Reordering Buffer Management - Theory and Experiments

by

Medha Vasanth

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science
in

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
December 2014

Abstract

Type your abstract here TODO

Acknowledgments

Acknowledgements go here.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Sorting Buffers or the Reordering Buffer Management Problem was first introduced in 2002 by Räcke, Sohler and Westermann, [1]. In this problem, a stream of N items which are characterized by the same attribute are to be processed by a server. A change in the attribute value between two consecutive items incurs a cost. For the sake of simplicity we assume the attribute to be it's "colour". Räcke et al. proposed that this cost can be minimized by placing a k ($k < N$) slot random access buffer before the server. The buffer can be used to permute or reorder the input stream in such a way that we have long sub-sequences of items of the same colour in the output sequence thereby minimizing the the cost of switching between colours. Our goal is to reorder the input sequence using the buffer such that the cost of switching between colours is minimized.

1.1 Motivation

This problem has many applications in computer science and economics. Recent advances in the area of computer game development has led to many sophisticated games that consist of rendering millions of 3D polygons with different attributes which involves switching between different attributes like colour and texture of the

polygons being rendered. Constant switching between attributes while rendering can cause delays and disrupt the smooth rendering of different scenes in a video game. In this case it would be beneficial to render all items of one attribute before the other since the cost incurred by switching between textures and colours is substantial. This has necessitated the need for cost efficient strategies to enable smooth rendering of multiple attributes in computer games. Reordering buffers can be used to permute the stream of polygons to render large sub-sequences of the same attribute, thereby minimizing the cost and delays in a graphics rendering system.

Another application of the sorting buffers problem arises in network communication system where a server has to send packets to different nodes on a network. Sending packets to a particular node in the network involves a startup cost incurred at the receiving node. In this case it is beneficial for the server to wait for all the packets for a particular node to arrive at the server rather than constant switching between the nodes. The sorting buffer would ensure that the server first has all the packets intended for a particular node before data transmission.

A paradigm shift in data storage techniques has resulted in most companies to store large amounts of data on a cloud based server. Standalone nodes have to store and access files from the server on the cloud. Placing a reordering buffer on the cloud server has the benefit of sending large number of files to the designated machine thereby reducing the cost of switching between different standalone nodes.

Reordering buffers can also be used to improve the performance of file servers. A high capacity file server consists of many files which are either open or closed. Read and write accesses to these files can be performed only when the particular file is open. Opening a closed file incurs a cost. Hence a reordering buffer can be used to reorder read and write accesses in such a way that all reads and writes and performed to a file that is currently open.

A preliminary application of the reordering buffers can be seen in the car painting

plant where a sequence of cars are to be painted with a final colour. A cost is incurred if two consecutive cars are to be painted with different colours. This cost can be minimized by reordering the sequence of the cars to be painted using a reordering buffer.

1.2 Problem Definition

In the *Reordering Buffer Management* problem, we have an input sequence of N items, each of these items are characterized by a particular attribute, which we refer to as the colour of the item. The defining characteristic of this problem is that there is a cost incurred when switching between two items of different colours. There are C different colours that can occur in the input stream. Our goal is to permute this input stream in such a way that the cost of switching between items of different colours is minimized.

Räcke, Sohler and Westermann [1], proposed that this problem can be solved by using a $k(k < N)$ slot random access reordering buffer that can be used to permute the input sequence. Every incoming item is first stored in the reordering buffer, when the buffer gets filled up with the first k items, then we choose to *evict* an item from the buffer and add it to the output sequence making room for the next item from the input sequence. We evict one item at each step of the algorithm. The colour to be evicted from the buffer is based on the reordering algorithm in question. This process of selecting the colour to evict and adding it to the output sequence is performed until the whole input sequence is processed and the buffer is empty.

We only consider lazy algorithms for our evaluation, a lazy algorithm satisfies the following two characteristics:

- if there is an item in the buffer with the current output colour, then the lazy algorithm does not make a colour change

- if there are vacant slots in the buffer, then the lazy algorithm does not remove items from the buffer

All reordering algorithms should satisfy the condition that the buffer remains empty at the beginning and end of the reordering, that is all the input items have to be processed by the buffer and added to the output sequence.

1.3 Contributions

In this thesis, we implement different existing reordering buffer algorithms and propose a new algorithm for reordering buffers. We also perform various experiments that study the behaviour of different algorithms for different kinds of input sequences, buffer sizes, colours and cost functions. We use the following parameters for our experiments:

- Input Sequences Types - Alternation, Delta, Random, Random Block and Sequential Block Sequence (refer to Sec 5.1 for details)
- Input Sequence Sizes
- Buffer Size
- Number of Colours
- Cost Function Type - Uniform, Cost Equals Colour, Cost Equals Quadratic Colour, Random Cost, Colour Difference (refer to Sec 5.2 for details)

The insight gained from these experiments have helped design a new algorithm that works well for different application scenarios. We also recommend suitable algorithms for different application scenarios listed in Sec 1.1.

1.4 Results Summary

Summary of results go here

Table 1: Summary of Experimental Results

Algorithm	Sequence Types	Cost Function	Applications	Competitive Rati
Bounded Waste		Uniform		$O(\log^2 k)$
Maximum Adjusted Penalty				$O(\log k)$
Random Choice				
Round Robin				
Threshold or Lowest Cost				$O(\frac{\log k}{\log \log k})$
New Algorithm				
Optimal Offline LP				

1.5 Organization of the thesis

The rest of the thesis is organized as follows. In Chapter 2, we review related literature. In Chapter 3, we present the pseudo-codes for the algorithms that we implement and propose our new algorithm for the Reordering Buffer Management problem. In Chapter 5, we describe the implementation details and the experimental results for the different algorithms. Chapter 6 concludes the thesis with a summary of our results and directions for future work.

