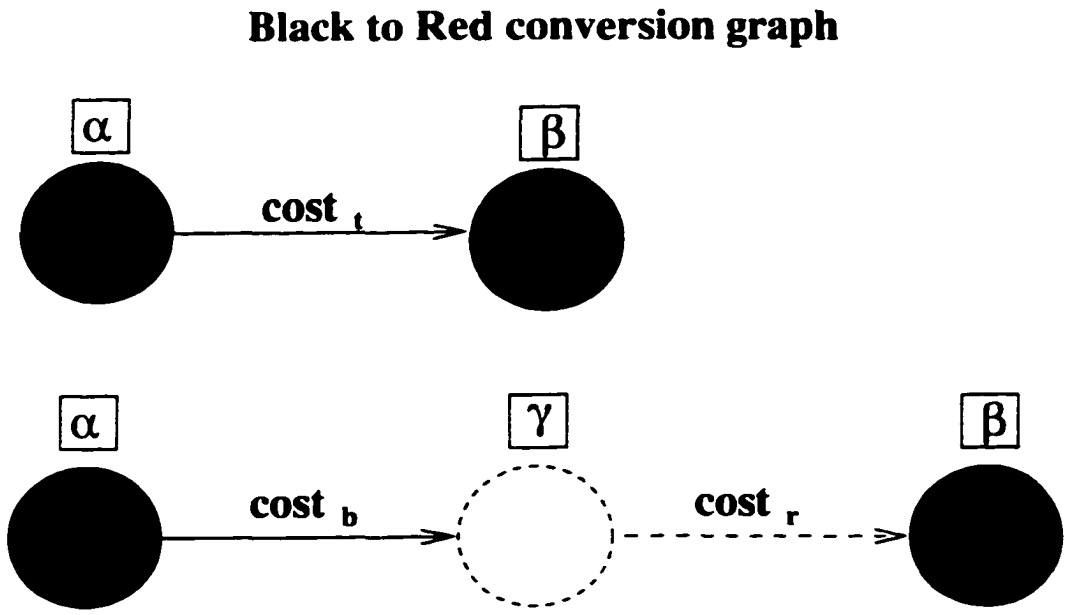


Figure 2.4: Conversion between Black graph and the original Red-Black graph.

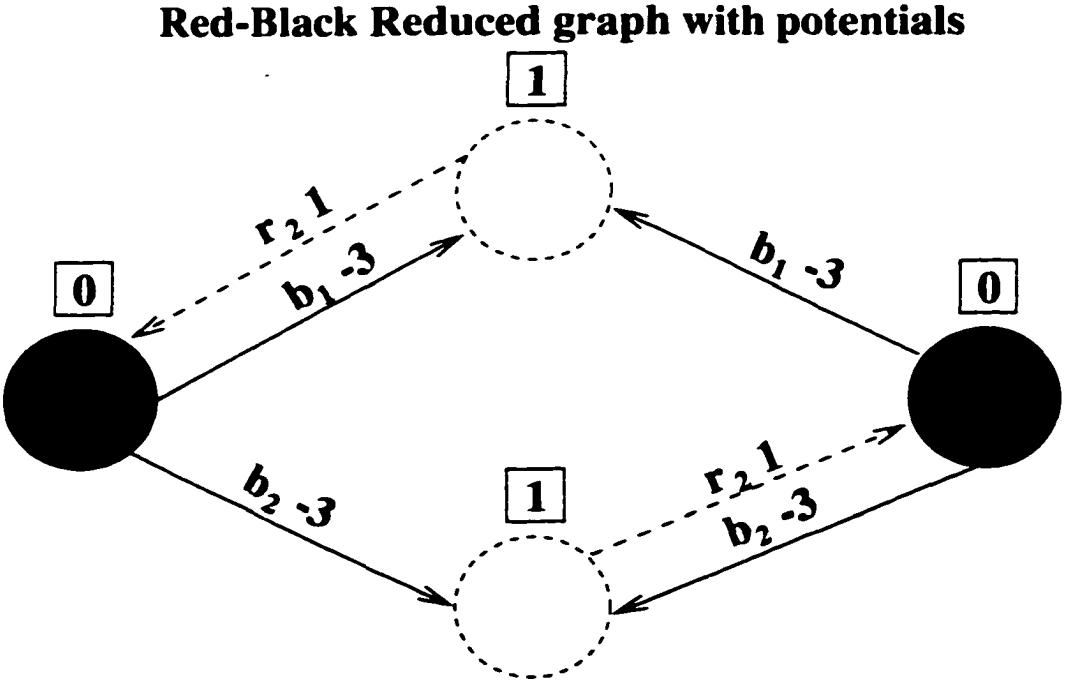


Figure 2.5: Reduced Red-Black graph for the algorithm $\mathcal{A}$ with potentials.

So we have seen how the technique of introducing potentials for algorithm description allows us to arrive at the potential competitiveness value, and how to verify whether or not such a value is correct. If the upper bound found by such a method matches the known lower bound, we indeed can conclude that the found value is the true competitiveness as well as that the algorithm is optimal. We use these techniques to explain and analyze algorithms in the next chapter.

CHAPTER 3

THEORETICAL FOUNDATIONS

In this chapter we introduce the 1-Bookmark Algorithm and analyze its properties. The idea behind this algorithm is described in Bein and Larmore [2]. To facilitate this process, several intermediate results are presented.

Consider a two-page cache. Whenever a new page has to be brought in, there is always a choice as to where it will go. Once the decision has been made, the page currently at that location is to be evicted. Clearly, the two pages in the cache are not identical. Bringing in a second copy of a page that is already in the cache would be meaningless because the request to such a page could be served by the page already in the cache.

Furthermore, the two pages are also not identical in terms of their usage. At any given time one of the pages has been referenced less recently than the other, since the requests can only come through a single channel, namely the *referee*. To distinguish between the two pages, we shall call the one that has been referenced most recently the *junior* page, and the one referenced least recently the *senior* page.

The 1-bookmark algorithm also makes use of one extra location called *bookmark*. The page in this location can be either ignored or used in determining the next step of the algorithm.

Master Graph

To describe all the possibilities that can take place we introduce the following notation. The adversary can issue a request that can cause the optimal (off-line)

algorithm to evict its junior page, that is such a page would be the longest in the future sequence of requests. We shall call such a request a. The adversary can also request a page that will cause the optimal algorithm to evict its senior page. We shall call such a request b. Finally, the adversary can request a page that will be a 'Hit' to a page in the cache of the optimal algorithm. Without loss of generality we consider that such a request is to the senior page which we denote by s. The reason is that the junior page is the same in the cache of any algorithm, since this page has to be brought in to serve the current request. Therefore, requests to such a page would not contribute anything to the cost of any algorithm that keeps the last requested page in its cache.

As described in the trackless model of computation, once a request is issued by the adversary, the referee determines whether such a page is present in the cache of the algorithm. Depending on whether the algorithm has the knowledge of the requested page or not, the referee can produce the following input to the algorithm:

- If the requested page is a 'Miss' to the algorithm, then the referee issues m.
- If the requested page is a 'Hit' to the algorithm, then the referee issues h.
- If the request is to a page that is not in the cache, but is available in the bookmark location, then the referee issues p.

Finally, the algorithm can produce the following responses to the referee:

- Bring the new page in and replace the junior page. The bookmark is not changed. This is denoted by a.
- Bring the new page in, and replace the junior page. Also, bookmark the evicted page by putting it in the bookmark location. This is denoted by $a+$.
- Bring the new page in and replace the senior page. The bookmark is not changed. This is denoted by b.

- Bring the new page in, and replace the senior page. Also, bookmark the evicted page by putting it in the bookmark location. This is denoted by $b+$.
- Serve the request with the senior page. Don't bring anything into the cache, nor change the bookmarks. The senior page becomes the junior page, and the junior becomes the senior. This is denoted by s.

Given the above requests and responses, one can produce a graph that describes all possible actions by the adversary as well as by any algorithm. Such a master graph is shown in the Figure 3.1. This corresponds to the graph in Figure 2.1 in the general example given in the preceding chapter.

The black dots represent the pages in the cache of the optimal algorithm. The light small empty circles represent the pages in the cache of the algorithm being analyzed. The large black and light circles enclose the junior pages of the optimal and the algorithm being analyzed respectively. The light squares represent the bookmark location of the algorithm. If the square is drawn around a page, it means that the page is currently bookmarked by the algorithm. If the square is empty, it means that the bookmark may or may not contain any pages, but the algorithm cannot make use of that information. Finally, if the dots and the circles are drawn next to each other, it means that they represent the same page. If they are drawn away from each other, they represent different pages.

The execution proceeds as follows. Upon receiving adversary's request, the optimal algorithm decides which page is to be evicted (if any), and communicates a, b, or s to the referee. Based on the information about the algorithm cache, the referee issues m, h or p. These actions are represented by black edges and their labels. The first letter of the label is the action of the optimal algorithm, and the second is the input communicated to the algorithm by the referee. The algorithm in turn determines which pages are to be evicted (if any) based on its current strategy. Having

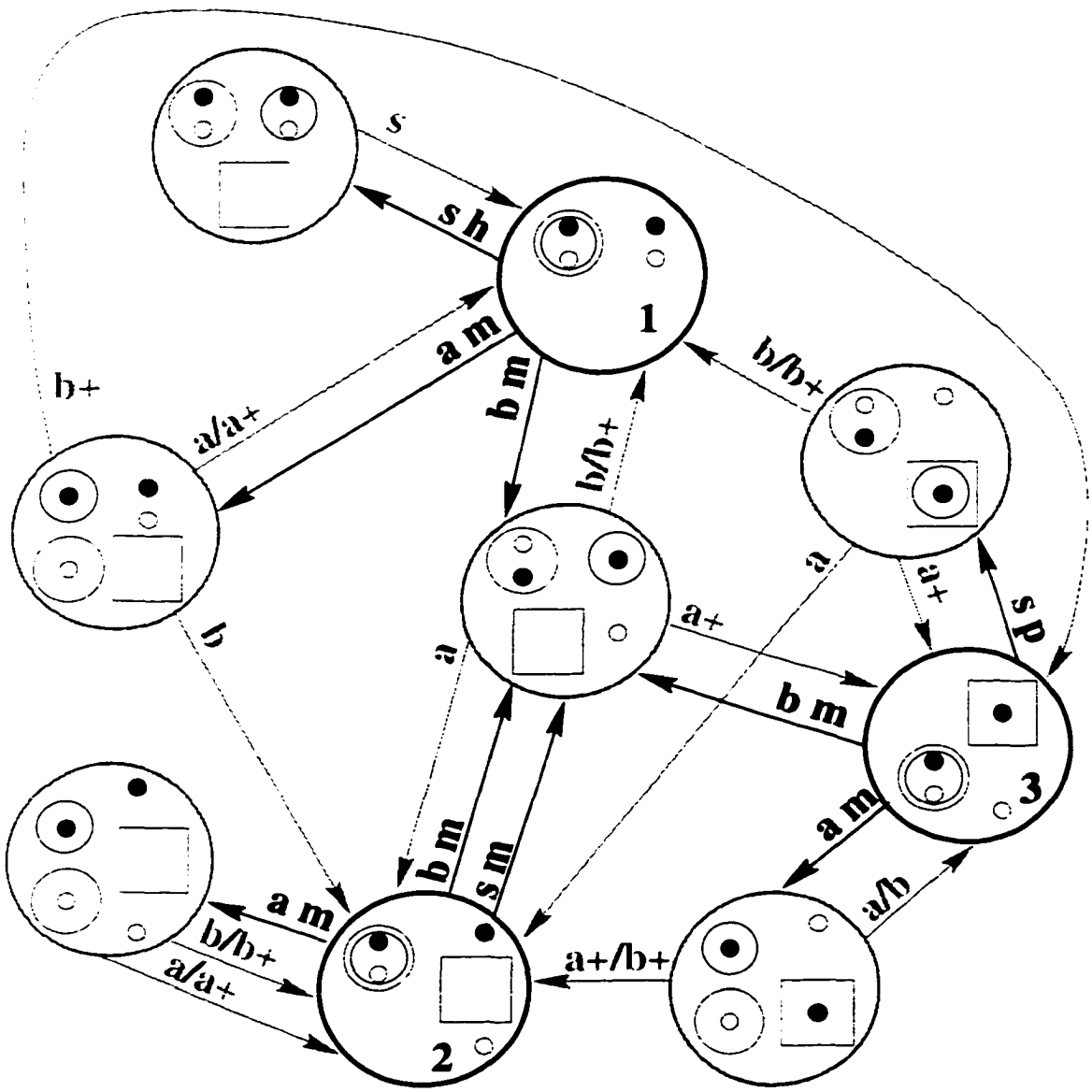


Figure 3.1: State machine for all possible adversary and algorithm moves.

made the decision the algorithm issues a response to the referee and goes to the state specified by a light arrow. The labels on the light arrows mean algorithm's responses.

Barely Random Algorithms

The master graph in the Figure 3.1 describes behavior of all possible adversarial requests and actions of all possible algorithms. We, however, are interested in a specific class of algorithms, namely *barely random* algorithms.

Definition 3 *An algorithm $\mathcal{A}$ is barely random if for any input sequence ρ it executes only one of $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, and the decision which algorithm to execute is a probability distribution of a finite number of deterministic algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$. That is the algorithm "flips a coin" a finite number of times, to decide which $\mathcal{A}_i$ to use once and for all for the duration of the game.*

We now describe two deterministic algorithms that will be used to construct a barely random algorithm.

The first algorithm ejects the least recently used page most of the time. There are, however, times when it ejects the most recently used page. The decision about which page to evict at any particular step is made according to an internal state machine. The machine consists of 3 states and specifies which transitions are to be made, and which page is to be evicted. The states are SATISFIED, BOOKMARKED and UNSATISFIED. When the state is SATISFIED and the input from the referee is m, the algorithm ejects the most recently used page and bookmarks it. That makes it similar to the MRU algorithm, so we shall call this algorithm Most Recently Used Modified, or MRUM for short.

The second algorithm maintains the same internal states as the first algorithm. However, it ejects the least recently used page at all times. Therefore, this algorithm is effectively LRU, so we shall refer to it as LRU.

The complete behavior of both algorithms is described by the graphs in the Figures 3.2 and 3.3. Here, $a+$ and $b+$ mean that the pages evicted are bookmarked, and p means that the request is to the bookmarked page.

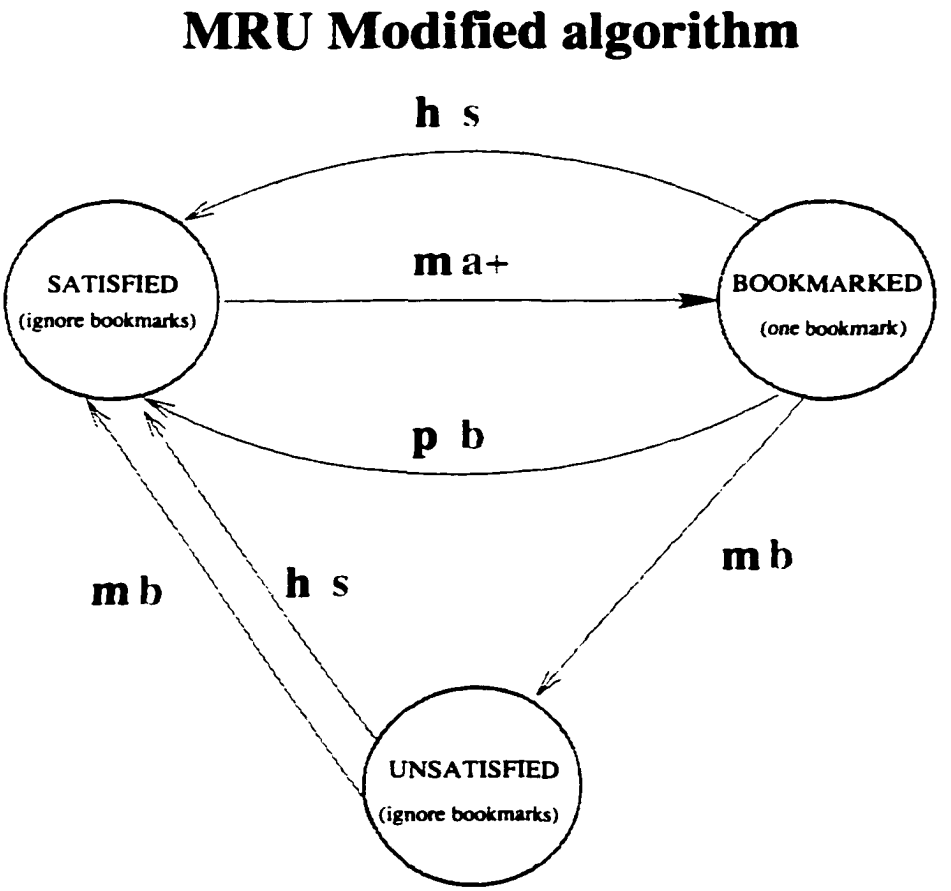


Figure 3.2: State machine for algorithm moves under MRUM.

Note that technically the LRU algorithm can be phrased with only one state and no bookmark. However, for our analysis, it will be useful to have the same three states for LRU as for MRUM. In fact, we make the following observation. Define the *ken* of an algorithm to be the set of pages which are either in the cache or bookmarked. Note that cardinality of the ken of both algorithms MRUM and LRU is 2 if the state is SATISFIED or UNSATISFIED, and 3 if the state is BOOKMARKED. In fact,

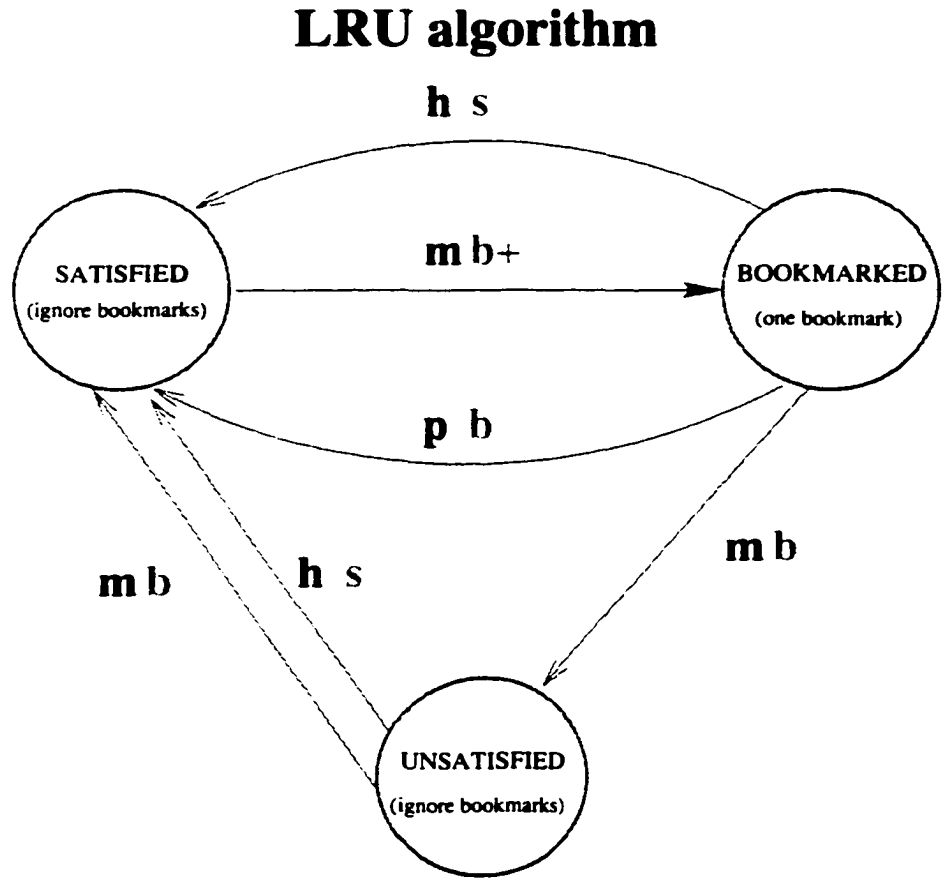


Figure 3.3: State machine for algorithm moves under LRU.

although algorithms MRUM and LRU differ, they have the same ken and state at every iteration.

Next, we are going to construct a barely random algorithm that will combine both LRU and MRUM. The combined algorithm will execute only one of them for a specific input sequence. The decision which algorithm to execute will be made only once and it will be done randomly with probability of each being selected equal to $p_1 = p_2 = \frac{1}{2}$. This decision will remain valid for the duration of each and every sequence for the entire game.

Combined Algorithm

The master graph specifies behavior of any algorithm given referee input h, m, or p. For a specific algorithm, the response to the referee input is fixed. Therefore, as every move of the algorithm can be unambiguously determined at each step, the behavior of the optimal algorithm is also fixed for the same input sequence. For example, consider the algorithm MRUM. Suppose the optimal algorithm is in the state 1 (refer to Figure 3.1), and MRUM is in the SATISFIED state. The adversary issues a command, which causes the optimal algorithm to evict its junior page. Then the black edge labeled am is traversed, and m is issued as the referee input to the algorithm. Then we consult the internal state machine of the algorithm (refer to Figure 3.2) and observe that upon encountering m in the UNSATISFIED state, the algorithm ejects its junior page and bookmarks it. That means that after serving such a request the optimal algorithm is in state 1, and MRUM is in the BOOKMARKED state. We denote this by a state that we call B1.

Similarly, the adversary can issue a request that will cause the optimal algorithm to evict its senior page. Then the black edge labeled bm is traversed. Whenever in the SATISFIED state, the algorithm replaces its junior page and bookmarks it. Thus, we traverse the red edge labeled $a+$. That means that after serving such a request the optimal algorithm is in state 3, and MRUM is in the BOOKMARKED state. We denote this by a state B3.

Finally, if the adversary issues a request that results in a 'Hit' to the senior page in the optimal cache, then the black edge labeled sh of the Master graph is traversed, and h is issued as the input from the referee. The algorithm responds by also encountering a 'Hit' to its page, because the optimal and the algorithm caches are the same in the SATISFIED state. Therefore, the light edge labeled s is traversed, and the systems finishes in state 1 for the optimal algorithm and state SATISFIED for the MRUM. We denote this by a state S1, which is also the start state.

Following the same strategy one can produce the graph that describes the states of the optimal algorithm as well as of MRUM. Note that for each of the 3 states of the optimal algorithm, MRUM can be in one of its 3 internal states. Therefore, we would expect the total number of states in the resulting graph to be the cross product of the states of both algorithms, which is 9. That is indeed the case as can be seen from the graph in Figure 3.4. The graph for LRU is constructed similarly, and is shown in Figure 3.5. These graphs correspond to the Red-Black graph in Figure 2.2 in the general example.

The black letters in the graph specify the requests of the adversary, the black numbers are the costs of serving that request to the optimal algorithm, and the light numbers are the costs incurred by the algorithm being described.

We notice, however, that the states S2 and S3 are essentially the same, except for the position of the bookmark. Since the states S2 and S3 both correspond to the SATISFIED state of the algorithm, and in the SATISFIED state bookmarks are not used, the states S2 and S3 represent effectively the same state, so can be safely collapsed into just one state S2.

Similarly, the states U2 and U3 also differ only in the position of the bookmark. Since in the UNSATISFIED state position of the bookmark is not important, these two states can also be safely collapsed into one state U2.

The two reduced graphs for both MRUM and LRU are shown in the Figures 3.6 and 3.7.

To produce a barely random algorithm we make two steps. Instead of choosing just one algorithm randomly and finding its competitiveness, we construct an algorithm that executes both LRU and MRUM simultaneously. The cost incurred by such an algorithm at each step is the sum of the costs of the corresponding steps of both algorithms. By proving the competitiveness of this combined algorithm, one can then deduce the result for the case when only one algorithm is executed. The graph

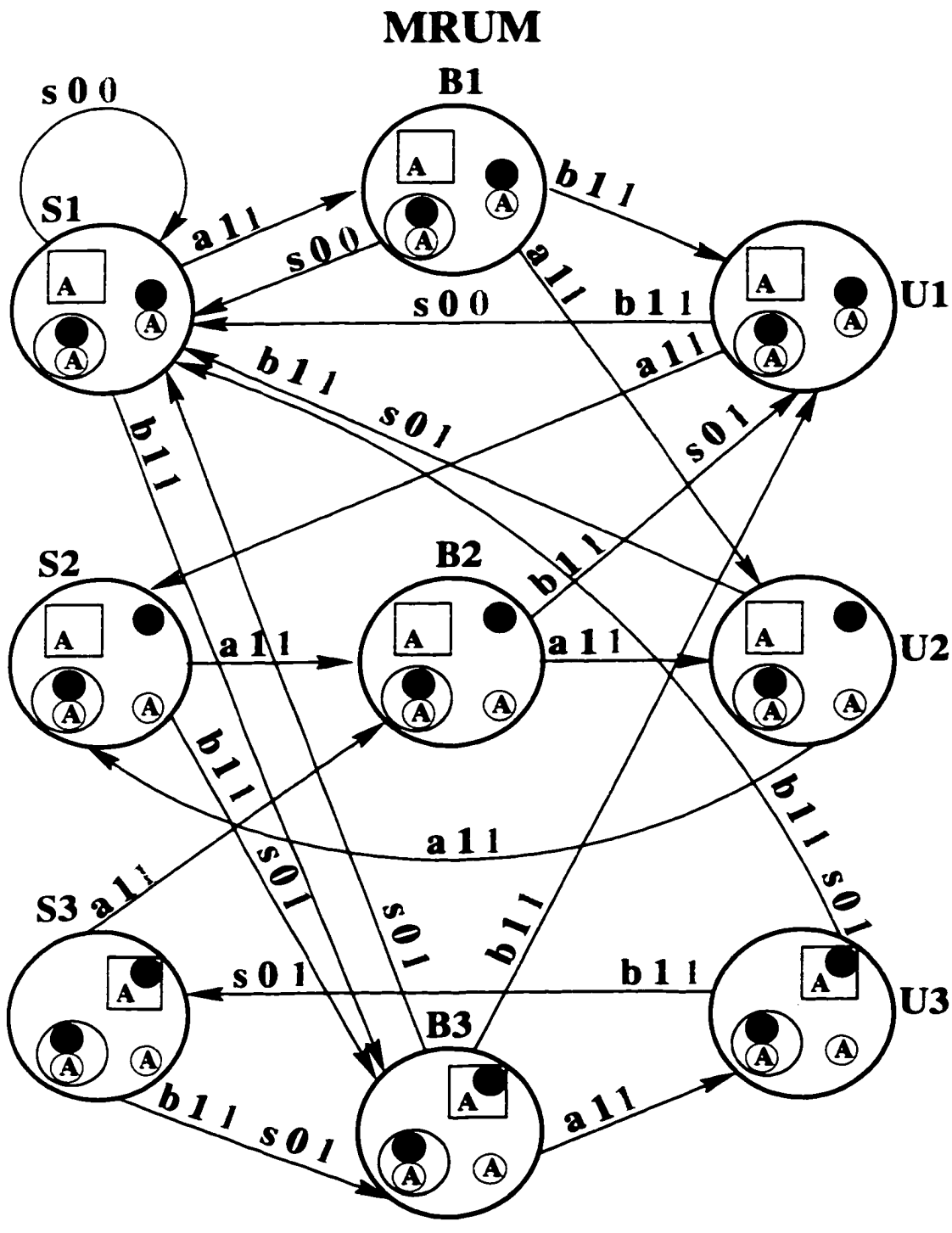


Figure 3.4: State machine for transitions under MRUM.

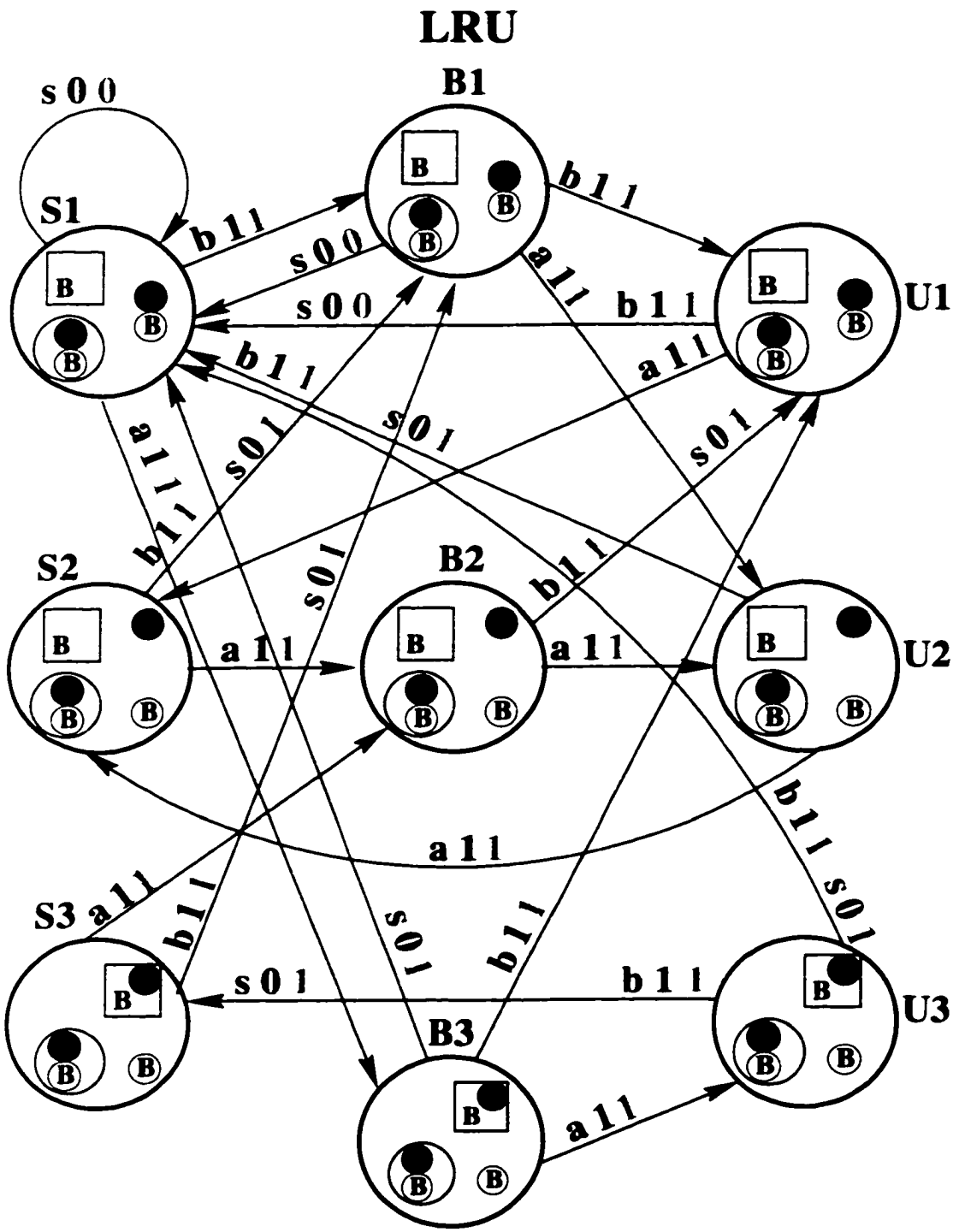


Figure 3.5: State machine for transitions under LRU.

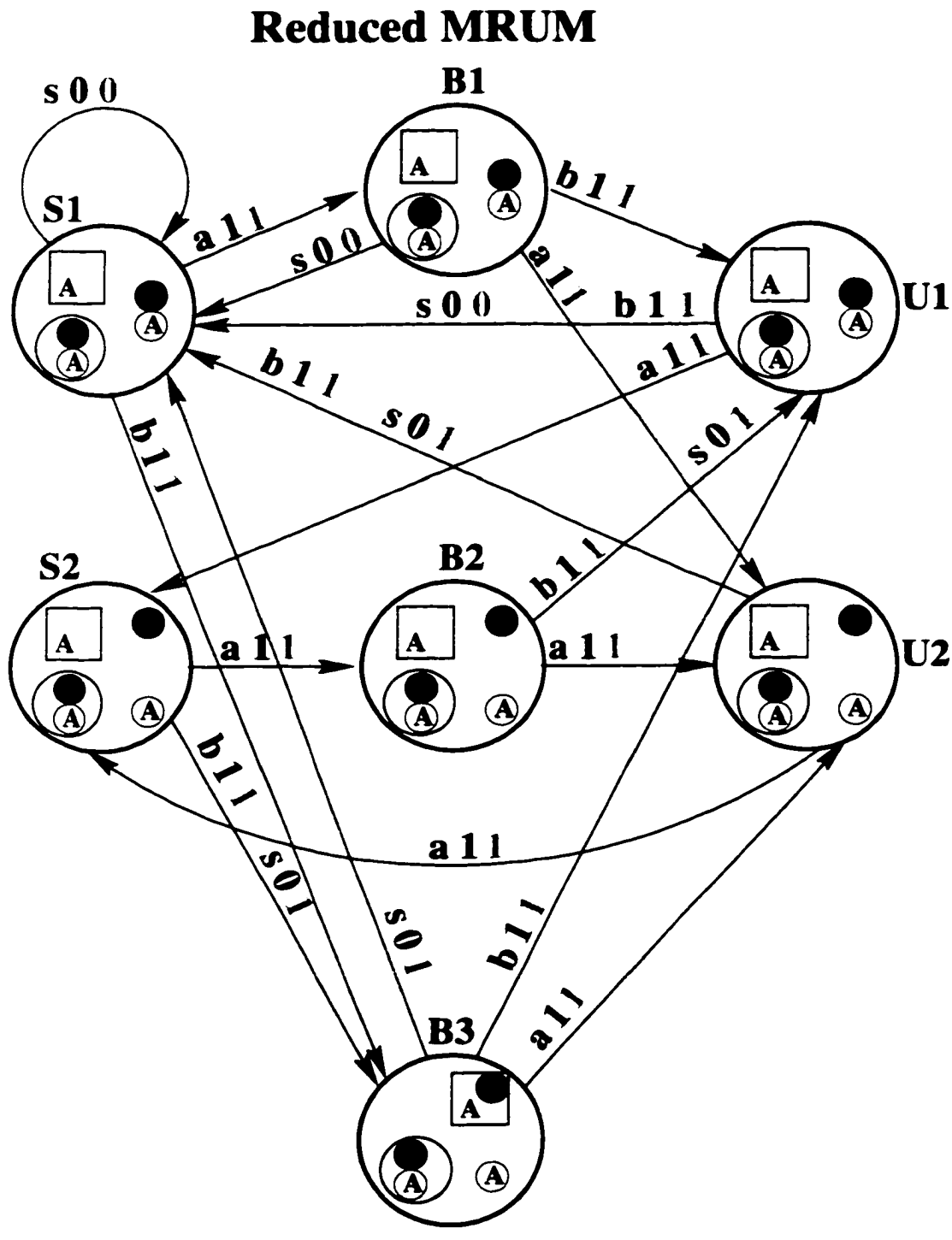


Figure 3.6: Reduced state machine for transitions under MRUM.

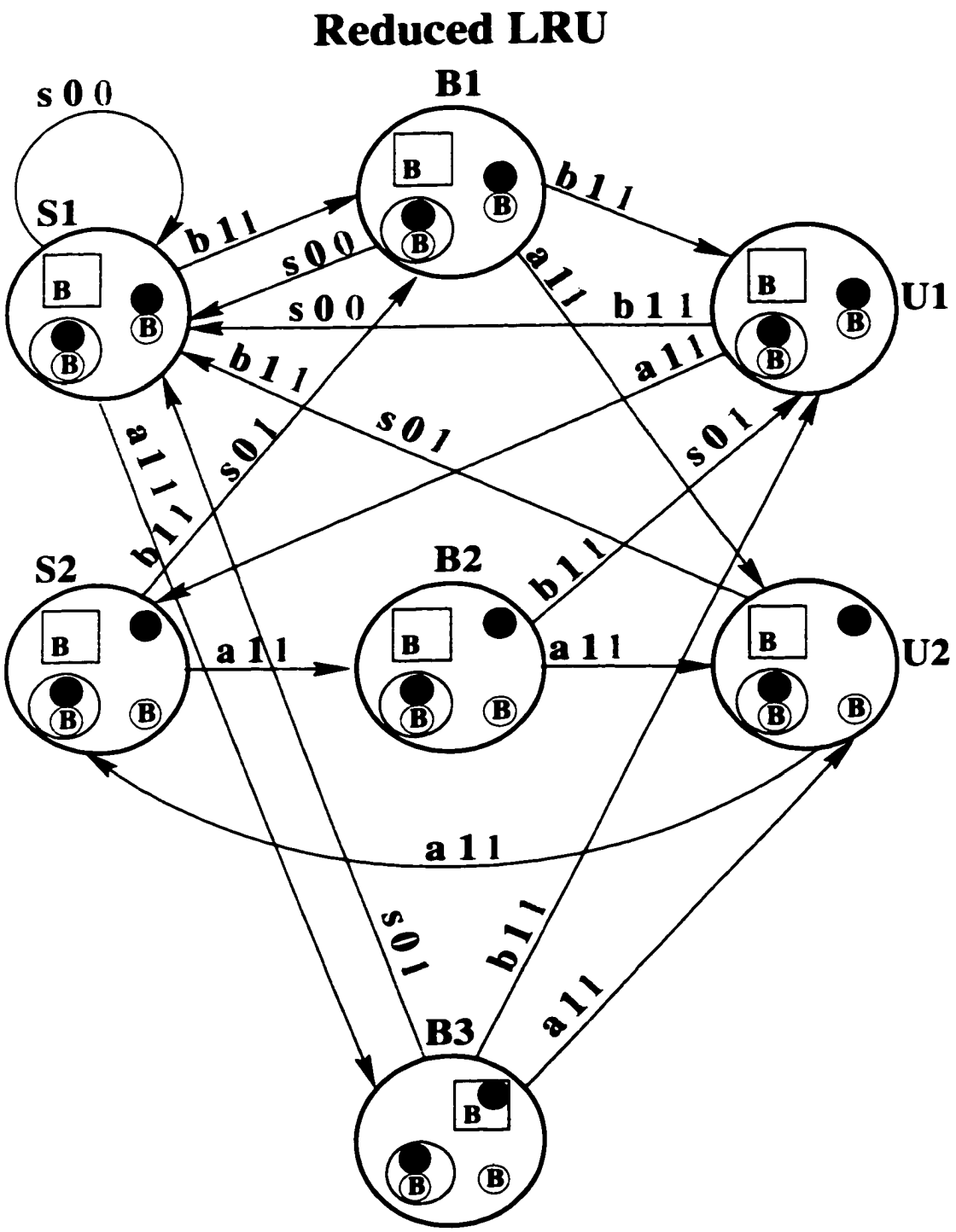


Figure 3.7: Reduced state machine for transitions under LRU.

representing the behavior of the combined algorithm is shown in the Figure 3.8. Note that the states B1, B2 and B3 of the combined algorithm are not the same as the states B1, B2, B3 of the original algorithms, but are rather the combined states that encapsulate corresponding states of the original algorithms.