Chapter 2

Previous Work

2.1 Reordering Buffers

The problem of Reordering Buffer Management was first introduced by Räcke, Sohler and Westermann in 2002 [1]. It was initially called the *Sorting Buffers* problem, where a service station had a buffer called the sorting buffer which was used to permute the input sequence. Each item in the input sequence was characterized by a particular attribute, which we refer to as the "colour" of the item. A random access buffer was used to permute these items such that the output sequence had long sub-sequences of items with the same colour. There was also a cost involved in switching from one colour to another. In their model, the cost of switching between colours was set to be uniform, in that the cost of switching to any colour was the same. Their model is also based on the *online* version of the problem where the input sequence is not known in advance. So the buffer has to work with partial knowledge of the input sequence. In their paper, the prove that strategies like FIFO, LRU and LCF are not suitable for the sorting buffers problem and prove a lower bound of $\Omega(\sqrt{k})$ for FIFO and LRU, and a lower bound of $\Omega(k)$ for LCF, where k is the buffer size. They also propose a deterministic algorithm called *Bounded Waste* and as with all online algorithms they use *competitive analysis* and prove that their algorithm has a competitive ratio of

$O(\log^2(k))$.

Krokowski *et al.* [2], provide experimental results of the strategies described in [1] for rendering images with different colours and textures. In addition they also propose the *Random Choice* and *Round Robin* strategy where the former randomly selects an item from the buffer to evict and the latter uses a selection pointer and evicts the item pointed to by the selection pointer. Their experiments reveal that Bounded Waste, Random Choice and Round Robin essentially achieve the same performance with Bounded Waste giving an additional 3% of reduced state changes. They have reported the results for Round Robin as this is the most simplistic strategy to implement. They also found that this approach reduces the rendering time by 35% making it time efficient. Simple data structures make this strategy to be efficiently implemented in both hardware and software.

While Bounded Waste was an efficient strategy for uniform costs, it performed poorly under the non-uniform cost model. Englert and Westermann [3], proposed a new deterministic strategy called *Maximum Adjusted Penalty* to handle non-uniform costs. If i denotes that item from the input sequence that is currently being processed, $c(i)$ denotes the colour of the item, and w_c denotes the cost of the colour c, they defined their cost function to be the following: $w_c = d(c(i), c(i+1))$. This cost function is specifically suited for applications such as disk scheduling and paging. As in the case of Bounded Waste, they also used competitive analysis and prove that the competitive ratio of their strategy is $O(\log(k))$, where k is the size of the buffer. They also prove that with a buffer size of k, any scheduling strategy can reduce the cost of switching by at most $(2k-1)$. While the competitive ratio was improved to $O(\log(k))$, this strategy is computationally intensive.

A slight variation of the problem was considered by Kohrt and Pruhs [4], in their version of the problem, the input was a sequence of items of different colours and a random access buffer was used to permute these items. But instead of minimizing

the number of switches in the output sequence, they considered the maximization objective where their goal was to maximize the number of switches that were eliminated from the input sequence. For example, if the input sequence had 10 items and 9 switches and after permuting the sequence, there were 5 switches, then 4 switches were eliminated from the input sequence. Their objective was to maximize this number. They presented the first polynomial time $\frac{1}{20}$ - approximation algorithm for the maximization problem in the offline setting of the sorting buffers problem.

9-approximation algorithm for sorting buffers [5]

A Constant factor Approximation Algorithm for Reordering Buffer Management [6]

An optimal randomized online algorithm for reordering buffer management [7]

Evaluation of online strategies for reordering buffers [8]

Correlation clustering with penalties and approximating the reordering buffer management problem [9]

An improved competitive algorithm for reordering buffer management [10]

Optimal online buffer scheduling for block devices [11]

Almost tight bounds for reordering buffer management [12]

Generalized reordering buffer management [13]

New Approximations for reordering buffer management [14]

Reordering Buffer Management with Advice [15]

Chapter 3

Existing Algorithms/Pseudocodes

This chapter presents the different algorithms available for the Reordering Buffer Management problem along with their pseudo-codes and a brief insight into their analysis. In Section 3.1 we describe our problem with appropriate mathematical notations that will be used throughout the thesis. Section 3.6 through Section 3.10 describe the different algorithms along with their pseudo-codes.

Table 2: Summary of Algorithms

Algorithm	Cost Function	Competitive Ratio	Thesis Section
First-In-First-Out (FIFO)	Uniform	$\Omega(\sqrt{k})$	3.3
Lease Recently Used (LRU)	Uniform	$\Omega(\sqrt{k})$	3.4
Largest Colour First (LCF)	Uniform	$\Omega(k)$	3.5
Bounded Waste	Uniform	$O(\log^2 k)$	3.6
Maximum Adjusted Penalty	Non-Uniform	$O(\log k)$	3.7
Random Choice	Uniform	not known yet	3.8
Round Robin	Uniform	not known yet	3.9
Threshold or Lowest Cost	Non-Uniform	$O(\frac{\log k}{\log \log k})$	3.10
Optimal Offline LP	Uniform	$O(1)$	**??**
New Algorithm	Non-Uniform	not known yet	4.2

3.1 Preliminaries

Our model consists of a server which receives the input sequence and a reordering buffer. Our input sequence has N items and we denote the sequence as $1 \ldots N$. Each item in the input sequence is characterized by an attribute and the input sequence needs to be reordered based on this attribute. For our model we assume that the attribute to be processed is the colour of the item. In our notation, C denotes the number of colours that can be present in the input sequence. We use the notation B to denote the random access buffer which is a linked list of size k ($k < N$). We use the notation i to denote the *current input item*, which is the first item in the input sequence that has not yet been processed. The colour of the current input item is denoted by $c(i)$. This item is i is added to the buffer B after which it will be processed. The reordering buffer B generates the output sequence which is a permutation of the input sequence denoted as $\pi\{1 \ldots N\}$. Note that the reordering buffer is initially empty and must end up empty after the permutation. The colour which is currently assigned to the output sequence is called the *current active colour*. We use the notation c_{active} to denote the current active colour.

A continual sub-sequence of items with the same colour in the output sequence is called a *colour block*. A *colour change* occurs when there is a colour switch between two sub-sequences of different colours. The cost incurred at this colour change is denoted as C_A, which is the cost incurred for the strategy A.

An online strategy is a strategy in which the input sequence is not known in advance. We only consider online strategies for our experiments. Online strategies are evaluated using competitive analysis where every strategy is compared against an optimal offline strategy. An offline strategy is where the input sequence is known in advance. Let C_{OPT} denote the cost of the optimal offline strategy, then an online strategy is said to be *c-competitive* if it produces a cost which is at most of $c \times C_{OPT}$,

that is $C_A = c \times C_{OPT}$.