The combined graph is arrived at in the following fashion: The adversary issues a request that causes the optimal algorithm to evict its junior page, that is the edge labeled *am* will be traversed. We consult the reduced graphs for both algorithms to determine which states those algorithms will finish in. This request leads to the state B1 for MRUM and the state B3 for LRU. By combining the states we produce a new node of the combined graph and label it B1. One could expect the number of nodes in the resulting graph to be the cross product of the states in both reduced graphs. That would mean the total of 49. However, because of the observation we have made earlier that the ken and the states of both algorithms are always the same, the final graph will also have only 7 states. We observe that to be the case.

The black letter above each edge specify the request of the adversary, the black number shows the cost of serving the request by the optimal algorithm, and the light number shows the combined cost of both algorithms executing the request.

Competitive Analysis

At this step we are ready to conduct competitive analysis. To enable us to do that we need to employ several methods that have been mentioned in the chapter on proof techniques. One such method is *potential functions*. By assigning potentials to each of the nodes of a graph, and by using a specific value for the competitiveness one can prove that if that competitiveness indeed satisfies the potential function, such a competitiveness is indeed valid.

The proof consists of two parts. We need to prove that $\frac{3}{2}$ is the upper bound first. We shall do this by using the Black graph constructed using the techniques

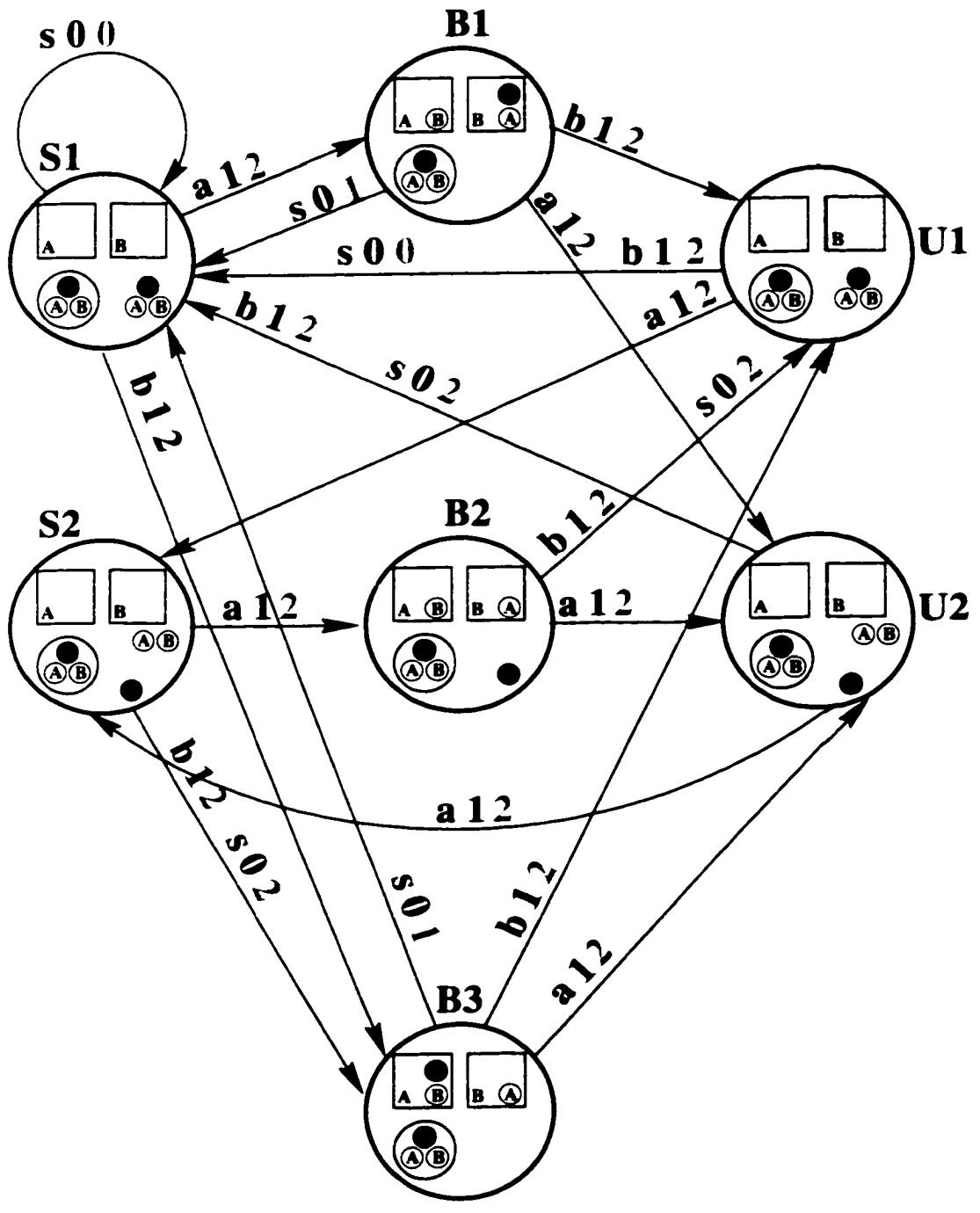


Figure 3.8: Reduced state machine for transitions under MRUM and LRU combined.

already explained. To show that it is optimal we also must prove the same value for the lower bound. However, in this particular case, we already do know that the lower bound is $\mathcal{H}_k = \frac{3}{2}$.

To prove the upper bound we need to show that the algorithm we are analyzing can always perform within the boundary of the competitiveness constant for *any* adversary.

So far we have only considered adversaries who will issue the worst possible requests at each step. This strategy is obviously sufficient to ensure lower bound. However, since we want to prove the upper bound, it is necessary to amend the graph to include the moves that most competitive adversary may not use. We shall call such moves *silly moves*. Silly moves correspond to the cases when adversary issues requests that result in the algorithm incurring a 'Hit' and paying nothing while the optimal algorithm incurs a 'Miss' and pays to bring the new page in. Such moves are represented by added edges in the Figure 3.9, and we shall refer to such edges as *silly edges*. We shall show that addition of the silly edges to the graph does not affect the competitiveness.

The graph in Figure 3.9 is now complete in that it has all the possible moves not only of the algorithm, but also of the adversary. Therefore, we can conduct analysis of both the lower and the upper bound on it. However, we note again that the lower bound of $\frac{3}{2}$ is already known so it only remains to show that the upper bound is also $\frac{3}{2}$ to produce an optimal algorithm.

To test the expected competitiveness it is necessary to assign a potential to each node. These values are deduced experimentally in order to ensure the potential function is satisfied for the upper bound. The graph in the Figure 3.10 shows the values of the potential function at each node. By applying these potential values one can confirm that having the potential value of 3 indeed satisfies the conditions of the potential function.

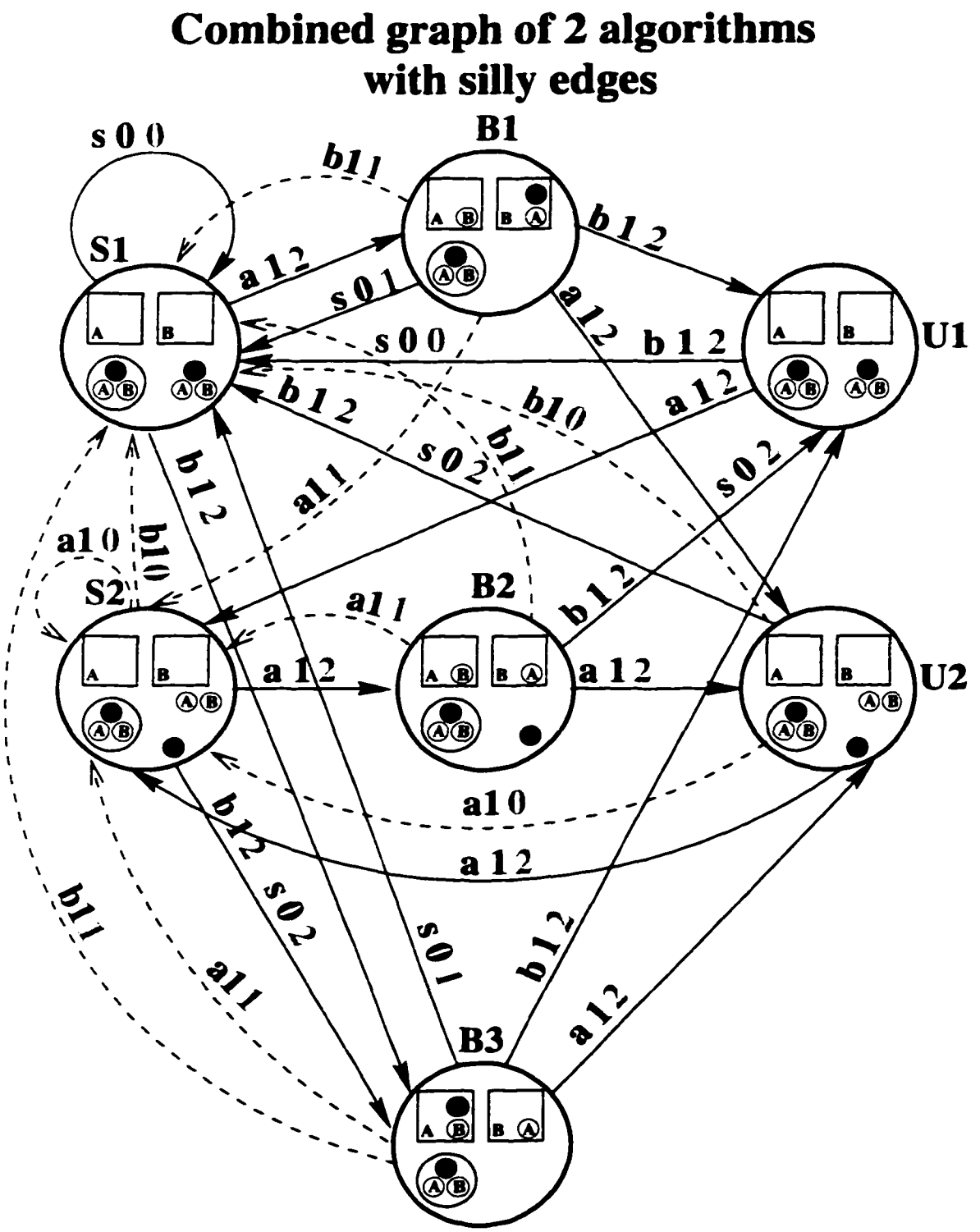


Figure 3.9: The combined graph with silly edges.

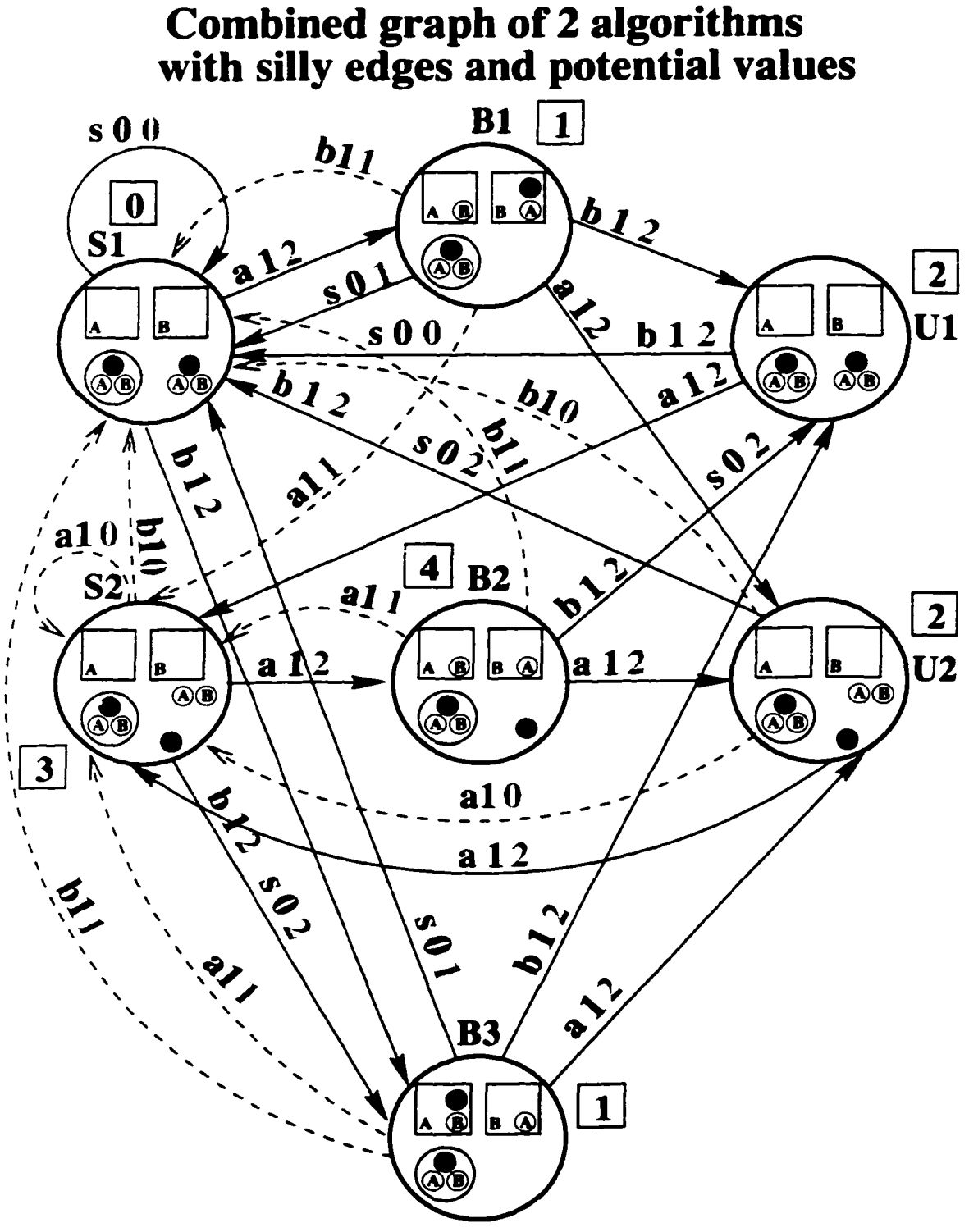


Figure 3.10: The combined graph indicating potential values.

Having proved that the competitiveness of the combined algorithm is 3, we need to show that the randomized version of this algorithm has competitiveness of $\frac{3}{2}$. We do this in the following theorem.

Theorem 2 *If the combined algorithm has competitiveness of 3, then the algorithm that only executes one of LRU and MRUM has the competitiveness of* $\frac{3}{2}$.

Proof: Let ρ be some input sequence. Let $X_1 = cost_{LRU}(\rho)$, $X_2 = cost_{MRUM}(\rho)$ and $Y = cost_{OPT}(\rho)$. Then $X_1 + X_2 \leq 3Y$ as proven above by producing a potential function for the Black graph.

Let p_1 be the probability that one algorithm is run and p_2 the probability that the other algorithm is run with $p_1 + p_2 = 1$. Furthermore, assign $p_1 = p_2 = \frac{1}{2}$. Let the barely random algorithm that only runs LRU or MRUM be $\mathcal{O}$, and let $\mathcal{Z} = cost_{\mathcal{O}}(\rho)$ for some input string ρ. Then the expected value of $\mathcal{Z}$ is going to be $\mathcal{Z} = \frac{1}{2}X_1 + \frac{1}{2}X_2 \leq \frac{1}{2}(3Y) = \frac{3}{2}cost_{OPT}$. Therefore, $\mathcal{O}$ is $\frac{3}{2}$-competitive. $\diamond$

We have used the black dots, circles and squares within the nodes to demonstrate the behavior of both the optimal and our algorithm at each move. However, now that the graph is constructed, they are no longer necessary, so we shall give a more abstract representation. The graph in Figure 3.11 shows the nodes, the transition edges and the potential values at each node. This is similar to the Black graph in the Figure 2.3 in the general model. The labels along each edge specify what request the optimal algorithm is serving, and the light number will be the real cost of serving the request to the algorithm. The real cost is calculated as $C \cdot cost_{OPT} - cost_A$ where A is our combined algorithm.

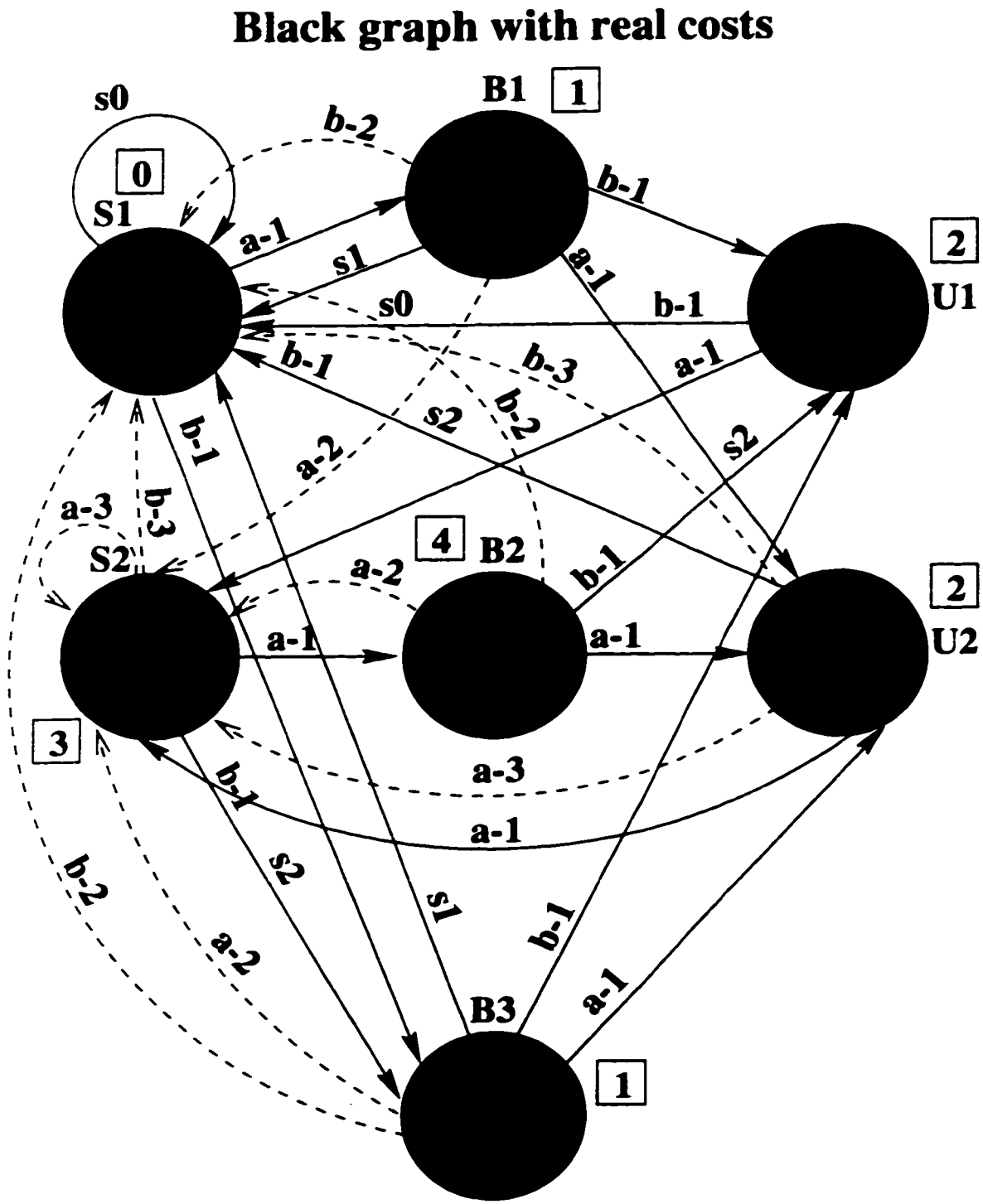


Figure 3.11: The Black graph indicating real costs.