3.2 Generic Pseudocodes/Generic Methods

In this section we present the basic structure and the common methods used by most of the reordering buffer management algorithms presented in this thesis. Unless otherwise specified, the algorithms use the structure as presented in this section. The *add* and *remove* methods are also common to most of our algorithms. These are also presented in this section.

The pseudo-code below presents the basic structure for most of our reordering buffer management algorithms:

Algorithm 1: Basic Reordering Buffer Management Structure

Input: Input stream $I = 1 \dots N$
Buffer B of size k
Output: Output Stream $O = \pi\{1 \dots N\}$

```
1  Initialize the buffer, B ← NIL ;
2  while |B| ≤ k do
3      add (i) ;
4  Initialize c_active ← −1 ;
5  for i ← k + 1 to N do
6      if c_active ∉ B then
7          c_active ← ChooseNewActiveColour(B) ;
8      remove (c_active) ;
9      add (i) ;
10 while |B| ≠ NIL do
11     if c_active ∉ B then
12         c_active ← ChooseNewActiveColour(B) ;
13     remove (c_active) ;
```

The pseudo-codes for the *add*, *remove* and the *ChooseNewActiveColour* methods are as given below. The details of the *ChooseNewActiveColour* method depends on the algorithm in question.

Function add(i)

Input: i, the input item to be added

1 add i to the end of the list ;

Function remove(c_{active})

Input: colour c_{active}

1 remove the first item of colour c_{active} from the buffer ;

3.3 First In First Out

The FIFO strategy operates by assigning time stamps to the items stored in the buffer. When an item σ_i is to be stored in the buffer, the FIFO strategy checks if the colour $c(\sigma_i)$ already exists in the buffer, if it does, then the item σ_i gets the same time stamp as the other items with the colour $c(\sigma_i)$, otherwise it gets a new time stamp. Time stamps are updated according to the current system time. After the buffer is full, the FIFO strategy starts evicting items that have the oldest time stamp in the buffer. The pseudo-code for the FIFO strategy is presented below:

Racke, Sohler and Westermann [1], present an $\Omega(\sqrt{k})$ lower bound on the competitive ratio of FIFO. Assume $l = \sqrt{k-1}$ is an even integer, with the set of colours $c_1, c_2 \ldots c_l, x, y$ and the input sequence $\sigma = (c_1 c_2 \ldots c_l x^k c_1 c_2 \ldots c_l y^k)^{l/2}$. Whenever a colour block appears in the input sequence, the FIFO strategy removes all the previous items from the sorting buffer, since they are all "older". It can be seen that for each of these blocks this results in l colour changes. Hence according to the FIFO strategy, at least $2l.l/2 = k - 1$ colour changes occur in the output sequence.

3.4 Least Recently Used

Like FIFO, the LRU strategy also assigns time stamps to the items in the buffer. Initially the time stamps for all the items in the buffer are set to zero. When an item

Function ChooseNewActiveColour(B)

Input: Buffer B of size k

Output: New active colour c_{active}

```
1 algorithm specific action ;
2 return c_active ;
```

Algorithm 2: FIFO

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

```
1  Initialize the buffer, B ← NIL ;
2  for i ← 1 to n do
3      if |B| < k then
4          add (σ_i) ;
5      else
6          c(σ) ← colour with the oldest time stamp in B ;
7          remove (c(σ)) ;
8          add (σ_i) ;
9  while |B| ≠ NIL do
10     c(σ) ← the colour with the oldest time stamp in B ;
11     remove (c(σ)) ;
```

σ_i is to be stored in the buffer, the LRU strategy checks if the colour $c(\sigma_i)$ already exists in the buffer, if it does not exist in the buffer, then the item is added to the buffer and it gets a new time stamp, which is the current system time. If an item of colour $c(\sigma_i)$ already exists in the buffer, then the item is added to the buffer and the time stamps of all the items of colour $c(\sigma_i)$ are updated to the current system time. When the buffer is full, LRU removes items that have the oldest time stamp. The pseudo-code for the LRU strategy is presented below:

As in the case of FIFO, the LRU strategy has a competitive ratio of $\Omega(\sqrt{k})$. It can be seen that for the input sequence defined in section 3.3, LRU produces the same output sequence as FIFO.

Function Add-FIFO(σ_i)

Input: σ_i, the input item to be added

if $\exists$ *an item* $x \in B$, *such that* $c(x) = c(\sigma_i)$ **then**
    $t(\sigma_i) \leftarrow t(x)$;
else
    $t(\sigma_i) \leftarrow$ current system time ;
add σ_i to the end of the buffer ;

Function remove($c(\sigma)$)

Input: colour $c(\sigma)$

remove all items of colour $c(\sigma)$ from the buffer ;

Algorithm 3: LRU

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

Initialize the buffer, $B \leftarrow NIL$;
for $i \leftarrow 1$ **to** n **do**
    if $|B| < k$ **then**
        `add` (σ_i) ;
    else
        $c(\sigma) \leftarrow$ colour with the oldest time stamp in B ;
        `remove` $(c(\sigma))$;
        `add` (σ_i) ;
while $|B| \neq NIL$ **do**
    $c(\sigma) \leftarrow$ the colour with the oldest time stamp in B ;
    `remove` $(c(\sigma))$;

Function Add-LRU(σ_i)

Input: σ_i, the input item to be added

if $\exists$ *an item* $x \in B$, *such that* $c(x) = c(\sigma_i)$ **then**
    $t(\sigma_i) \leftarrow$ update time stamp to current time $\forall c(x) = c(\sigma_i)$;
else
    $t(\sigma_i) \leftarrow$ current system time ;
add σ_i to the end of the list ;

3.5 Largest Colour First

The Largest Colour First strategy adds items into the buffer until the buffer is full. When the buffer is full, it selects the colour $c(\sigma)$, which has the maximum number of items in the buffer and removes all items of that colour, thereby clearing as many slots in the buffer as possible. The pseudo-code for the LCF strategy is presented below:

Algorithm 4: LCF

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

```
1  Initialize the buffer, B ← NIL ;
2  for i ← 1 to n do
3      if |B| < k then
4          add (σ_i) ;
5      else
6          c(σ) ← colour with maximum items in B ;
7          remove (c(σ)) ;
8          add (σ_i) ;
9  while |B| ≠ NIL do
10     c(σ) ← the colour with maximum items in B ;
11     remove (c(σ)) ;
```

Function Add-LCF(σ_i)

Input: σ_i, the input item to be added

```
1  add σ_i to the end of the list ;
```

3.6 Bounded Waste

The Bounded Waste strategy selects a current output colour $c(\rho_i)$ and continues to remove items of this colour until the buffer contains no items of this colour. To decide

the next current output colour, the strategy assigns a *penalty* P_c to the colours in the buffer. The penalty for colour c is defined as the number of occurrences of colour c in the buffer. Initially the penalties for all the colours are set to zero. At each colour change, the penalty for each colour P_c is increased by the number of occurrences of colour c in the buffer. All the items with colour c' which has the maximum penalty $P_{c'}$ are removed from the buffer. The pseudo-code for the Bounded Waste strategy is presented below:

Algorithm 5: Bounded Waste

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

```
1  Initialize the buffer, B ← NIL ;
2  for i ← 1 to n do
3      if |B| < k then
4          add (σ_i) ;
5      else
6          if c(ρ) ∈ B then
7              c(σ) ← c(ρ)
8          else
9              c(σ) ← colourWithMaxPenalty (B) ;
10         remove (c(σ)) ;
11         add (σ_i) ;
12 while |B| ≠ NIL do
13     if c(ρ) ∈ B then
14         c(σ) ← c(ρ)
15     else
16         c(σ) ← colourWithMaxPenalty (B) ;
17     remove (c(σ)) ;
```

Function colourWithMaxPenalty(B)

Input: Buffer B of size k

```
1 foreach colour c in the buffer do
2     P_c ← the number of occurrences of colour c
3 c' ← c(max(P_c)) ;
4 P_c' ← 0 ;
5 return c' ;
```

Function Add-BW(σ_i)

Input: σ_i, the input item to be added

```
1 add σ_i to the end of the buffer ;
```

3.7 Maximum Adjusted Penalty

The Maximum Adjusted Penalty strategy works well when each colour has a non-uniform *weight* associated with it. In this case, each colour is has a non-uniform weight b_c associated with it. The MAP strategy works by removing items of the active colour until no more items of this colour are available in the buffer. Like the BW strategy, MAP also assigns penalties to the colours in the buffer. The penalty P_c for colour c is the number of occurrences of colour c which we denote as n_c, times the weight for colour c, i.e. $P_c = n_c.b_c$. The new active colour or the current output colour is selected as follows: for each colour c' in the buffer, the strategy compares if $P_c - k.b_c \geq P'_c - k.b'_c$, if so, then c' is selected as the new active colour and the counter is reset to zero for this colour. The pseudo-code for the MAP strategy is presented below:

3.8 Random Choice

The Random Choice strategy (RC) is a more practical variant of the MAP strategy which avoids the computational overhead of MAP. Items are added into the buffer

Algorithm 6: Maximum Adjusted Penalty

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

```
1  Initialize the buffer, B ← NIL ;
2  for i ← 1 to n do
3      if |B| < k then
4          add (σ_i) ;
5      else
6          if c(ρ) ∈ B then
7              c(σ) ← c(ρ)
8          else
9              c(σ) ← computeMaximumAdjustedPenalty (B) ;
10         remove (c(σ)) ;
11         add (σ_i) ;
12 while |B| ≠ NIL do
13     if c(ρ) ∈ B then
14         c(σ) ← c(ρ)
15     else
16         c(σ) ← computeMaximumAdjustedPenalty (B) ;
17     remove (c(σ)) ;
```

Function computeMaximumAdjustedPenalty(B)

Input: Buffer B of size k

```
1  foreach colour c in B do
2      P_c ← n_c.b_c
3  foreach colour c' in B do
4      foreach colour c in B do
5          if P_c − k.b_c ≥ P'_c − k.b'_c then
6              flag ← true for colour c' ;
7  P_c' ← 0 ;
8  return c' ;
```

until the buffer is full. When the buffer is full, the RC strategy selects the current output colour randomly from the different colours that are currently in the buffer. Appropriate random number generators have to be used to generate random numbers. The RC strategy can also be seen as a randomized variation of the MCF or the BW strategies. The pseudo-code for the RC strategy is presented below:

Algorithm 7: Random Choice

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

```
1  Initialize the buffer, B ← NIL ;
2  for i ← 1 to n do
3      if |B| < k then
4          add (σ_i) ;
5      else
6          if c(ρ) ∈ B then
7              c(σ) ← c(ρ)
8          else
9              c(σ) ← generate random numbers to select a colour from B ;
10         remove (c(σ)) ;
11         add (σ_i) ;
12 while |B| ≠ NIL do
13     if c(ρ) ∈ B then
14         c(σ) ← c(ρ) ;
15     else
16         c(σ) ← generate random numbers to select a colour from B ;
17     remove (c(σ)) ;
```

3.9 Round Robin

The Round Robin strategy is an efficient variant of the RC strategy, which uses a selection pointer to select the current output colour. When the next active colour is to be chosen, it advances the selection pointer to the next item in the buffer, this

colour is chosen to be the current output colour and all items this colour are removed until there is no more item of this colour to be removed. This strategy can also be modified to be a variation of the MCF or the BW strategy. The pseudo-code for the RR strategy is presented below:

Algorithm 8: Round Robin

Input: Input stream $\sigma = \sigma_1, \sigma_2 \ldots \sigma_n$
Buffer B of size k
Selection Pointer p which can have values between $1, 2 \ldots k$

Output: Stream $\rho = \rho_1, \rho_2, \ldots \rho_n$

```
1  Initialize the buffer, B ← NIL ;
2  for i ← 1 to n do
3      if |B| < k then
4          add (σ_i) ;
5      else
6          if c(ρ) ∈ B then
7              c(σ) ← c(ρ)
8          else
9              p ← p + 1 if p > k then
10                 p ← p%k ;
11             c(σ) ← c(p) ;
12         remove (c(σ)) ;
13         add (σ_i) ;
14 while |B| ≠ NIL do
15     if c(ρ) ∈ B then
16         c(σ) ← c(ρ) ;
17     else
18         c(σ) ← c(p) ;
19     remove (c(σ)) ;
```

3.10 Threshold or Lowest Cost

The Threshold or Lowest Cost (TLC) [10], strategy evicts one item at each step of the algorithm. For the first k steps the input items are stored in the sorting buffer, and

the elimination happens one element at each step from steps, $k+1, k+2, \ldots k+n$. We use the notation B_j to denote the contents of the buffer at step j. The algorithm maintains a counter for each input item in the buffer, we denote this counter as ϕ_j^i, which is element $i's$ counter at step j. The counters ϕ_j^i are monotonically non-decreasing in j and $\phi_j^i = 0$ for all $j < i$. The counters are only updated when we make a paid colour as described below.