CHAPTER 4

EXPERIMENTAL RESULTS

Having introduced theoretical basis for the algorithm's behavior in the previous chapter it would be interesting to see how our algorithm behaves in practice. We begin by first converting our algorithm into pseudocode. Then the algorithm is written as a software program and run with input strings of various lengths. Finally, we observe the output in terms of individual page locations as well as in terms of global efficiency by analyzing its performance over strings of different types and comparing it to the performance of currently known algorithms.

Let MRUM and LRU be the two algorithms described in the preceding chapter (refer to Figures 3.2 and 3.3). At step 1, one of the algorithms is chosen according to the *barely random* strategy described earlier. The chosen algorithm starts in the SATISFIED state, and proceeds according to the pseudocode specified in Figure 4.1. As we see, upon reading the next input request the algorithm first determines whether or not the request is to any of the pages known by the algorithms. Such requests can only be to the pages currently in the cache or in the bookmark location. Note that unlike in the original algorithm, where we simply ignore information about the bookmark if we are not in the BOOKMARKED state, here we actively erase the bookmarks when we leave the BOOKMARKED state. Since a bookmark whose value cannot be read, and a bookmark with an empty value are the same from the point of view of the algorithm, this does not affect functionality of the algorithm.

When request is indeed to an unrecognized page, then algorithm behaves differently in the SATISFIED state depending on which of MRUM and LRU was initially

```
1. Version is MRUM or LRU with probability 1/2;
2. State = SATISFIED;  book = EMPTY;
3. Read(r);
4. switch(r)
     case r = junior          /* request to junior cache page */
          no action
     case r = senior          /* request to senior cache page */
          book = EMPTY;
          State = SATISFIED;
     case r = book            /* request to bookmarked page */
          Eject senior; book = EMPTY;
          State = SATISFIED;
     else                     /* request is unrecognized */
          if State = SATISFIED
                              Version = MRUM: book = junior; Eject junior;
                              Version = LRU: book = senior; Eject senior;
                              State = BOOKMARKED
          if State = BOOKMARKED
                              Eject senior; book = EMPTY;
                              State = UNSATISFIED;
          if State = UNSATISFIED;
                              Eject senior; book = EMPTY;
                              State = SATISFIED;
5. goto 3;
```

Figure 4.1: Pseudocode description of the 1-bookmark algorithm.

chosen. In other states the behavior is the same which is to be expected given the descriptions of these algorithms in the preceding chapter.

Based on such input from the referee, the algorithm makes a decision and moves to another state updating its information in the process. This pseudocode directly reflects the algorithm described in the previous chapter.

We introduce one slight change to the algorithm. Instead of letting the algorithm randomly decide which of MRUM and LRU to execute for the duration of the entire input string, we allow that randomization be done for each of the input requests. That is equivalent to substituting a random algorithm in place of a barely random one. We shall call this algorithm RAN. The pseudocode for this algorithm is given below.

```
1. Version is MRUM or LRU with probability 1/2;
2. State = SATISFIED;   book  = EMPTY;
3. Read(r);
4. switch(r)
     case r = junior          /* request to junior cache page */
            no action
     case r = senior          /* request to senior cache page */
            book =  EMPTY;
            State = SATISFIED;
     case r = book            /* request to bookmarked page */
            Eject senior; book =  EMPTY;
            State = SATISFIED;
     else                     /* request is unrecognized */
            if State = SATISFIED
                             with probability 1/2 do MRUM:
                             book = junior; Eject junior;
                             with probability 1/2 do LRU:
                             book = senior; Eject senior;
                             State = BOOKMARKED
            if State = BOOKMARKED
                             Eject senior: book = EMPTY;
                             State = UNSATISFIED;
            if State = UNSATISFIED;
                             Eject senior; book = EMPTY;
                             State = SATISFIED;
5. goto 3;
```

Figure 4.2: Pseudocode for RAN.

The tables below represent execution of RAN for a randomly generated string of length 8, as well as behavior of LRU, MRUM and the OPT algorithms for the same input string. The information about bookmarks is shown for each algorithm and for each step, even though it is important to remember that the OPT algorithm does not use this information, and the other three algorithms only use it if they are in the SATISFIED state.

The leftmost column indicates which algorithm the cache is shown for. The columns titled with the bold letters **J**, **S** and **B** list the contents of the junior, senior and bookmark cache locations respectively. The execution proceeds left to right, top to bottom.

INPUT STRING = DOKPEZRH

ALG	J	S	B
REQUEST: D			
LRU	D	A	B
MRUM	D	B	A
OPT	D	A	A
RAN	D	A	B
REQUEST: K			
LRU	K	O	B
MRUM	K	O	A
OPT	K	A	A
RAN	K	O	B
REQUEST: E			
LRU	E	P	O
MRUM	E	P	K
OPT	E	A	A
RAN	E	P	O
REQUEST: R			
LRU	R	Z	E
MRUM	R	E	Z
OPT	R	A	A
RAN	R	E	Z
Cost	LRU: 8 MRUM: 8 OPT: 8 RAN: 8		

ALG	J	S	B
REQUEST: O			
LRU	O	D	B
MRUM	O	D	A
OPT	O	A	A
RAN	O	D	B
REQUEST: P			
LRU	P	K	O
MRUM	P	O	K
OPT	P	A	A
RAN	P	K	O
REQUEST: Z			
LRU	Z	E	O
MRUM	Z	E	K
OPT	Z	A	A
RAN	Z	E	O
REQUEST: H			
LRU	H	R	E
MRUM	H	R	Z
OPT	H	A	A
RAN	H	R	Z

Input String Design

We have seen what steps the algorithm makes for a string that has randomly generated characters from the input alphabet. However, this may or may not represent the true behavior of the algorithm in all situations, especially in the worst case scenario. To produce the results for all possible string types we address the following questions:

1. What would be the worst case for the algorithm?
2. What would be the best case for the algorithm?
3. What is the average case for the algorithm?

To answer these questions we need to look closer at what the algorithm does. The algorithm applies MRUM and LRU each with probability $\frac{1}{2}$. Therefore, the worst case sequence for the algorithm would be an input string that requests pages that the chosen algorithm decided to evict. For example, if MRUM was randomly chosen, then the worst case sequence would request the pages that MRUM had evicted. Similarly, if LRU was randomly chosen, then the worst case sequence would request the pages that LRU had evicted. However, the adversary does not know which of the two algorithms has been chosen, so it can only apply the worst case input string for one of the algorithms, or alternate between the two according to some strategy.

In order to do that the adversary must know what the worst sequences for MRUM and LRU would be. We consider this question next. One can deduce these strings by simply looking at the graphs specifying the behavior of both MRUM and LRU. We give more detailed analysis below.

We would like to construct a substring that when repeated would force the algorithm to evict the highest number of pages while the optimal algorithm would evict the least, hence maximizing the competitiveness ratio. Furthermore, we would like our substring to satisfy the following criteria:

1. The substring has to be the minimal possible length.

2. At the beginning and the end of the substring the contents of the algorithm's cache must be the same.

3. At the beginning and the end of the substring the internal state of the algorithm must be the same.

The first criterion is due to the fact that there may exist an infinite number of strings that force the algorithm to have its worst case behavior. To make comparison of all algorithms possible we must use the same input string for all of them. Since all such substrings would give the maximum competitiveness ratio, without loss of

generality we can choose one such string that has minimum length. Also, the adversary can potentially generate a very long input string consisting of all fresh requests up to the maximum number of elements in the request alphabet. However, such a string would cause page faults not only in the cache of the algorithm, but also in the optimal cache, so the competitiveness ratio would be minimized, therefore such a string would be meaningless as a worst case candidate.

The second and the third criteria ensure that repeating derived substring will guarantee the same behavior of the algorithm for the entire duration of the string. Consider, for example, a case in which the contents of the cache are the same at the end of the string but the internal state is not. In that case processing the same substring again the algorithm may demonstrate different behavior since it had started in a different internal state, so its transition diagram would be different. Analogously, if the algorithm is in the same internal state, but the contents of the cache are not the same, then instead of incurring a 'Miss' for a particular request the algorithm can incur a 'Hit' if that page happens to be in the cache for the current configuration.

We claim that the worst possible string for an algorithm consists only of requests, each of which results in a 'Miss' to the algorithm cache. This result is due to the following lemma.

Lemma 2 *Any worst case input string for an algorithm does not contain requests that result in a 'Hit' to the algorithm.*

Proof: By definition a worst case input string forces the competitiveness ratio to be maximized. Let $\mathcal{A}$ be some algorithm, and let ρ be its worst case sequence of length n. Let $cost_{OPT}(\rho)$ be the cost of serving the request sequence by the optimal algorithm, and $cost_{\mathcal{A}}(\rho)$ be the cost of serving the request sequence by the algorithm $\mathcal{A}$. Then the competitiveness ratio will be $C = \frac{cost_{\mathcal{A}}(\rho)}{cost_{OPT}(\rho)}$. Note that this is a strict equality since the sequence is the worst case sequence.

Let ρ' be another sequence of requests that contains a request that causes a 'Hit' to the algorithm cache. We show that ρ' cannot be a worst case sequence.

Assign 0 to be the cost of a request that causes a 'Hit' to the algorithm, and 1 if a request causes a 'Miss'. Write ρ as $\rho = \rho_1 r \rho_2$ and ρ' as $\rho' = \rho_1 r' \rho_2$, where $|\rho_1| < n$, $|\rho_2| < n$ and $|r| = |r'| = 1$. The request r causes a 'Miss' to the algorithm, and request r' causes a 'Hit'. Then the cost of serving ρ will be $cost_{\mathcal{A}}(\rho) = cost_{\mathcal{A}}(\rho_1) + cost_{\mathcal{A}}(r) + cost_{\mathcal{A}}(\rho_2) = cost_{\mathcal{A}}(\rho_1) + 1 + cost_{\mathcal{A}}(\rho_2) = n$ and the cost of serving $cost_{\mathcal{A}}(\rho') = cost_{\mathcal{A}}(\rho_1) + cost_{\mathcal{A}}(r') + cost_{\mathcal{A}}(\rho_2) = cost_{\mathcal{A}}(\rho_1) + 0 + cost_{\mathcal{A}}(\rho_2) = n - 1$.

Clearly, $cost_{\mathcal{A}}(\rho') < cost_{\mathcal{A}}(\rho)$. So the ratio $C' = \frac{cost'_{\mathcal{A}}(\rho)}{cost_{OPT}(\rho)} < \frac{cost_{\mathcal{A}}(\rho)}{cost_{OPT}(\rho)} = C$. Therefore, $C' < C$, so the ratio C' cannot be the competitive constant, which means that the request sequence ρ' that forces this ratio cannot be a worst request sequence.

We have handled the proof for a single request that results in a 'Hit'. Strings containing more than one such request can be shown not to be worst case by applying the above argument to each instance of such requests. $\diamond$

The next step is to construct such a sequence that would ensure a page fault on every request to the algorithm. We consider MRUM and LRU in turn.

To construct a sequence for MRUM that has the maximum number of page faults we consult the diagram of the algorithm in Figure 3.2. Since we want the algorithm to finish in the same state in which it started at the end of the substring, there are two possibilities. The adversary can issue a substring that will force the algorithm to go through the state transition SATISFIED $\longrightarrow$ BOOKMARKED $\longrightarrow$ SATIISFIED, or it can force it to go through the SATISFIED $\longrightarrow$ BOOKMARKED $\longrightarrow$ UNSATISFIED $\longrightarrow$ SATISFIED transition. We consider these situations in turn.

Since we are interested in the shortest possible substring one is to minimize the number of alphabet elements used to construct such a string. Therefore, we only

introduce new symbols when all other ones are in use. Initially, the cache starts with A as the junior page, B as the senior page and A in the bookmarked location.

We then add requests to the substring to force algorithm to have a page fault on each request. To ensure the transition to the BOOKMARKED state, a fresh page needs to be requested, say page C. Following that a 'Hit' to the bookmarked page is needed to ensure transition back to the SATISFIED state. The bookmark holds the location of the page A, so we request page A. However, the contents of the cache are not the same as they were at the beginning of the substring. Therefore, the sequence needs to be expanded further. Page B is neither in the cache, nor in the bookmarked location, therefore one can request it to keep the number of symbols minimized. That would place A in the bookmarked location, so to ensure the transition back to the SATISFIED state, we request A. This completes the substring creation as both the internal states and the contents of the cache are the same. So the resultant substring is CABA with a cost of 4 page faults.

The creation of the substring is shown below. The number specifies the step currently taken, the first letter represents the state in which algorithm currently is, and the letter after the slash shows what the next request is. The bold letter in the box represents the current junior page while the letter in a standalone box shows what the bookmarked location has in it. If the location is empty, that means the algorithm is not allowed to use information in the bookmarked location.

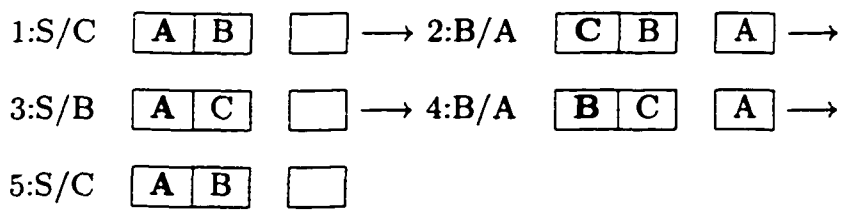


Figure 4.3: Derivation of the worst case substring for MRUM.

As we see the algorithm returns both to the initial state and cache configuration upon completion of the substring.

The optimal algorithm would have a cost of 2 page faults in this case having the knowledge of the entire input substring. This achieves the competitiveness constant of $\frac{cost_{MRUM}(CABA)}{cost_{OPT}(CABA)} = \frac{4}{2} = 2$ for MRUM. The execution of the optimal algorithm for CABA is shown below in Figure 4.4. The bold letters represent the junior page at each step.

Figure 4.4: Optimal algorithm processing worst case substring for MRUM.

The other option is to consider the algorithm changing states in the SATISFIED $\longrightarrow$ BOOKMARKED $\longrightarrow$ UNSATISFIED $\longrightarrow$ SATISFIED order. By producing the worst case sequence CDACBA and tracing algorithm steps we quickly convince ourselves that this sequence does not achieve the maximum competitiveness ratio. This is because the cost incurred by MRUM is 6 while the cost incurred by the optimal algorithm is 4. So the ratio $\frac{cost_{MRUM}(CDACBA)}{cost_{OPT}(CDACBA)} = \frac{3}{2}$ is less than that obtained in the case of the SATISFIED $\longrightarrow$ BOOKMARKED $\longrightarrow$ SATISFIED transition.

The next step is to develop a substring that would force worst case behavior upon the LRU algorithm. Following the same strategy as for MRUM we try to obtain a sequence of requests each of which would result in a page fault to the algorithm while minimizing the number of alphabet elements used. The initial cache configuration is also A in the junior cache location, B in the senior cache location, and A as the bookmarked page. By bringing in a page that is neither in the cache, nor bookmarked, say C, we force the first page fault, and the algorithm changes state to BOOKMARKED. According to LRU strategy B is evicted and put in the bookmarked location. We then supply B as the next request to change back to the SATISFIED state. To ensure the contents of the cache at the start and the end of the algorithm are the same we provide A, and the content of the cache becomes A B.

However, even though it may appear that we have reached the same configuration that we had at the beginning of the execution, the substring CBA is not sufficient.

The reason is that the internal state of the algorithm is BOOKMARKED and not SATISFIED. Applying CBA one more time quickly brings the algorithm to the same configuration that it had at the start. So CBACBA is the minimal substring that guarantees worst case cost incurred by the LRU algorithm.

Derivation of the substring for LRU is shown in the Figure 4.5. Again, the bold letters represent the junior page and the standalone box shows the bookmarked page. If the box is empty, it means that the algorithm has information about the bookmarked page unavailable. The number denotes current step, the letter before the slash represents the state of the algorithm, and the letter after the slash represents the next request.

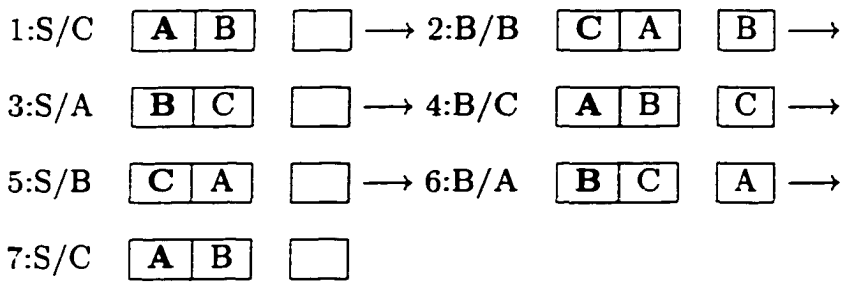


Figure 4.5: Derivation of the worst case substring for LRU.

The optimal algorithm would incur a cost of 3 page faults. Its execution is shown in the Figure 4.6 for the sequence CBACBA.

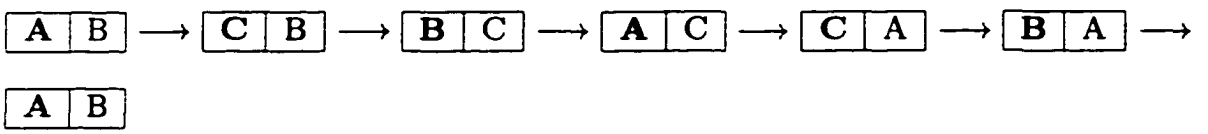


Figure 4.6: Optimal algorithm processing worst case substring for LRU.

Therefore, this would produce the maximum cost ratio of $\frac{cost_{LRU}(CBACBA)}{cost_{OPT}(CBACBA)} = 2$ as expected.

The approach using SATISFIED $\longrightarrow$ BOOKMARKED $\longrightarrow$ UNSATISFIED $\longrightarrow$ SATISFIED as a possible transition sequence yields the potential string CDACBA as the worst case scenario and the cost of 6. This is not the worst possible substring,

however, because the optimal cost in this case would be 4, so the competitiveness ratio $\frac{cost_{MRUM}(CDACBA)}{cost_{OPT}(CDACBA)} = \frac{3}{2}$ is less than in the case of CBACBA as the input sequence.

Now we are ready to answer the question which input would be the worst case scenario for the combined algorithm. If the adversary provides the worst case sequence for MRUM, that will ensure the competitiveness constant of 2 for MRUM, or that the algorithm will have the cost exactly twice larger than that of the optimal algorithm. However, if we examine the optimal service of such a sequence we observe that it is exactly the service provided by the LRU algorithm. So the cost to the combined algorithm will be $2 \cdot cost_{OPT} + cost_{OPT} = 3cost_{OPT}$.

Analogously, if the adversary provides the worst case sequence for LRU, then LRU will have the cost of $2 \cdot cost_{OPT}$. However, in this case service provided by the MRUM algorithm and service provided by the optimal algorithm identify. So the total cost to the combined algorithm will be $cost_{OPT} + 2 \cdot cost_{OPT} = 3cost_{OPT}$.

The fact that the worst sequence for one algorithm is the best case sequence for the other is not a coincidence. The reason is that the two algorithms differ exactly in one instance. When a page has to be evicted one of the algorithms evicts and bookmarks it, while the other keeps it. Since the optimal algorithm has to do one of these actions, one of the LRU and MRUM identifies with the optimal algorithm. If another fresh request comes in, the bookmarks can be safely erased, since the request will be also a fresh request for the optimal cache, so all algorithms will incur a cost so the competitiveness constant will not change.

Therefore, for any ratio $m : n$, where $m + n = 1$ of LRU and MRUM worst case input strings the resulting cost will be $(m \cdot cost_{OPT} + 2n \cdot cost_{OPT}) + (2m \cdot cost_{OPT} + n \cdot cost_{OPT}) = 3(m+n) \cdot cost_{OPT} = 3cost_{OPT}$. Since only one of the two algorithms is executed for each input string, the cost is $\frac{3}{2}cost_{OPT}$ on average as proven previously.

Therefore, the worst case, the average case and the best case input strings are all the same for the combined algorithm and all lead to $\frac{3}{2}cost_{OPT}$, which in turn produces competitiveness of $\frac{3}{2}$.

Algorithm Performance

Comparative performance of LRU, MRUM, RAN versus the optimal algorithm over longer strings is shown in the three graphs below. In the first graph in the Figure 4.7 the input string was designed to be the worst case scenario for LRU. As we see the cost of LRU is the highest in that it achieves its worst possible bound of 2 specified by its competitiveness. At the same time MRUM achieves optimal performance, and RAN algorithm stays in between the two values fluctuating around the $\frac{3}{2}$ mark.

In the second graph we design a sequence that forces MRUM to have its worst case cost. In this case LRU has optimal performance, and RAN is again in the middle. This graph is shown in the Figure 4.8.

Finally, for a sequence that is equally distributed in terms of worst case LRU and MRUM request substrings, we have a distribution of both LRU and MRUM centered around the $\frac{3}{2}$ mark as the costs of both algorithms converge to $\frac{3}{2}cost_{OPT}$. This graph is shown in the Figure 4.9. Note that not only is the cost of RAN also centered around the $\frac{3}{2}$ mark, but RAN is sometimes able to achieve better performance than both algorithms !

The length of the input string is 100 requests in all graphs.

Since RAN is $\frac{3}{2}$-competitive, this algorithm has better performance than the existing 2-competitive deterministic algorithms, and it is the best possible as it achieves the $\frac{3}{2}$ lower bound for randomized paging algorithms.

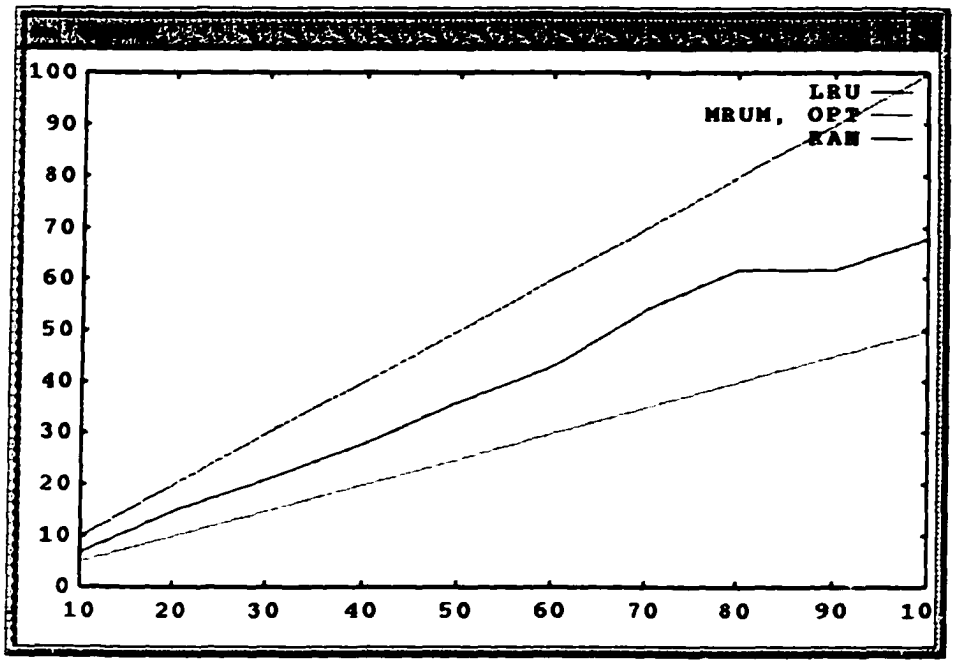


Figure 4.7: The performance of the 4 algorithms for LRU worst case input string.

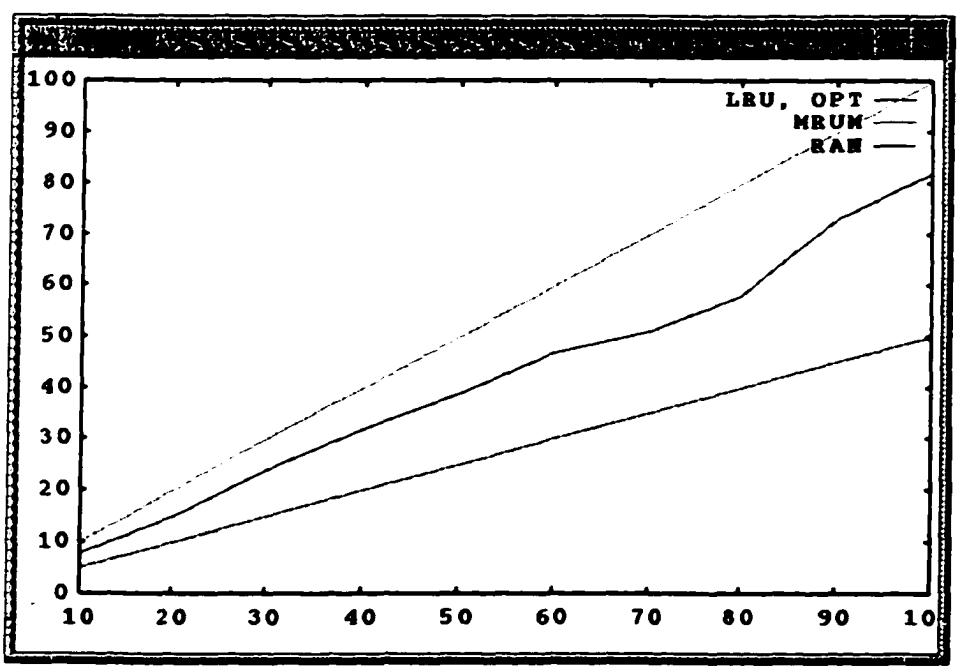


Figure 4.8: The performance of the 4 algorithms for MRUM worst case input string.

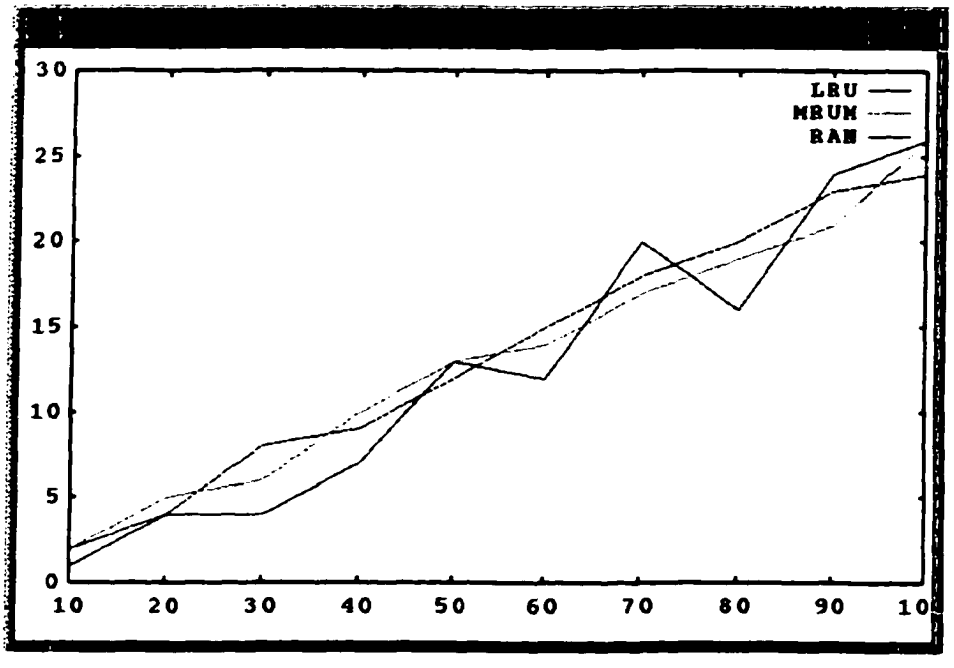


Figure 4.9: The performance of the 4 algorithms for the combined MRUM and LRU worst case input string.

CHAPTER 5

APPLICATIONS

Having deduced a new algorithm in the preceding chapters it would be interesting to estimate the practical gain this algorithm would yield. We first analyze the benefits created by the trackless approach. Then our algorithm is viewed in the framework of trackless algorithms, and finally existing algorithms for the paging problem are reviewed and our algorithm is compared to them.

As we have discussed the trackless approach introduces a new paradigm of information handling. When information is received the algorithm can choose to either store this information or to discard it. We consider both of these possibilities in turn.

There are clearly obvious advantages to storing information. First of all, having the knowledge of information already processed one can resort to it when making decisions as to how to serve the requests currently under consideration. For example, if a particular solution has been obtained to a particular request, instead of processing the request anew, the algorithm can simply provide the solution to an earlier solved problem as the output hence improving efficiency.

Second of all, having the knowledge of previously processed information, it is possible to conduct an analysis of the data for possible future response strategies. For example, if the requests demonstrate a certain pattern of access, then the algorithm can anticipate future requests to some degree and based on that pattern decide which page to keep and which to evict. Even though processing information can impose an overhead on the overall performance of such algorithms, being able to have access to

the past requests can prove quite beneficial for large amounts of requests and if the derived behavior effectively predicts future patterns of page accesses.

On the other hand, there are disadvantages to storing information. The first and the most obvious disadvantage is the need for a physical space to hold the information. If much extra space is required for a paging algorithm to simply keep track of past requests, and if the incoming requests do not demonstrate a clear pattern of behavior or any other feature that lends itself to future optimization, then the extra space is simply being wasted, and one can be better off simply using that extra space to increase the cache size without the need to hold any information about past events.

The second disadvantage is that the information that is stored may also need to be maintained. For example, if the algorithm relies on the stored information to design its own strategy, then such a strategy would be severely affected if the information about past requests was invalid.

Consider for example, what happens if an algorithm is to store Web pages requested from the Internet. The volume of such pages is potentially very high, and it may very well be that the extra space allocated will be exceeded much before the pages display a pattern of page access, or in any other way enable the algorithm to develop an efficient strategy. More importantly, original page contents may have changed while the page was stored at the algorithm location. Then there are additional requirements on maintaining time and date stamps for each page to ensure the information remains valid.

Even maintaining time and date stamps may not be sufficient to ensure information integrity. Time and date stamps are set by the system of the original user, so it may or may not accurately reflect the actual modification time. One can very well imagine a situation when a page has been modified while the time and date stamp have not.

There are a number of ways of dealing with this problem to prevent stale pages from being accessed both in the paging model, such as write-back caches, as well as in the Web page access model, such as signature transmission. While these techniques may be highly effective it is clear that they introduce some overhead to system performance. It would be interesting to see how a system without any knowledge of past requests will perform.

This is the idea behind trackless model of computation as applied to paging. The model saves significantly in terms of space and time required for the functioning of the algorithm. This model has been well described above, so we simply note the fact that the performance of a trackless algorithm cannot in general be as competitive as the performance of an algorithm that is allowed to keep track of past requests.

The next question is what is the minimum amount of information required to have a competitive algorithm while not introducing a significant overhead. A result by Achlioptas et al. [1] presents an algorithm called EQUITABLE which has complexity H_k, so we see that this algorithm is optimally competitive since it achieves the lower bound for randomized paging algorithms. However, it requires a rather high number of memory locations to store information about past requests. This number is $\lceil 5k^2\mathcal{H}_k \rceil$, so we see that even for a 2-page cache the number of such locations is 30.

Our algorithm presents a way of achieving optimal competitiveness while bringing down the number of bookmarks needed. As we have seen the 1-Bookmark Algorithm uses only one extra page location to keep track of past requests for a 2-page cache. Furthermore, it achieves $\frac{3}{2}$-competitiveness which is the lower bound, and hence the 1-Bookmark Algorithm is optimal.

The applications of this new algorithm are diverse. It can be used to ensure better cache performance on computer systems where space and cost constraints play a significant role. It can also be used to enhance performance of existing systems by allocating the existing space for caching purposes, like for example in the virtual

memory model. Additionally, it can be used in environments where a deterministic strategy would be potentially damaging as in the case of possible long sequences of requests which are worst case input for existing deterministic algorithms.

Appendix A

SOURCE CODE AND OUTPUT EXAMPLES

The program for running the algorithms with different types of input is given below. We also provide examples of output for the cases with random request distribution, worst MRUM sequence, worst LRU sequence, and where one half of the sequence is worst case for MRUM and the other half is worst case for LRU. The length of the strings of these examples is 8 requests. The reason is that strings of lesser length may not be sufficient to demonstrate the differences in behavior of the algorithms, while the strings of greater length cause the output to grow beyond reasonable size. The examples for the described types of strings are given below.

```
////////////////////////////////////////////////////////////////////////
//
//                  Comparative Algorithm Demonstration.
//
// Who:   Edward Benjamin Mikhalkov
// When:  January 5, 2000
// What:  Finished LRU, MRUM and OPT handling.
// Input: The length of the input string.
// What:  The program generates output demonstrating behavior of the
// LRU, MRUM, OPT and RAN algorithms.
// Thesis supervisor: Dr. Wolfgang Bein.
////////////////////////////////////////////////////////////////////////
```

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#define SATISFIED     1   /* State labels. */
#define BOOKMARKED    2
#define UNSATISFIED   3

#define TRUE          1
#define FALSE         0
#define END_OF_INPUT '\0'
#define CARDINALITY   26  /* The number of letters that input       */
                          /* alphabet has. */
#define MAX_LENGTH 10000 /* The maximum length of input string.    */

/* Structure that maintains information that is known to          */
/* each algorithm.                                                */
typedef struct algoState_
{
  int  State;
  char Junior;
  char Senior;
  char Bookmark;
  int  Cost;
} algoState;

char INPUT_STRING[MAX_LENGTH]; /* The input string to be fed       */
```

```
                                /* to algorithms.                    */
int  LENGTH;                    /* The length of the input string.   */
int cur_position;

/*************** Function declarations. ***************/
void generateInput(int);  /* A function to generate */
                          /* an input string.       */

void worstLRU(int);       /* These are the functions to create */
void worstMRUM(int);      /* different testing scenarios.      */
void averageCase(int);
void averageLRU_MRUM(int);

int generateRandom(int);  /* A function to generate a random */
                          /* number in the range from 0 to   */
                          /* the given parameter.            */

void processInput();      /* A function to produce the        */
                          /* demonstration of the algorithms' */
                          /* behavior. */

void initializeAlgo(algoState*); /* Function to initialize     */
                                 /* all algorithms to the same */
                                 /* starting conditions.       */

void printCache(algoState*); /* This function displays the cache */
                             /* at any time during the execution */
```

```
                                /* of the algorithm.                */

void applyLRU(algoState*);  /* Functions to calculate the     */
                            /* contents of the cache of each  */
void applyMRUM(algoState*); /* algorithm.                     */
void applyOPT(algoState*);
void applyRAN(algoState*);

char refereeResponse(char, char, char); /* This function provides */
                                        /* the response which is  */
                                        /* the only thing that    */
                                        /* the algorithm sees.    */

/*************** Function definitions. ***************/
int generateRandom(int n)
{
  float f;

  f = RAND_MAX / (4.0 * random()); /* Ensure will work for 32-bit */
  f = 1.0 / (4.0 * f);             /* architectures as well.      */

  return (int)(n*f);
}

void averageCase(int length)
{
  int i;
```

```
    int curr_symbol;

    /* Fill in the string with the possible input characters A-Z. */
    for (i = 0; i < length; i++)
      {
        curr_symbol = generateRandom(CARDINALITY);
        /* Move to the required range, cast to a character and store.*/
        INPUT_STRING[i] = (char) (curr_symbol + 65);
      }
}

void generateInput(int length)
{
  averageCase(length);   /* Generate input string depending on */
  //worstLRU(length);      /* which case we are interested in.   */
  //worstMRUM(length);
  //averageLRU_MRUM(length);
}


void worstLRU(int length)
{
  int i;
  int curr_symbol;

  /* Fill in the string with the possible input characters A-Z. */
  for (i = 0; i < length; i++)
```

```
    {
      switch(i%3) {
      case 0:
        INPUT_STRING[i] = 'C';
        break;
      case 1:
        INPUT_STRING[i] = 'B';
        break;
      case 2:
        INPUT_STRING[i] = 'A';
        break;
      }
    }
}

void worstMRUM(int length)
{
  int i;
  int curr_symbol;

  /* Fill in the string with the possible input characters A-Z. */
  for (i = 0; i < length; i++)
    {
      switch(i%4) {
      case 0:
      INPUT_STRING[i] = 'C';
        break;
```

```
        case 1:
        INPUT_STRING[i] = 'A';
          break;
        case 2:
        INPUT_STRING[i] = 'B';
          break;
        case 3:
        INPUT_STRING[i] = 'A';
          break;
        }
      }
}

void averageLRU_MRUM(int length)
{
  worstMRUM(length);
  worstLRU(length/2-1);
}

void processInput()
{
  algoState lru;
  algoState mrum;
  algoState opt;
  algoState ran;

  /* Initialize all to the start conditions. */
```

```
initializeAlgo(&lru);
initializeAlgo(&mrum);
initializeAlgo(&opt);
initializeAlgo(&ran);


/* Print out initial states.                */
printf("Alg    Junior Senior   Bookmark\n");
printf("=======================================\n");
printf("LRU: ");
printCache(&lru);
printf("MRUM:");
printCache(&mrum);
printf("OPT: ");
printCache(&opt);
printf("RAN: ");
printCache(&ran);
printf("---------------------------------------\n");

/* Begin processing.                        */
for (cur_position = 0; cur_position < LENGTH; cur_position++)
  {
    printf("Next request: %c\n", INPUT_STRING[cur_position]);

    printf("LRU: ");
    applyLRU(&lru);
    printCache(&lru);
```

```
        printf("MRUM:");
        applyMRUM(&mrum);
        printCache(&mrum);

        printf("OPT: ");
        applyOPT(&opt);
        printCache(&opt);

        printf("RAN: ");
        applyRAN(&ran);
        printCache(&ran);

        printf("----------------------------------------\n");

    }

  printf("Cost   LRU: %d, MRUM: %d, OPT: %d RAN: %d\n\n",
lru.Cost, mrum.Cost, opt.Cost, ran.Cost);

}

void printCache(algoState * a)
{
  printf("  +-----------+   +-----+   \n");
  printf("        | %c  | %c  |   | %c  | \n", a->Junior,
a->Senior, a->Bookmark);
```

```
    printf("       +-----------+   +-----+   \n\n");
  }

  void initializeAlgo(algoState * a)
  {
    a->State    = SATISFIED;
    a->Junior   = 'A';
    a->Senior   = 'B';
    a->Bookmark = 'A';
    a->Cost     =  0;
  }

  void applyLRU(algoState * lru)
  {
    char resp;

    resp = refereeResponse(lru->Junior, lru->Senior, lru->Bookmark);

    if ( cur_position >=LENGTH ) /* End of the input string reached. */
      return;