At each step, the algorithm selects an active colour and removes one item of this colour from the buffer at each step until the buffer has no more items of this colour. At this point the algorithm selects a new active colour, and this is done as follows: if there is a colour c in the buffer such that $\sum_{i \in B_j \wedge c(i)=c} \phi_j^i \geq w(c)$, then choose $\sum_{i \in B_j \wedge c(i)=c} \phi_j^i \geq w(c)$, any such colour c to be the new active colour. If there is no such colour c, then select the colour c with the lowest cost $w(c)$ and update the counters and follows: add the quantity $\frac{w(c)}{k}$ to all the items in the buffer. The pseudo-code for the TLC strategy is presented below:

Algorithm 9: TLC

Input: Input stream $I = 1 \dots N$
Buffer B of size k

Output: Output Stream $O = \pi\{1 \dots N\}$

Initialize the buffer, $B \leftarrow NIL$;
while $|B| \leq k$ **do**
    `add` (i) ;
Initialize $c_{active} \leftarrow -1$;
for $i \leftarrow k+1$ **to** N **do**
    if $c_{active} \notin B$ **then**
        $c_{active} \leftarrow$ `TLCActiveColour`(B) ;
    `RemoveTLC` (c_{active}) ;
    `add` (i) ;
while $|B| \neq NIL$ **do**
    if $c_{active} \notin B$ **then**
        $c_{active} \leftarrow$ `TLCActiveColour`(B) ;
    `RemoveTLC` (c_{active}) ;

Function TLCActiveColour(B)

Input: Buffer B of size k

Output: New active colour c_{active}

for *every item* $i \in B$ **do**
    $S^c = \sum_{c(i)=c} \Phi_i$;
$c_{active} \leftarrow c_{min}$;
for *every item* $i \in B$ **do**
    $\sigma_c = S^c$;
    if $\sigma_c \geq w(c)$ **then**
        $c_{active} \leftarrow c$;
        return c_{active} ;
add $\frac{w(c_{min})}{k}$ to the counter Φ_i of every item in the buffer ;
return c_{active} ;

Function RemoveTLC(c_{active})

Input: colour c_{active}

remove the first item of colour c_{active} from the buffer ;

Chapter 4

New Algorithm

In this chapter, we present our new algorithm for the Reordering Buffer Management problem along with the analysis. Section 4.1 describes the mathematical notations used to formally describe the problem. Section 4.2 describes our algorithm and presents the pseudo-code for our algorithm. Section 4.3 presents our analysis for the algorithm.

4.1 Preliminaries

Staying consistent with the other algorithms, we have used the notation $I = 1 \dots N$ to denote the input sequence, where N is the size of the sequence. B represents the reordering buffer and k denotes the buffer size. Our model uses $C = 1 \dots c$ to represent the number of colours that can be present in the input sequence and the weight/cost for colour c is represented as $w(c)$. We use the notation c_{min} to denote the colour that has the lowest cost. $O = \pi\{1 \dots N\}$ represents our output sequence which is a permutation of the input sequence I.

During one run of the algorithm, i denotes the *current input item*, which is the first item of the input sequence that has not yet been processed. The colour of this item is denoted as $c(i)$. This item has to be added to the buffer B before it can be

output. The colour that has presently been assigned to the output sequence is called the *current active colour* denoted by c_{active}. Since we only consider lazy algorithms, we change the active colour only when we no longer have an item of the colour c_{active} in the buffer.

Our algorithm maintains and updates a monotonically increasing counter which we denote as Φ_i, which is the counter for the item i. At every step where a new active colour is to be chosen, our algorithm inspects the values of this counter for all items of a particular colour c and then selects a new active colour as described in Section 4.2.

4.2 New Algorithm/ModifiedTLC

Our algorithm works as follows: For the first k steps of the algorithm, we add the first k items of the input sequence into the buffer. Since we follow the lazy strategy, the eviction process begins only after the first k steps, that is, we start evicting items from steps $k+1, k+2, \ldots k+N$. Since we only evict one item per step of the algorithm, our buffer is emptied at step $k+N$.

Initially the counter $\Phi_i = 0$, for all items in the buffer. Whenever we have to choose a new active colour, we check if the sum of counters of all items of a particular colour c is greater than the weight for that colour, $w(c)$. That is, $\sum_{c(i)=c} \Phi_i > w(c)$, if so we select colour c to be the new active colour, c_{active}, and evict one item of that colour. If the buffer has no such colour c for which the sum of counters is greater than the weight of the associated colour, then we select the colour with the lowest cost to be the new active colour and increment the counters of all items in the buffer with the value $\frac{w(c_{min})}{k}$.

To optimize the performance of our algorithm, we store the sums of these counters as an array upfront, and track the number of times for which our counters need to

be updated before the active colour is chosen. To do this we use two arrays of size C to store the sums and the count of the items of a particular colour in the buffer. We populate these arrays as a first step when we have to choose a new active colour. Using these aggregated values, we perform one update at the end of our active colour selection process making our algorithm time efficient. Doing this prevents us looping over the buffer to compute values of the sums, thereby reducing our running time to $O(k)$.