    /* Update according to the referee's response. */
    switch(lru->State) {
    case SATISFIED:
      if ( resp == 'h') /* We have a hit to the Senior page. */
        {
          lru->Senior = lru->Junior;
```

```
        lru->Junior = INPUT_STRING[cur_position];
      }
    else {
      /* The page is not in the cache. */
          lru->State  = BOOKMARKED;
          lru->Bookmark = lru->Senior;

          /* Bookmark the evicted page. b+ */
          lru->Senior = lru->Junior;

          /* Bring the new page in. */
          lru->Junior = INPUT_STRING[cur_position];
          (lru->Cost)++; /* Update cost for bringing the page in. */
   }
   break;
 case BOOKMARKED:
      if ( resp == 'h') /* We have a hit to the Senior page. */
     {
       lru->State  = SATISFIED;
       lru->Senior = lru->Junior;
       lru->Junior = INPUT_STRING[cur_position];
     }
   else {
     /* The page is not in the cache. */
     if ( resp == 'p' ) /* Hit to the Bookmarked page. */
       {
         lru->State  = SATISFIED;
```

```
            lru->Senior = lru->Junior;
            /* Bring the new page in. */
            lru->Junior = INPUT_STRING[cur_position];
            /* Update cost for bringing the page in. */
            (lru->Cost)++;
          }
        else { /* Miss */
            lru->State  = UNSATISFIED;
            lru->Senior = lru->Junior;
            /* Bring the new page in. */
            lru->Junior = INPUT_STRING[cur_position];
            (lru->Cost)++; /* Update cost for bringing the page in. */
        }
      }
      break;
    case UNSATISFIED:
          if ( resp == 'h') /* We have a hit to the Senior page. */
        {
          lru->Senior = lru->Junior;
          lru->Junior = INPUT_STRING[cur_position];
        }
      else {
        /* The page is not in the cache. */
            lru->State  = SATISFIED;
            lru->Senior = lru->Junior;
            /* Bring the new page in. */
            lru->Junior = INPUT_STRING[cur_position];
```

```
            (lru->Cost)++; /* Update cost for bringing the page in. */
        }
        break;
    }
}

void applyMRUM(algoState * mrum)
{
    char resp;

    resp = refereeResponse(mrum->Junior, mrum->Senior, mrum->Bookmark);

    if ( cur_position >=LENGTH ) /* End of the input string reached. */
        return;

    /* Update according to the referee's response. */
    switch(mrum->State) {
    case SATISFIED:
        if ( resp == 'h') /* We have a hit to the Senior page. */
            {
                mrum->Senior = mrum->Junior;
                mrum->Junior = INPUT_STRING[cur_position];
            }
        else {
            /* The page is not in the cache. */
                mrum->State  = BOOKMARKED;
                /* Bookmark the evicted page. a+ */
```

```
            mrum->Bookmark = mrum->Junior;
            /* Bring the new page in.*/
            mrum->Junior = INPUT_STRING[cur_position];
            (mrum->Cost)++; /* Update cost for bringing the page in. */
        }
        break;
    case BOOKMARKED:
        if ( resp == 'h') /* We have a hit to the Senior page. */
        {
          mrum->State  = SATISFIED;
          mrum->Senior = mrum->Junior;
          mrum->Junior = INPUT_STRING[cur_position];
        }
      else {
        /* The page is not in the cache. */
        if ( resp == 'p' ) /* Hit to the Bookmarked page. */
          {
            mrum->State  = SATISFIED;
            mrum->Senior = mrum->Junior;
            /* Bring the new page in.*/
            mrum->Junior = INPUT_STRING[cur_position];
            (mrum->Cost)++; /* Update cost for bringing the page in. */
          }
        else { /* Miss */
            mrum->State  = UNSATISFIED;
            mrum->Senior = mrum->Junior;
            /* Bring the new page in.*/
```

```
            mrum->Junior = INPUT_STRING[cur_position];
            (mrum->Cost)++; /* Update cost for bringing the page in. */
        }
      }
      break;
    case UNSATISFIED:
        if ( resp == 'h') /* We have a hit to the Senior page. */
      {
        mrum->Senior = mrum->Junior;
        mrum->Junior = INPUT_STRING[cur_position];
      }
     else {
       /* The page is not in the cache. */
           mrum->State  = SATISFIED;
           mrum->Senior = mrum->Junior;
           /* Bring the new page in.*/
           mrum->Junior = INPUT_STRING[cur_position];
           (mrum->Cost)++; /* Update cost for bringing the page in. */
     }
     break;
  }
}

void applyOPT(algoState * opt)
{
  int i;
  bool evictSenior, evictJunior;
```

```
if ( cur_position >=LENGTH ) /* End of the input string reached. */
  return;

i = cur_position;
evictSenior = evictJunior = FALSE;
/* Check which one occurs the longest in the future. */
while (i < LENGTH)
  {
    if ( opt->Junior == INPUT_STRING[i] )
      {
        evictSenior = TRUE;
        i = LENGTH;
      }

 if ( opt->Senior == INPUT_STRING[i] )
      {
        evictJunior = TRUE;
        i = LENGTH;
      }
 i++;
  }

if ( (evictJunior == FALSE) && (evictSenior == FALSE) )
  {
    /* Neither page occurs in the future. */
    /* WLOG we evict the Senior page.     */
```

```
      evictSenior = TRUE;
    }

  /* Update the optimal cache. */
  if (opt->Junior == INPUT_STRING[cur_position])
    {
      /* The optimal cache has a 'Hit' to the Junior page.  */
      /* No need to do anything.                            */
    }
  else {
    if (opt->Senior == INPUT_STRING[cur_position])
      {
        /* The optimal cache has a 'Hit' to the Senior page. */
        /* We just update the labels.                        */
        opt->Senior = opt->Junior;
        opt->Junior = INPUT_STRING[cur_position];
      }
    else {
      /* We have a 'Miss'. Need to update cache. */
      if (evictSenior == TRUE)
        {
          opt->Senior = opt->Junior;
          opt->Junior = INPUT_STRING[cur_position];
          (opt->Cost)++;
        }
      else {
        opt->Junior = INPUT_STRING[cur_position];
```

```
        (opt->Cost)++;
      }
    }
  }
}

void applyRAN(algoState* ran)
{
  if (generateRandom(2))
    {
      applyLRU(ran);  /* Apply LRU half the time, and */
    }
  else
    {
      applyMRUM(ran); /* MRUM the other half.          */
    }
}


char refereeResponse(char junior, char senior, char bookmark)
{
  /* We have a 'Hit' to a page in the cache. */

  if (cur_position >= LENGTH)
    return END_OF_INPUT;          /* End of the string reached. */

  while ( junior == INPUT_STRING[cur_position]) /* Skip hits to */
```

```
  /* the Junior pages as they do not contribute to anything. */
  {
    if (cur_position >= LENGTH)
      {
        /* Last element(s) were 'Hits' to Junior. */
        return END_OF_INPUT;    /* End of the string reached. */;
      }
    else {
      cur_position++;
    }
  }

/* Analyze the relation to the pages currently known. */
if (senior == INPUT_STRING[cur_position]) /* Hit to Senior. */
  {
    return 'h';
  }
else {

  /* The request is to the Bookmarked page. */
  if ( bookmark == INPUT_STRING[cur_position])
    {
      return 'p';
    }
  else {

    /* None of the pages matched, we have a 'Miss'. */
```

```
        return 'm';
      }
    }
  }
```

```
int main() {
  int i;

  /* Get the input from the user. */
  printf("Please enter the length of the input string: ");
  scanf("%d", &LENGTH);

  /* Produce the input string. */
  generateInput(LENGTH);

  /* Display the input string. */
  printf("\nINPUT_STRING = ");
  for (i = 0; i < LENGTH; i++)
    printf("%c", INPUT_STRING[i]);
  printf("\n\n");

  /* Compare the behavior of all algorithms. */
  processInput();
  return 1;
}
```

RANDOM DISTRIBUTION EXAMPLE.

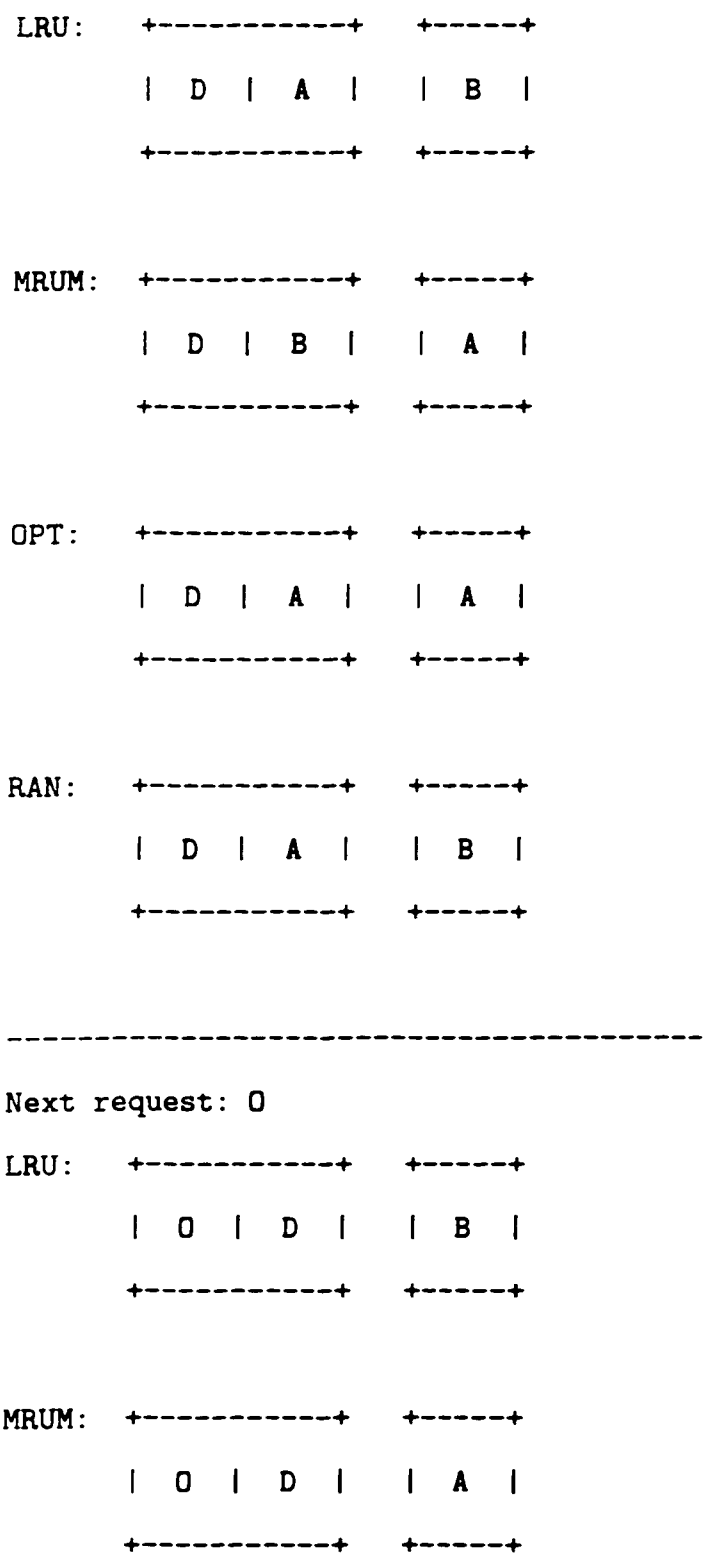
==========

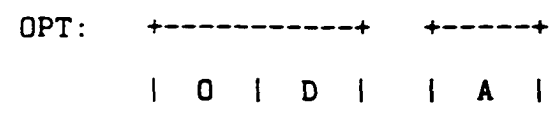
Please enter the length of the input string: 8

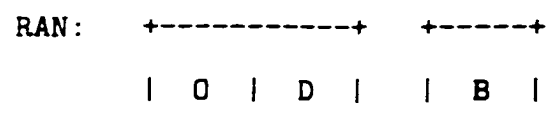
INPUT_STRING = DOKPEZRH

Alg	Junior	Senior	Bookmark
LRU:	A B		A
MRUM:	A B		A
OPT:	A B		A
RAN:	A B		A

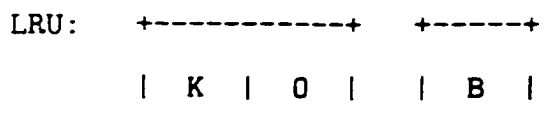
Next request: D

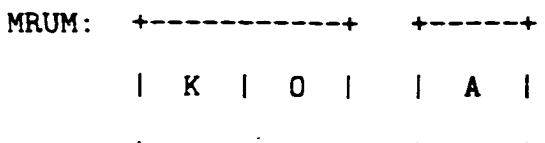


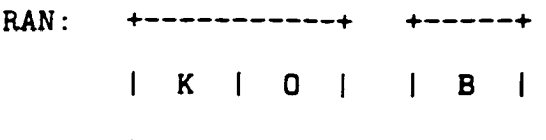




Next request: K

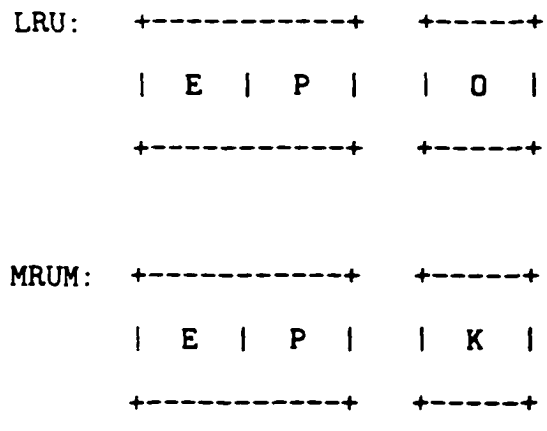


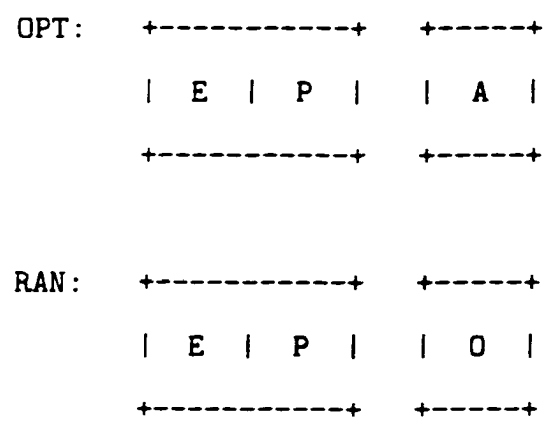




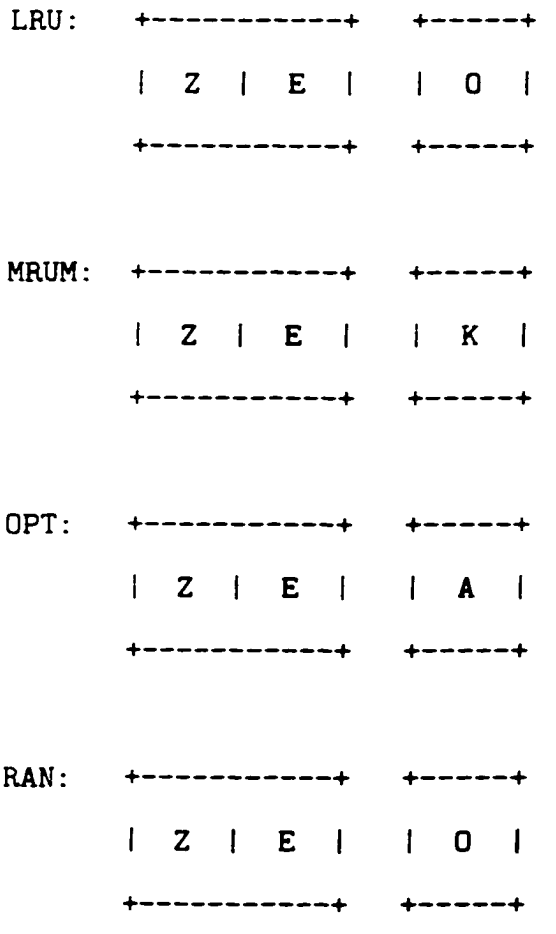
Next request: P

Next request: E





Next request: Z



Next request: R

Next request: H

```
OPT:   +-----------+   +-----+
       |  H  |  R  |   |  A  |
       +-----------+   +-----+

RAN:   +-----------+   +-----+
       |  H  |  R  |   |  Z  |
       +-----------+   +-----+
```

Cost LRU: 8, MRUM: 8, OPT: 8 RAN: 8

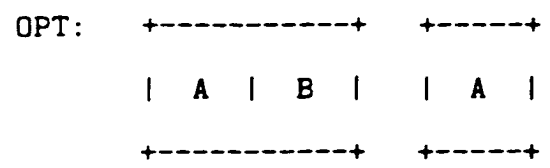
WORST MRUM SEQUENCE.

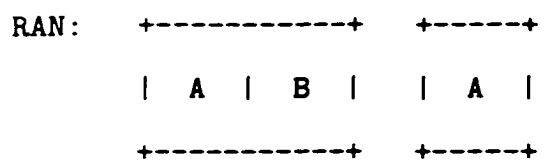
==========

Please enter the length of the input string: 8

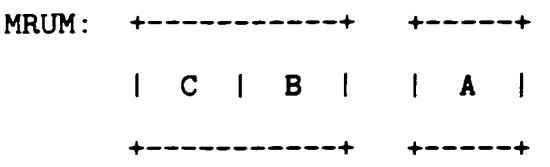
INPUT_STRING = CABACABA

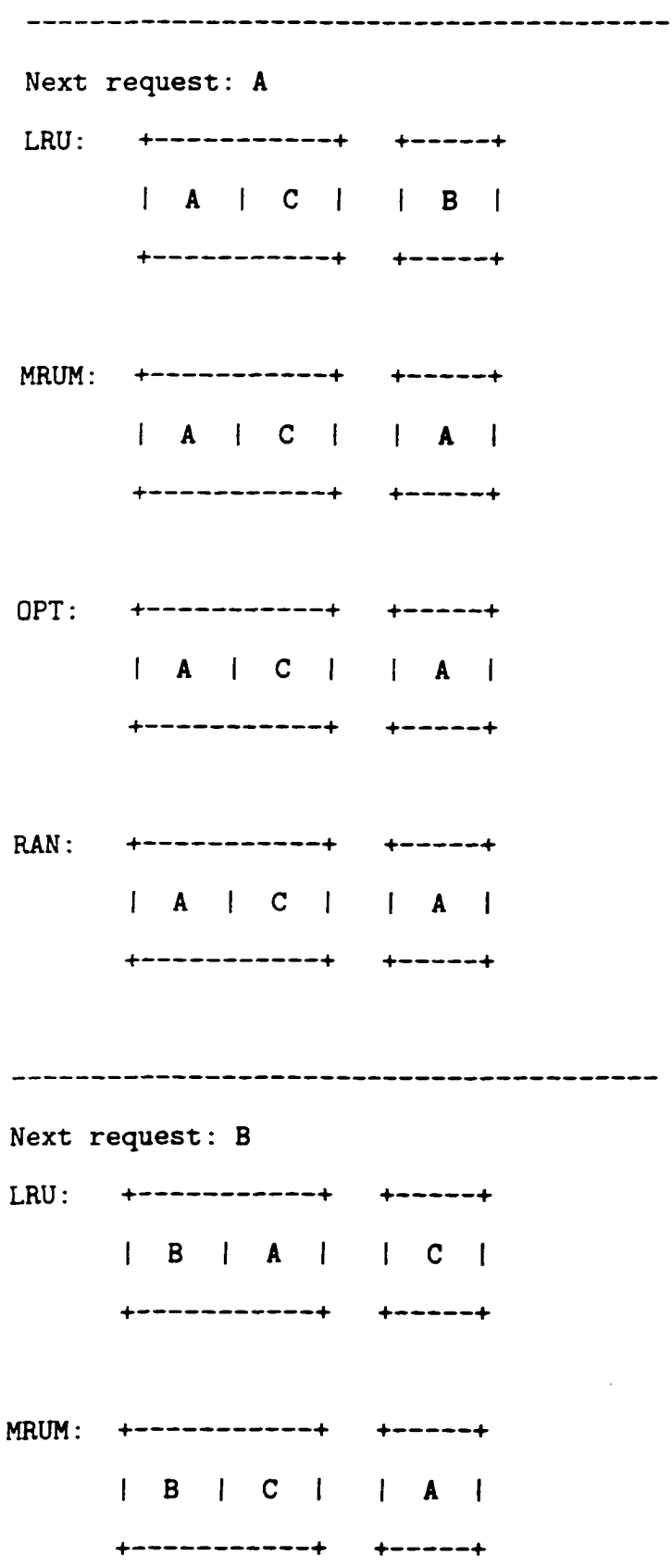
```
Alg    Junior Senior   Bookmark
=======================================
LRU:   +-----------+   +-----+
       |  A  |  B  |   |  A  |
       +-----------+   +-----+