The pseudo-code for our algorithm is presented below:

Algorithm 10: ModifiedTLC

Input: Input stream $I = 1 \dots N$
Buffer B of size k

Output: Output Stream $O = \pi\{1 \dots N\}$

```
1  Initialize the buffer, B ← NIL ;
2  while |B| ≤ k do
3      add (i) ;
4  Initialize c_active ← −1 ;
5  for i ← k + 1 to N do
6      if c_active ∉ B then
7          c_active ← ModifiedTLCActiveColour(B) ;
8      RemoveMTLC (c_active) ;
9      add (i) ;
10 while |B| ≠ NIL do
11     if c_active ∉ B then
12         c_active ← ModifiedTLCActiveColour(B) ;
13     RemoveMTLC (c_active) ;
```

4.3 Analysis

Function ModifiedTLCActiveColour(B)

Input: Buffer B of size k

Output: New active colour c_{active}

for *every item* $i \in B$ **do**
    $S^c = \sum_{c(i)=c} \Phi_i$;
    $C^c =$ number of items of colour c in B ;
$c_{active} \leftarrow c_{min}$;
$j = 0$;
for *every item* $i \in B$ **do**
    $\sigma_c = S^c + (C^c \times \frac{w(c_{min})}{k} \times j)$;
    if $\sigma_c \geq w(c)$ **then**
        $c_{active} \leftarrow c$;
        break ;
    $j \leftarrow j + 1$;
add $\frac{w(c_{min})}{k} \times j$ to the counter Φ_i of every item in the buffer ;
return c_{active} ;

Function RemoveMTLC(c_{active})

Input: colour c_{active}

remove the first item of colour c_{active} from the buffer ;

Chapter 5

Implementation and Experiments

In this chapter, we describe our the implementations of our algorithms along with the experimental results obtained. In Section 5.1 we give detailed descriptions of all the data sets we have generated. Section 5.2 describes all the cost functions we have used for our experiments. Section 5.3 gives specific details about the programming language and underlying data structures used for our implementations. In Section **??** we list the hypothesis that we expect to verify with our experiments and present experimental results in Section 5.4.

5.1 Data Sets

We generate input sequences of sizes 100, 1000, 10,000 and 100,000. Each input item in the sequence is assigned a colour within the range $1 \ldots C$. In order to recommend algorithms for various application scenarios we generate data sets that follow a specific pattern as well as random data sets. We have performed experiments on the following data sets for all our algorithms for different size combinations of the input parameters.

- *Random Sequence*: The colours for the input items are generated randomly between the range $1 \ldots C$ with equal probability. An example of a random sequence with 3 colours and input size 10 is as follows: 3, 2, 3, 1, 2, 3, 2, 1, 1,

2.

- *Sequential Block Sequence*: We use the notation m to denote the block size. We assign a colour c, chosen from the range $1 \ldots C$ to all input item belonging to the same block. We have assigned colours to each block, starting from the first block, in an increasing order from 1 going up to C. We repeat the same pattern of colours for the entire size of the input sequence. For our experiments we have used a block size of 5. An example of the sequence for block size 3, 2 colours and input sequence size 10 is as follows: 1, 1, 1, 2, 2, 2, 1, 1, 1, 2.

- *Random Block Sequence*: In this case, we have a block sequence where the size of the blocks are randomly chosen within the range $1 \ldots C$ and this randomly chosen size also corresponds to the colour assigned to each item in that particular block. An example of this sequence for input size 10 and 3 colours is as follows: 3, 3, 3, 1, 2, 2, 1, 1, 2, 2. For this example,

- *Alternation Sequence*: In the alternation sequence, we generate the first item with colour c and assign the colours $c+1$, $c+2 \ldots C$ for each successive item. For the sake of simplicity we have assigned colours in the increasing order of the set of colours starting from 1. We repeat the same pattern until the end of the input sequence. An example of the sequence for 3 colours and input sequence size 10 is as follows: 1, 2, 3, 1, 2, 3, 1, 2, 3, 1.

- *Delta Sequence*: In the delta sequence we generate input items that are always within a certain range. We use the notation Δ to denote this range. For this sequence, we generate the first item c randomly within the range $1 \ldots C$ and starting with the second item, we generate each item in the range of $c \pm \Delta$, where Δ can take the value 0. This ensures that all the items are within the range of Δ from the previous item. An example of the sequence for Δ value 2,

3 colours and input sequence size 10 is as follows: 2, 3, 1, 2, 2, 3, 1, 1, 2, 3.

5.2 Cost Functions

Since some of our algorithms are designed for the non-uniform cost model, we have designed the following cost functions and tested them with all the data sets listed in 5.1 to recommend algorithms for specific application scenarios. Our implementations are designed based on the assumption that we always consider the cost of the colour to which we are switching to, or the next *new active output colour*. Our cost functions are as defined below:

- *Uniform Cost*: In this case the cost assigned to every colour is uniform. For the sake of simplicity we have assumed a unit cost model for our uniform cost function.

- *Cost Equals Colour*: For this cost model, the cost assigned to the colour is the value of the colour itself, for example, an item with colour 3 has a cost of 3.

- *Cost Equals Quadratic Colour*: In this case, the cost assigned to the colour is equal to the square of the value of the colour, for example, the cost of colour 3 is set to be 9. The reason behind having such a cost function is to examine our algorithms' performance when subject to input sequences where the cost of switching is very high and varies between very low and very high costs.

- *Random Cost*: In this case we randomly assign a cost to each colour between the range $1 \ldots R$. For our experiments we have set R to be 5. The reason behind this is to limit the cost assigned to colours to be a low cost and compare our algorithms' performance when the cost of the colours in the input sequence is low.

- *Colour Difference Cost Function*: This is the only cost model that takes into account the colour that we are presently at, and the colour to which we would like to switch to. We call these two colours the *from colour* and *to colour*. Here the cost of switching from the *from colour* to the *to colour* is defined as the absolute difference between the two colours. This cost model is especially applicable in scenarios where the cost of switching depends on the two colours in question. An application of this cost model can be seen in the disk scheduling problem where switching is synonymous to moving the disk head to point to a particular disk block. In this case we want to minimize the distance between moving the head too far from the current point.

5.3 Implementation Details

In this section we describe the assumptions made in our implementations and give a basic overview of our approach to the different algorithms we have implemented. Table **??** lists all the algorithms that we have implemented along with the size of the input sequences, buffer sizes and number of colours. We have performed experiments with different data sets (refer to Section 5.1).

We have primarily used Java with the JavaSE-1.7 as our execution environment for most of our implementations. The native C GNU Linear Programming Kit (GLPK Simplex Optimizer), version v4.52 is used for the Linear Program that we have chosen to implement. Experimental results are presented in Section 5.4.

Our input/output sequences and buffers are implemented as Java Array Lists. We build our input sequences according to the data sets listed in 5.1 by adding an item at the end of the list. Our algorithms also construct the output sequence in a similar manner by appending items to the end of the list. The main intuition behind using

Table 3: Algorithms implemented along with parameter values

Algorithm	Input Sequence Size N				Buffer Size k				Colours C			
Bounded Waste	100	1000	10000	100000	2	10	50	100	2	10	50	100
Maximum Adjusted Penalty	100	1000	10000	100000	2	10	50	100	2	10	50	100
Random Choice	100	1000	10000	100000	2	10	50	100	2	10	50	100
Round Robin	100	1000	10000	100000	2	10	50	100	2	10	50	100
Threshold or Lowest Cost	100	1000	10000	100000	2	10	50	100	2	10	50	100
Optimal Offline LP	100	500	-	-	2	10	50	100	2	10	50	100
New Algorithm	100	1000	10000	100000	2	10	50	100	2	10	50	100

array lists for implementing our sequences and buffers is because it has the property of random access which how our algorithms specify the buffers.

Every item is designed to have the colour, the input time which we set at the time of creation, the output time which specifies the time at which the item was evicted from the buffer and an optional counter which is specific to the algorithm. We begin our input time stamps from 1 and check to ensure that every item has an output time that is greater than the time at which it was input.

We define all the statistics that we would like to measure about our algorithms' performance on the sequences, and compare the input and output sequences to decide the proportion of switches that have been reduced, the proportion of switching cost that has been reduced, the minimum, maximum and the average time that an item spends in the buffer.

Our basic outline for the reordering buffers happens in three stages: *initialization*, *reordering* and *eviction*. In the *initialization* phase, we fill the first k items of the input sequence into the buffer. In the *reordering* phase we first pick a new colour to evict from the different colours presently in the buffer, which we call as the *new active colour* and evict one item of this colour in each time step. We then set the output time on the evicted item, add the item to the output sequence and refill the buffer.

We continue to do this until all the input items are in the buffer have been processed by the buffer. In the *eviction* phase, we empty the buffer and start evicting one item at each step until the buffer is empty. We also set output times and add the item to the output sequence as in the *reordering* phase.

Our assumptions for individual algorithms and implementation details are as follows:

- *Bounded Waste*: Our outline for Bounded Waste differs from the other algorithms, in that when we have chosen the colour that we would like to evict from the buffer, we evict all the items of that colour in one step as opposed to one in each step. This causes many successive items to have the same output time, and we expect that the average time that an item spends in the buffer will also be reduced as a consequence of this. Our implementation follows our understanding of Bounded Waste as explained in Section 3.6.

- *Maximum Adjusted Penalty*

- *Random Choice*: As explained in Section 3.8, we randomly select a colour present in the buffer and remove one item of that colour in each step, until there are no more items of that colour in the buffer. To choose a new active output colour, we randomly select a colour present in the buffer and continue until the input sequence has been processed and the buffer is empty.

- *Round Robin*: For the Round Robin reordering, we initially set the selection pointer to point to the first item in the buffer, which is at index 0. When we have to choose a new active colour, we select the colour pointed to by the selection pointer in the buffer and remove one item of that colour at each step. But since our array list implementation of the buffer compresses the buffer to the left when an item is evicted, it skips a colour and moves to the next colour.

We continue to do this until the input sequence has been processed and our buffer is empty.

- *Threshold or Lowest Cost* 3.10
- *New Algorithm* **??**
- *Optimal Offline LP*: The problem definition for the linear program is as specified in [11] and we have used the GNU Simplex Optimizer and C implementations to generate data sets and matrices for the linear program.

5.4 Experiments and Results

Section 5.3 details all the algorithms we have implemented, along with the details of the implementation, our assumptions for the algorithms and language and programming details. We have performed different experiments on these algorithms for the all the data sets listed in Section 5.1 and all cost functions listed in Section 5.2 to gain some insight to designing a new algorithm. These results are listed in the following subsections.

Table **??** lists all our algorithms, along with the different sizes used for different input parameters. Experiments are performed once for each deterministic algorithm. For randomized algorithms with random data sets and random cost functions, we repeat the experiments 100 times and average the results from all experiments.[1]

For each of our experiments, we record the following values: the algorithm that is being executed, the kind of input sequence used, the cost function, the size of the input sequence, the buffer size, number of colours, count of switches in the input sequence, count of switches in the output sequence, ratio of the output/input sequence

[1]We have used Excel 2010 to analyze all our results and used Excel Pivot Tables to compute averages and plot charts for our experiments.

switches count, cost of the input sequence, cost of the output sequence and the ratio of the output/input sequence cost. We use these ratios to plot our charts since it gives us an insight into how each algorithm performs under different data sets/ cost function combinations. The ratios gives us some insight into the percentage of how many switches were eliminated or how much of the cost was reduced. For example, let us assume that our input sequence has a size of 10, and has 8 switches in the input and 3 switches in the output, our ratio is $3/8 = 0.375$. This indicates that the output sequence has 37.5% of the switches present in the input sequence. We use similar ratios to analyze the cost as well.

Our experimental results reveal that the lengths of the input sequences do not play a significant role in the way our algorithms perform with different buffer sizes, varied number of colours or even different data sets and cost functions. We briefly explain our results as seen with the input sequences and show some graphs to infer that the performance of our algorithms' does not depend on the size of the input sequence. This is illustrated in Fig 1.

While different algorithms have different performance for different combinations of data sets and cost functions, the general trend suggests that they do not heavily depend on the size of the input sequence. This has been true in the case of all kinds of data sets when subjected to all different cost functions as well.

We describe how different algorithms (with uniform and non-uniform costs) perform with the different data sets and cost function combinations in the following subsections and supplement them with appropriate figures.