MRUM:  +-----------+   +-----+
       |  A  |  B  |   |  A  |
       +-----------+   +-----+
```



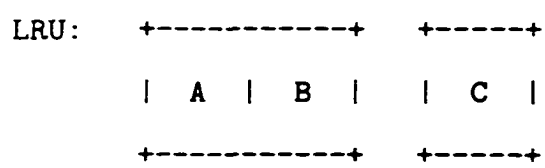


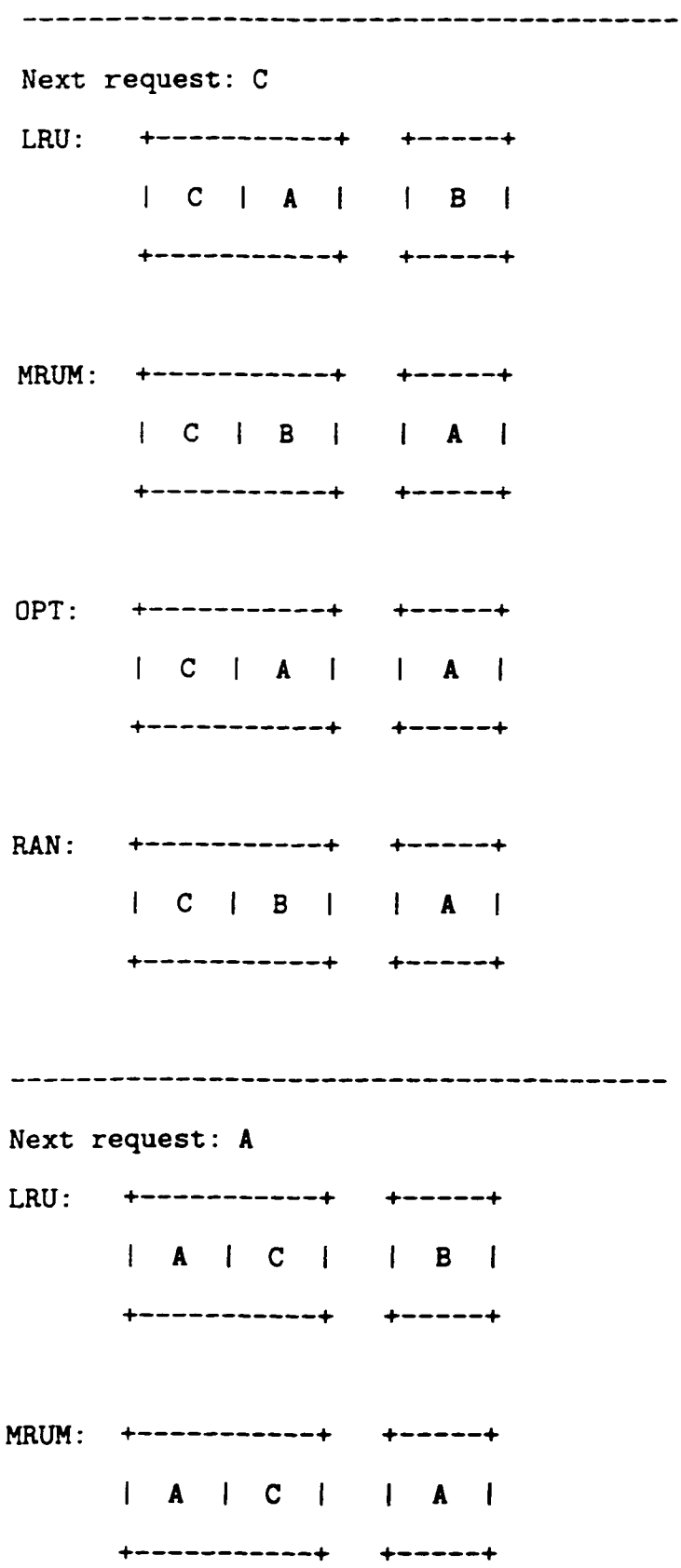
Next request: C



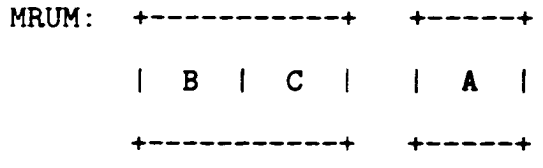


Next request: A

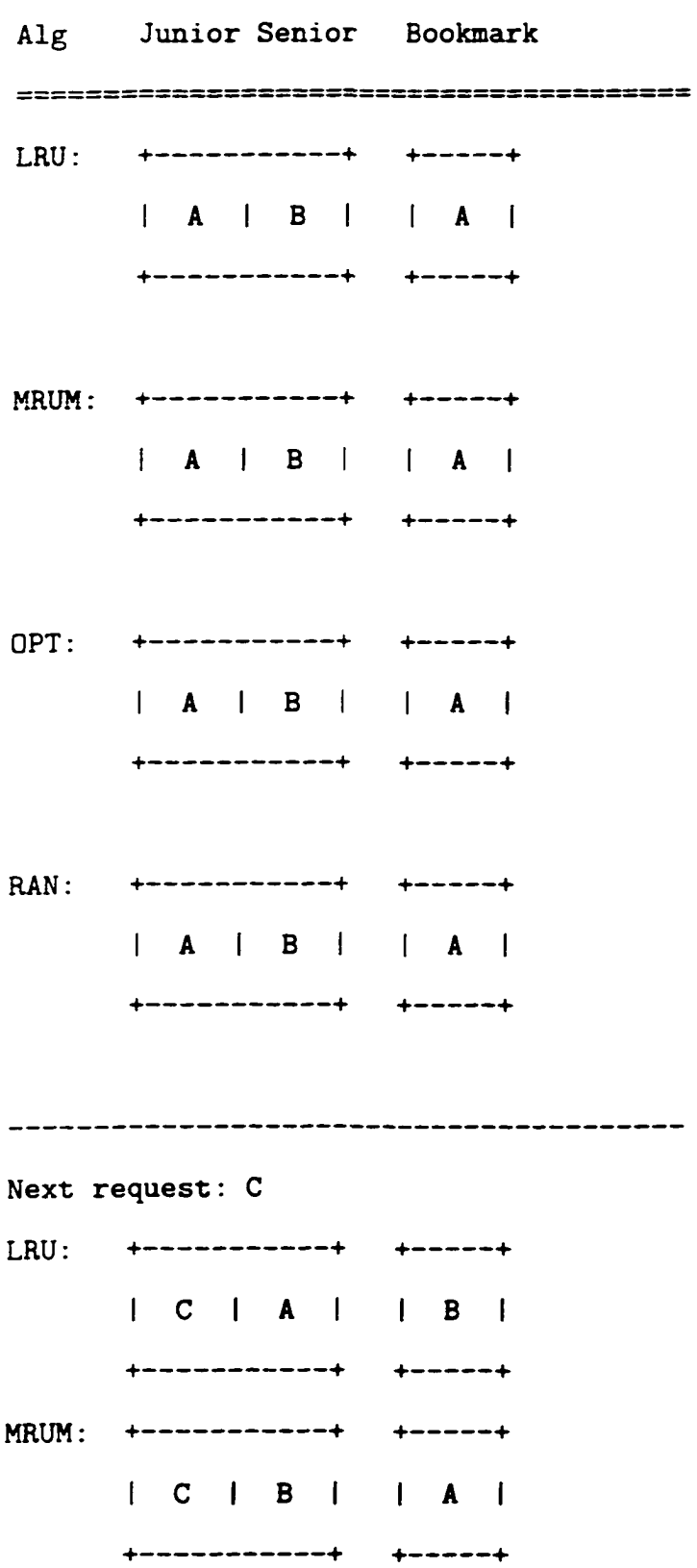


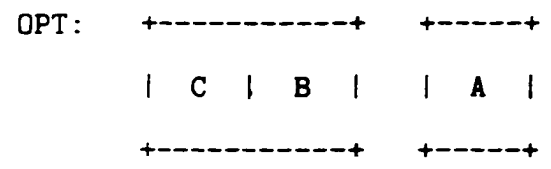


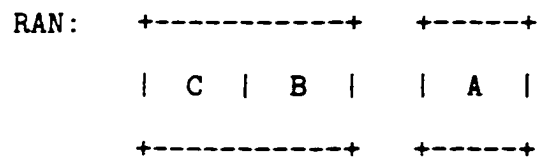
Next request: B



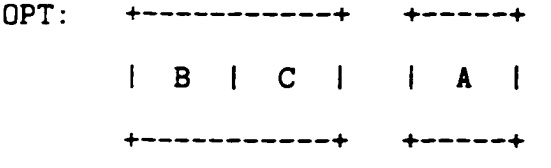
```
----------------------------------------

Next request: A

LRU:   +-----------+   +-----+
       | A | B |   | C |
       +-----------+   +-----+

MRUM:  +-----------+   +-----+
       | A | B |   | A |
       +-----------+   +-----+

OPT:   +-----------+   +-----+
       | A | B |   | A |
       +-----------+   +-----+

RAN:   +-----------+   +-----+
       | A | B |   | C |
       +-----------+   +-----+

----------------------------------------

Cost   LRU: 4, MRUM: 8, OPT: 4 RAN: 7


          WORST LRU SEQUENCE.
          ===================

Please enter the length of the input string: 8

INPUT_STRING = CBACBACB
```

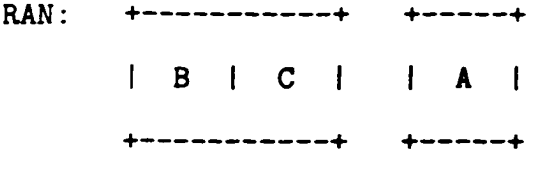


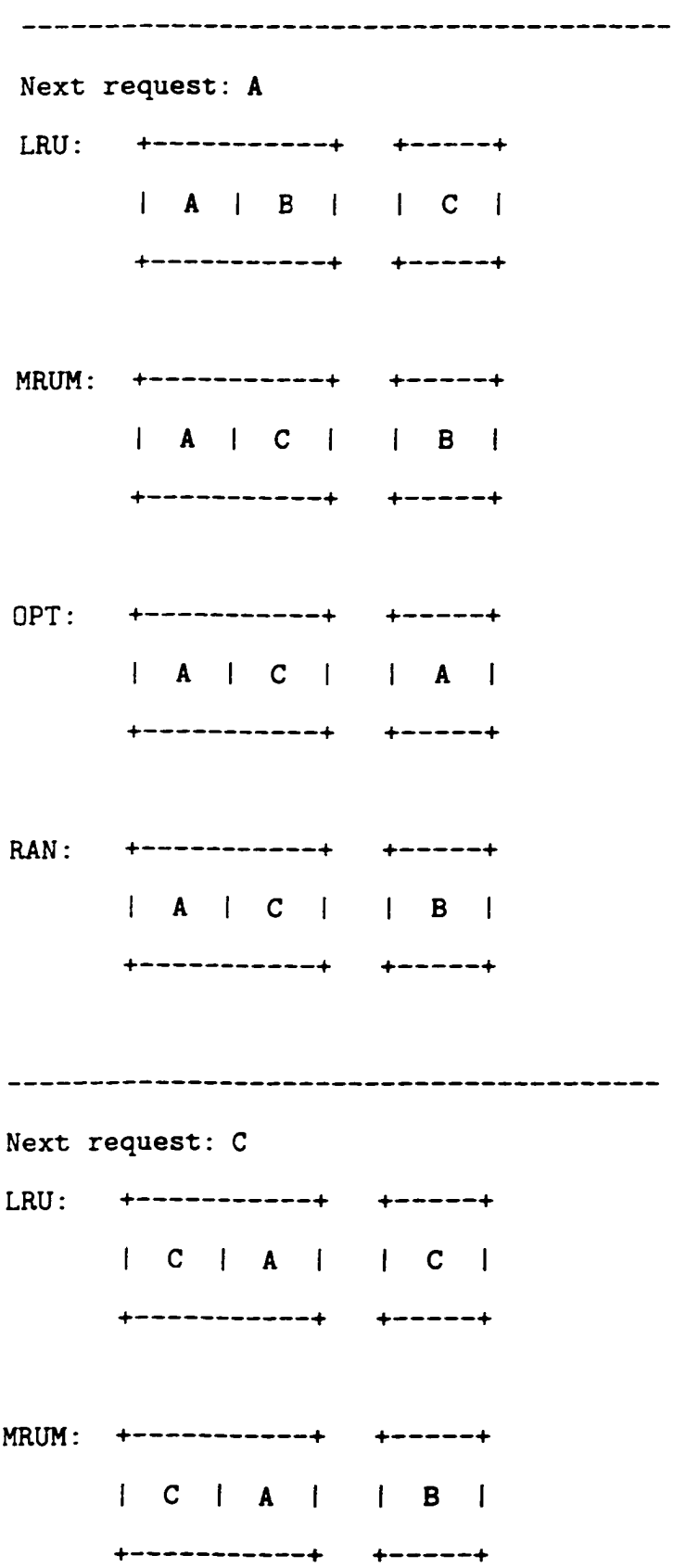


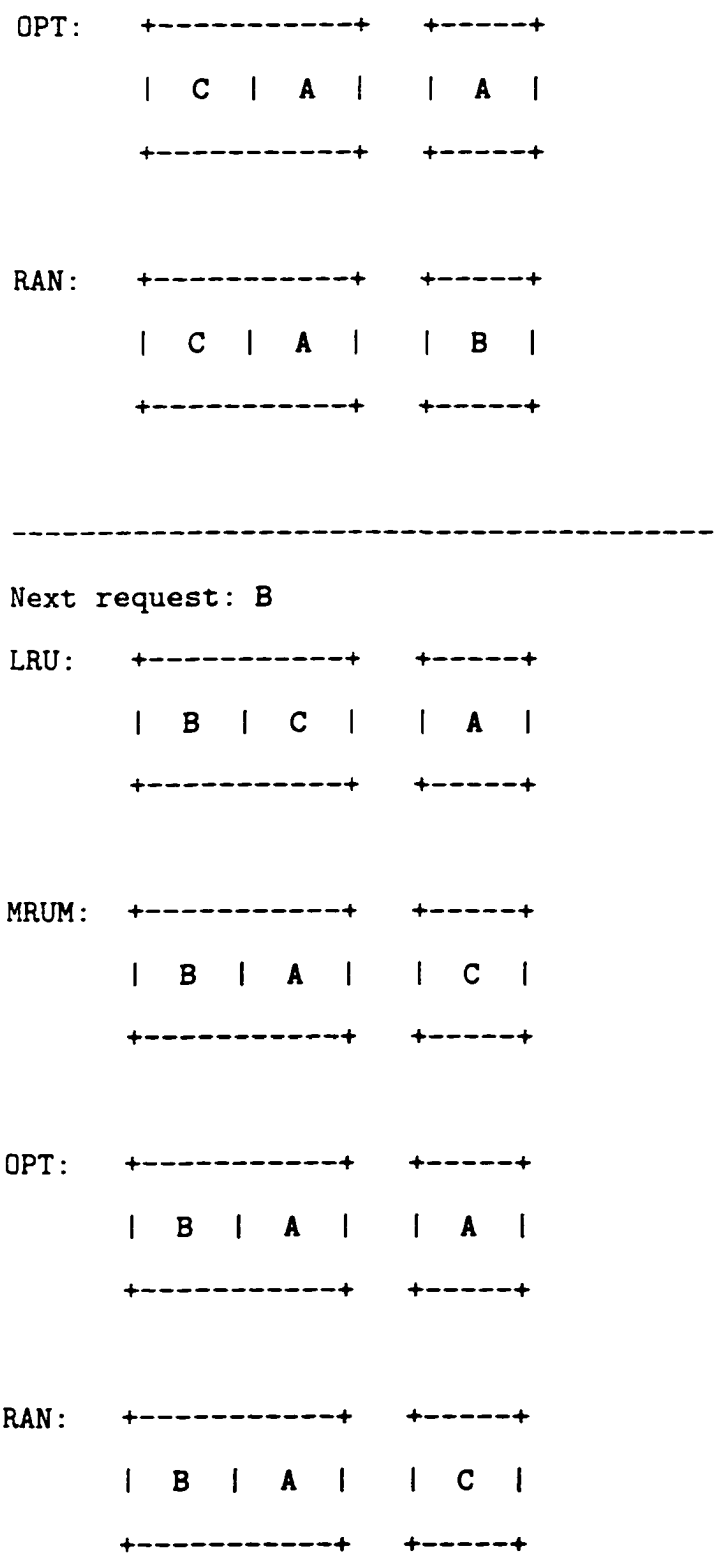


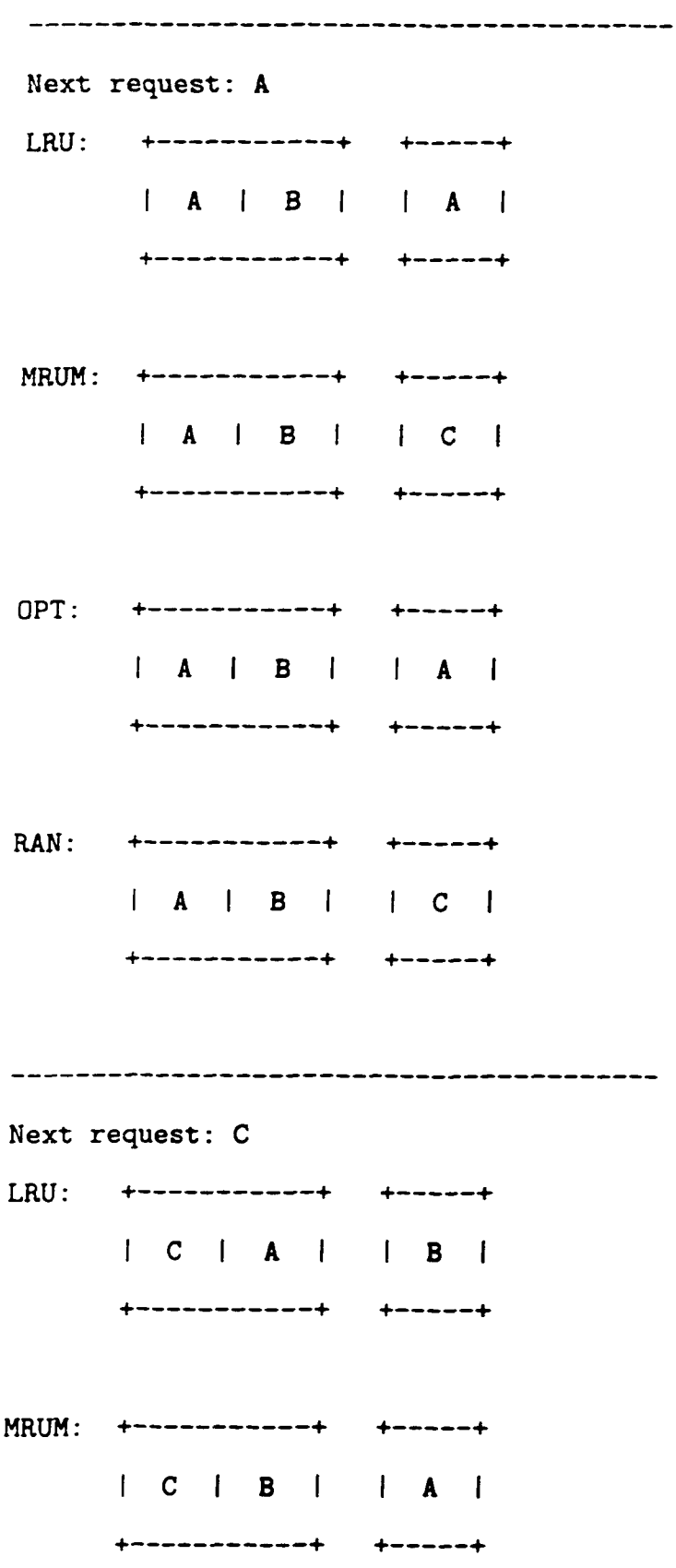
Next request: B

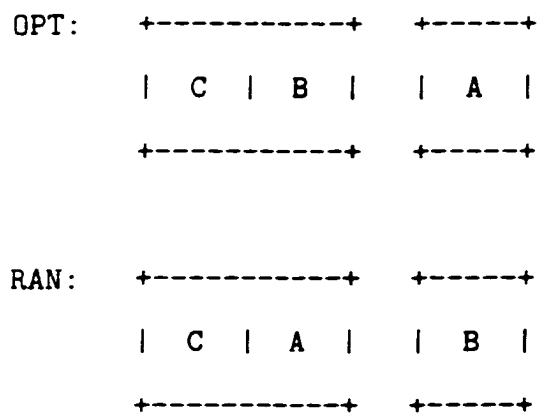




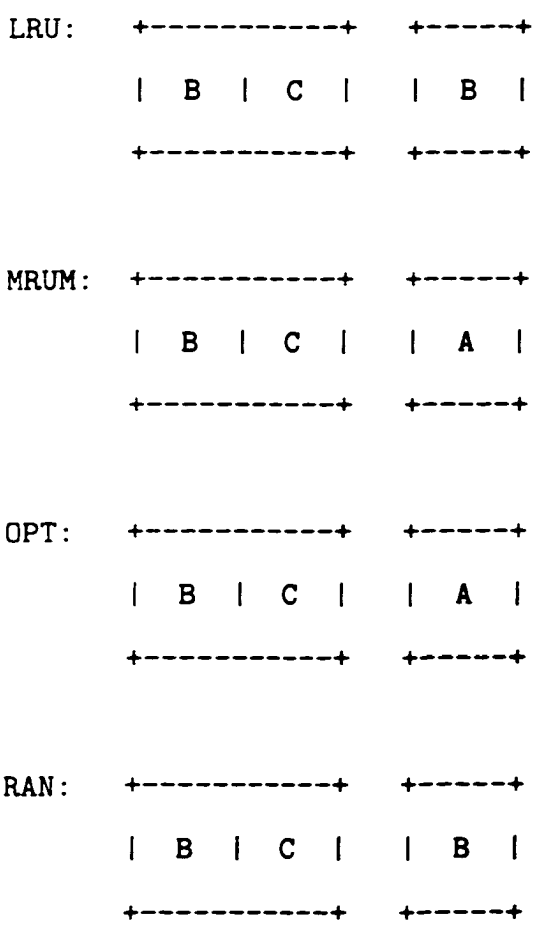




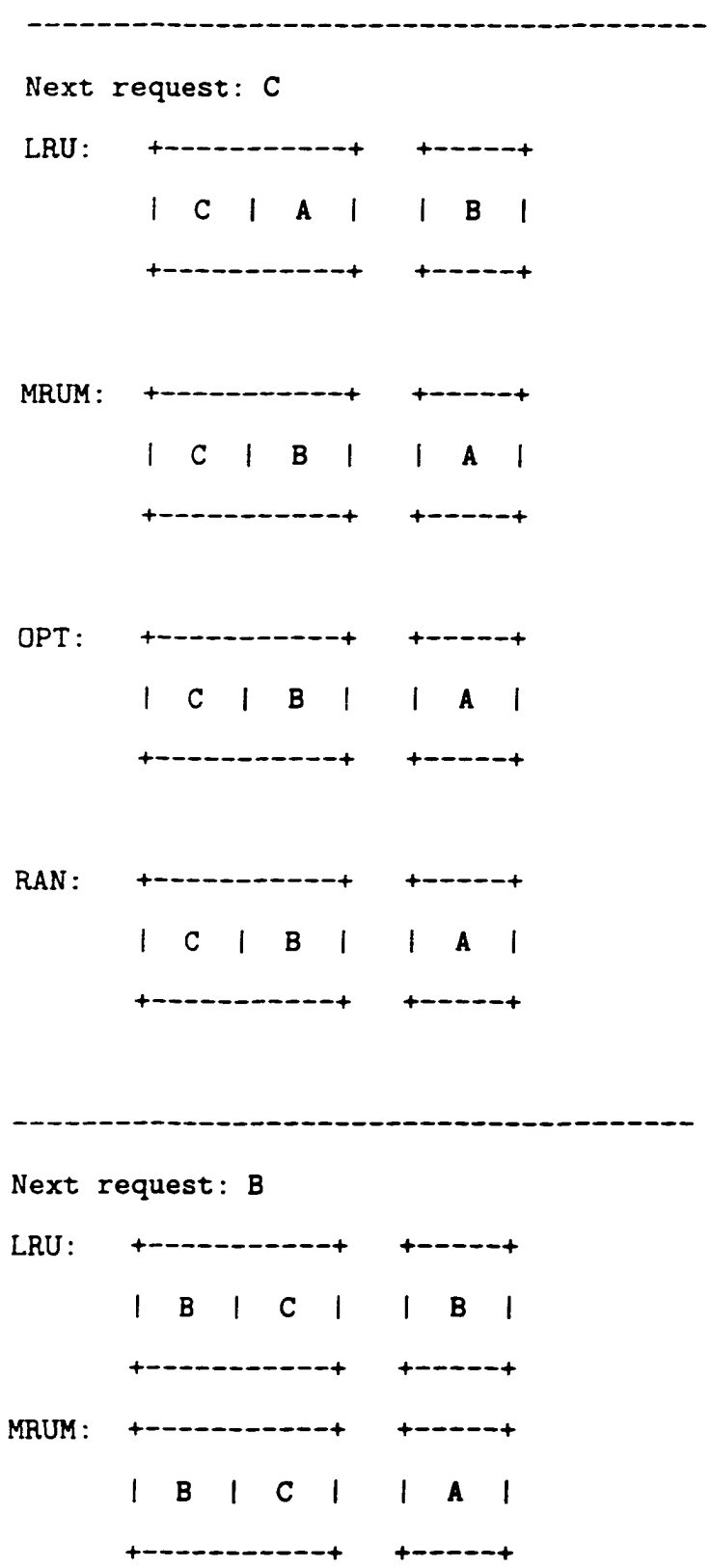


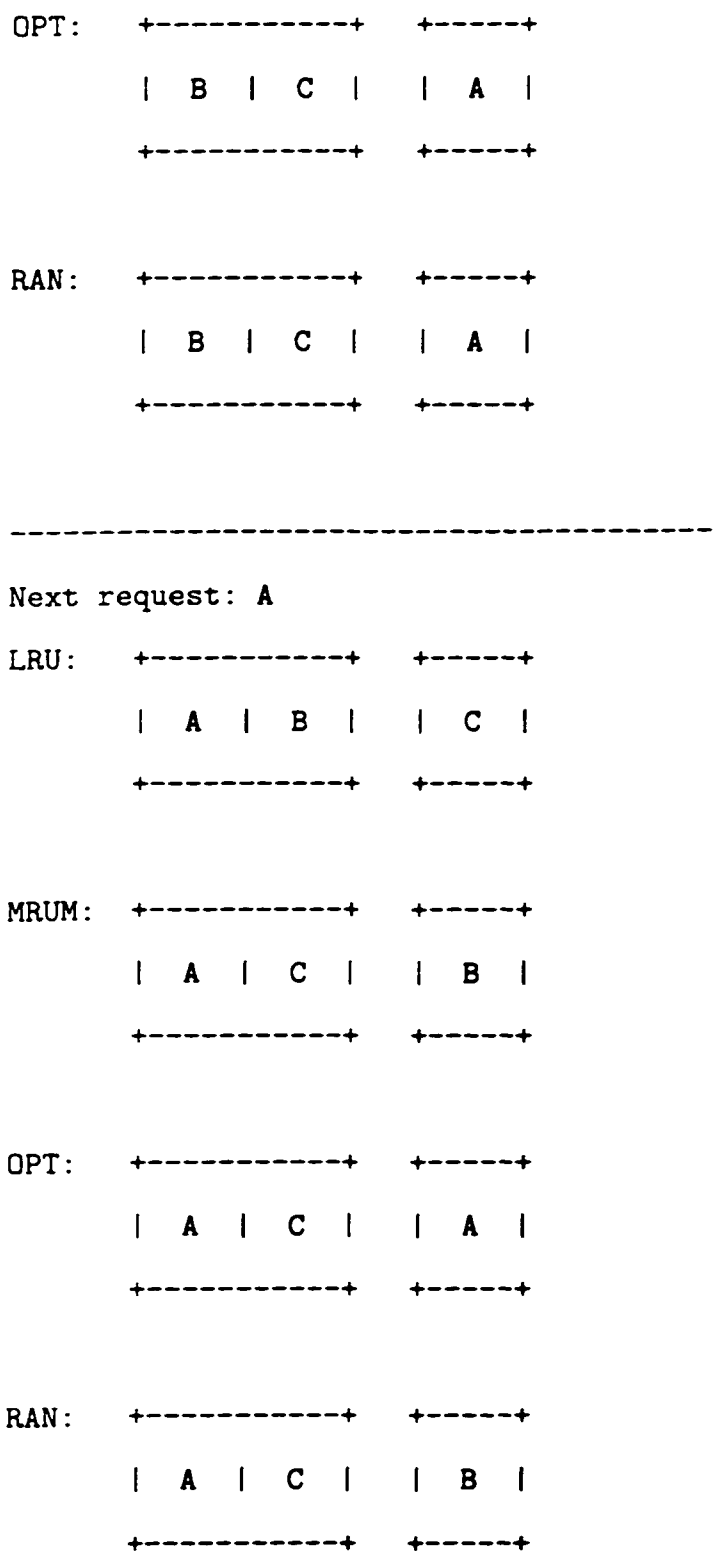


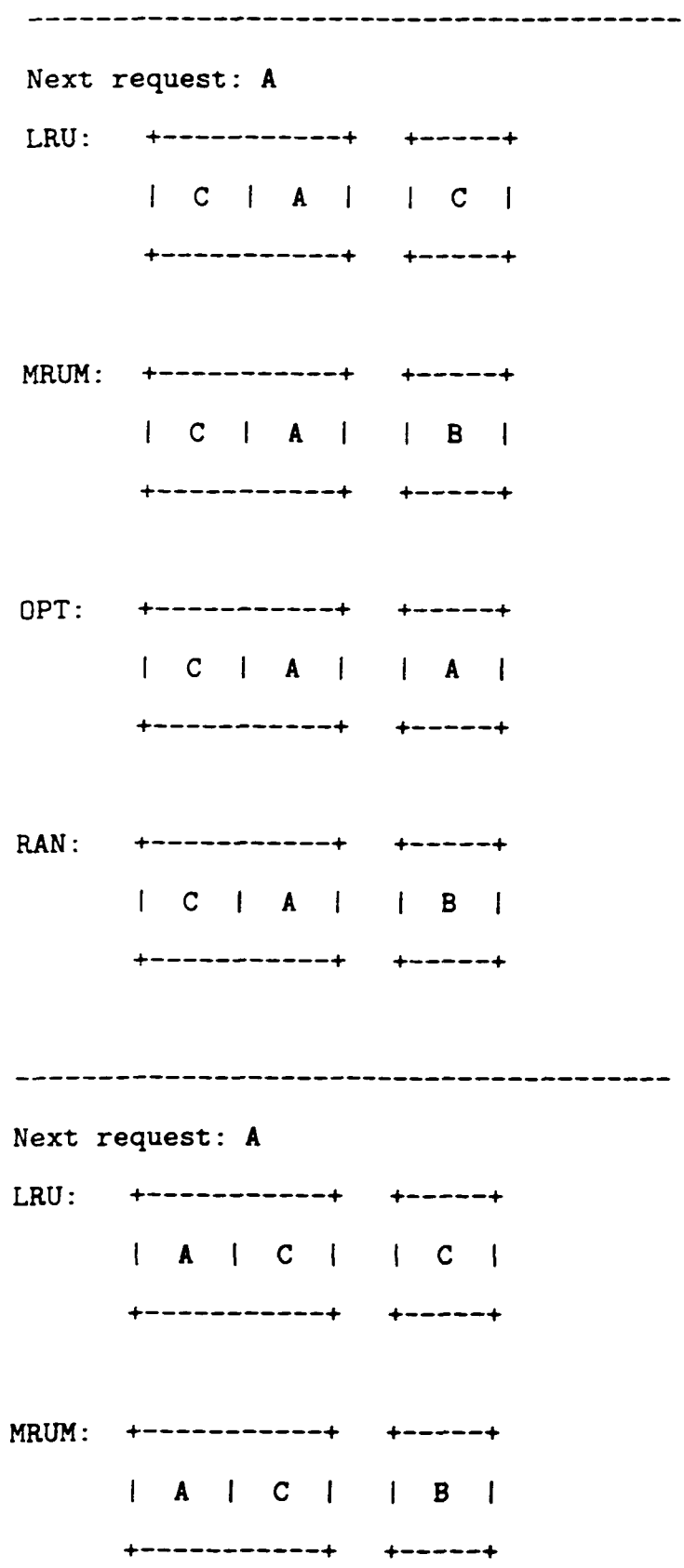
Next request: B



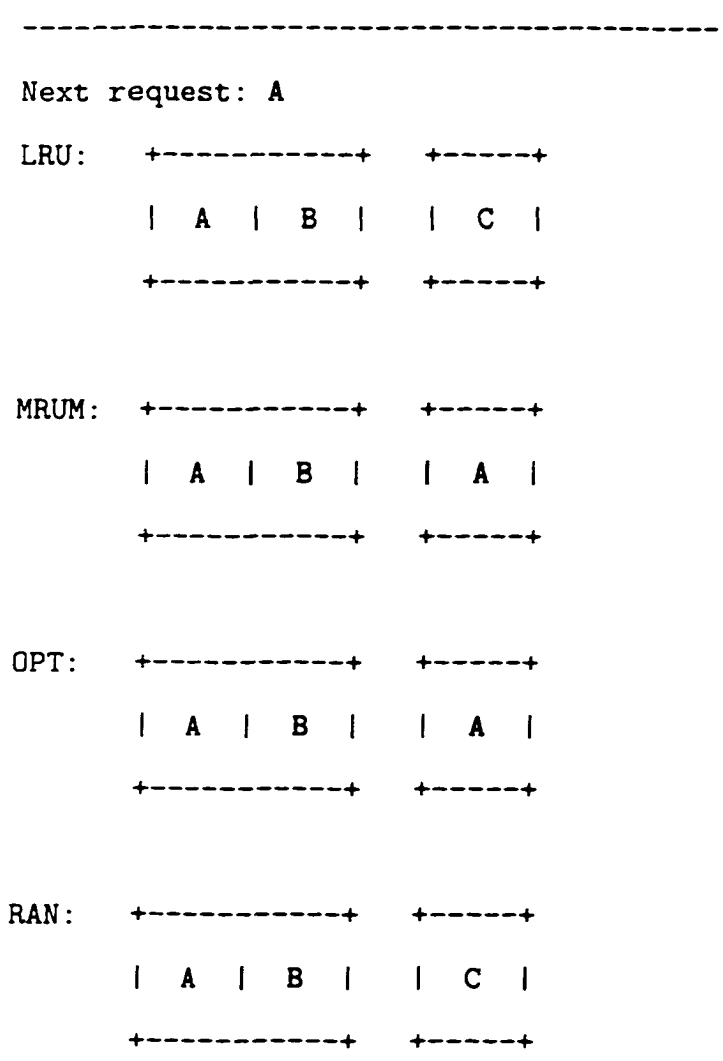
```
----------------------------------------

Cost    LRU: 8, MRUM: 4, OPT: 4 RAN: 5


            COMBINED WORST MRUM / WORST LRU SEQUENCE.
            ========================================

Please enter the length of the input string: 8
INPUT_STRING = CBAACABA

Alg     Junior Senior   Bookmark
=======================================
LRU:    +-----------+   +-----+
        |  A  |  B  |   |  A  |
        +-----------+   +-----+

MRUM:   +-----------+   +-----+
        |  A  |  B  |   |  A  |
        +-----------+   +-----+

OPT:    +-----------+   +-----+
        |  A  |  B  |   |  A  |
        +-----------+   +-----+

RAN:    +-----------+   +-----+
        |  A  |  B  |   |  A  |
        +-----------+   +-----+
```







Next request: B



Cost LRU: 5, MRUM: 4, OPT: 3 RAN: 3

BIBLIOGRAPHY

[1] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. In Proc. 4th European Symp. on Algorithms, volume 1136 of Lecture Notes in Computer Science, pages 419–430. Springer, 1996.

[2] Wolfgang W. Bein, Rudolph Fleischer, and Lawrence L. Larmore. Limited bookmark randomized on-line algorithms for the paging problem. Information Processing Letters. Submitted.

[3] Wolfgang W. Bein and Lawrence L. Larmore. Trackless online algorithms for the server problem. Information Processing Letters. To appear.

[4] Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. IBM Systems Journal, 5:78–101, 1966.

[5] Allan Borodin and Ran El-Yaniv. Online Computation and Competitive Analysis. Cambridge University Press, 1998. `http://www.cup.org/Titles/56/0521563925.html`.

[6] Daniel Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. Communications of the ACM, 28:202–208, 1985.

[7] Neal Young. Competitive paging and dual-guided on-line weighted caching and matching algorithms. PhD thesis, Department of Computer Science, Princeton University, 1991.

VITA

Graduate College
University of Nevada, Las Vegas

Edward Benjamin Mikhalkov

Local Address:
1600 East University Avenue, Apt. 105
Las Vegas, Nevada 89119 U.S.A.

Home Address:
922 - 50th Avenue East
Vancouver, BC, Canada V5X 1B5

Degrees:
Bachelor of Science (Honours), Computer Science, 1997
University of British Columbia

Special Honors and Awards:
Lioness Club International Award & Scholarship, 1992
Kes Chetty Memorial Scholarship, 1993

Dissertation/Thesis Title:
Trackless On-Line Paging and Computer Memory Management

Dissertation/Thesis Examination Committee:
Chairperson, Dr. Wolfgang W. Bein, Ph. D.
Committee Member, Dr. Lawrence L. Larmore, Ph. D.
Committee Member, Dr. Laxmi Gewali, Ph. D.
Graduate Faculty Representative, Dr. George Miel, Ph. D.