5.4.1 Uniform Cost

In this section we present the performance of different uniform cost algorithms across different input sequence types and cost functions. In the case of uniform cost algorithms, the number of switches and the cost incurred by the algorithm are the same

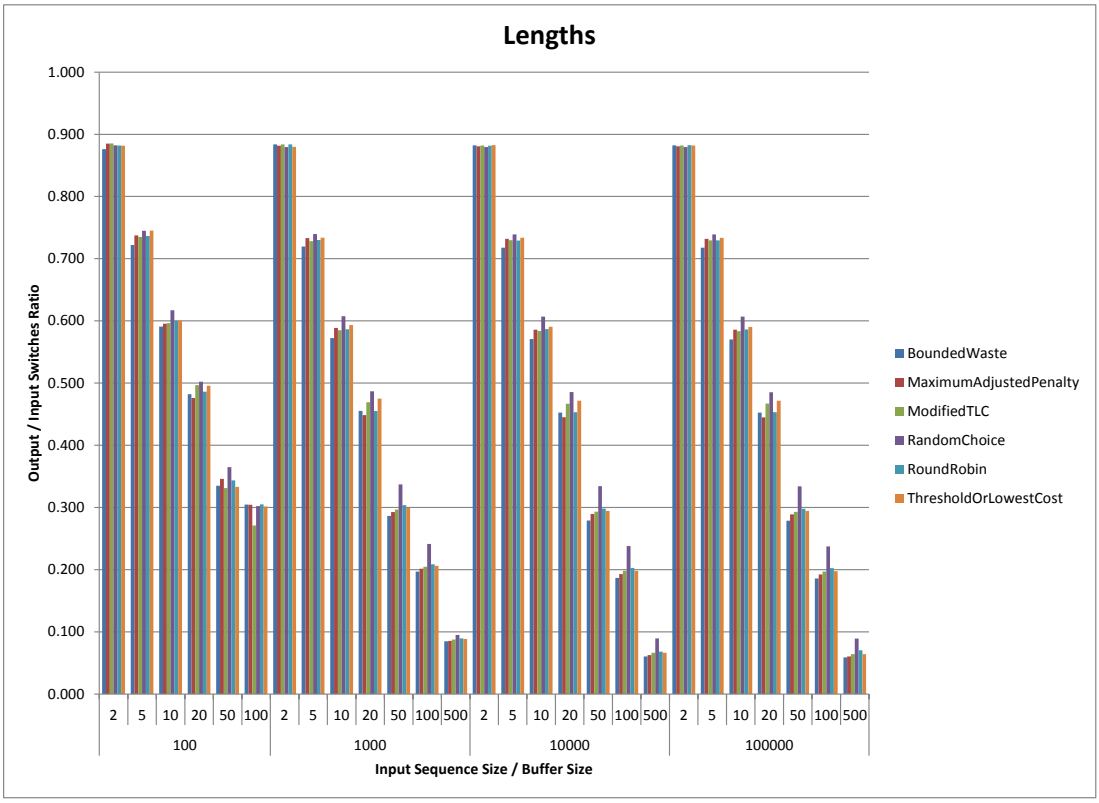

Figure 1: Input Sequence Size does not impact the performance of algorithms

since the exact same cost is incurred when we switch from one colour to the next. For our experiments, the uniform cost incurred on each colour switch is equal to the buffer size (k).

Alternation Sequences

In the case of Alternation Sequences, we observe that Bounded Waste (BW) achieves the best performance among all algorithms for larger buffer sizes. Random Choice (RC) achieves the lowest reordering while Round Robin (RR) achieves a reordering ration comparable to BW for large buffer sizes.

For very small buffer sizes, we observe that all the algorithms fail to achieve any significant reordering with the Output/Input Switch ratio being close to 1.0 in these cases. These results are illustrated in Fig. 2.

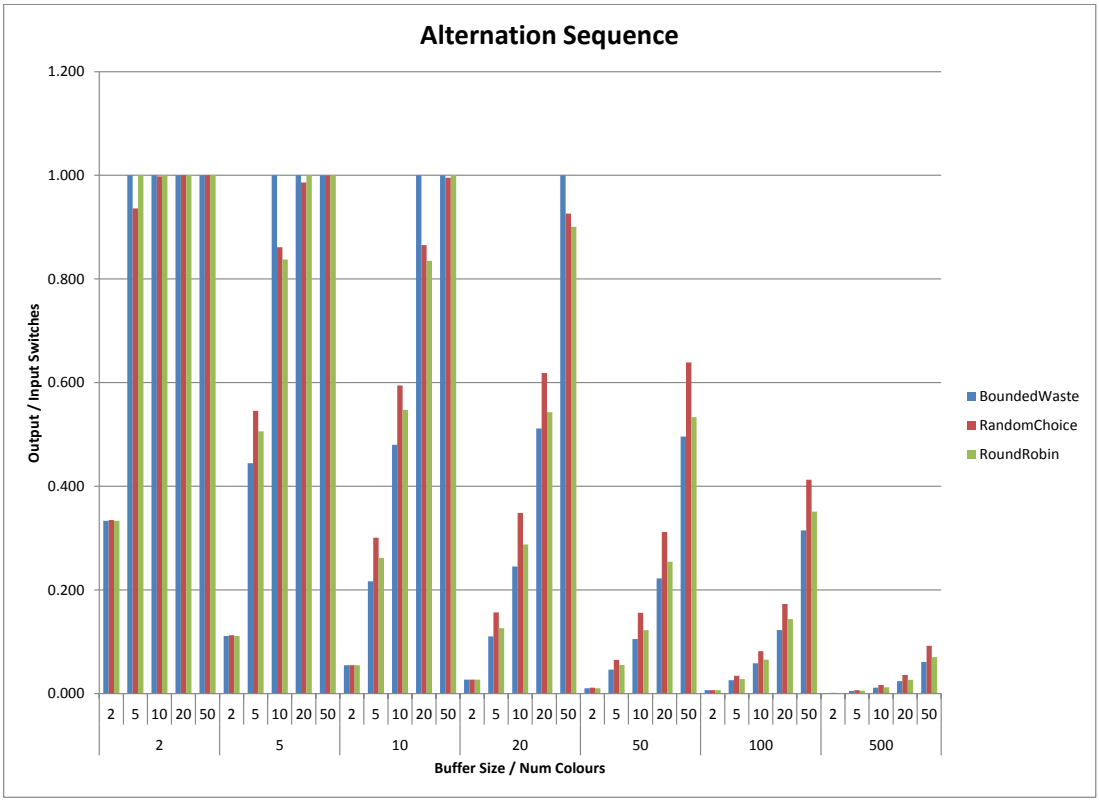

Figure 2: Bounded Waste achieves better performance

Delta Sequences

In general, we observe that Delta Sequences achieve a better reordering ratio than Alternation Sequences across all algorithms. The reordering ratio decreases as the buffer size increases.

We observe that all algorithms, BW, RC and RR achieve comparable performance

for Delta Sequences with BW doing marginally better than the other two algorithms across all buffer size and number of colours combinations. As in the case of Alternation Sequences, RR achieves marginally better performance than RC making it the next choice after BW. These results are illustrated in Fig. 3

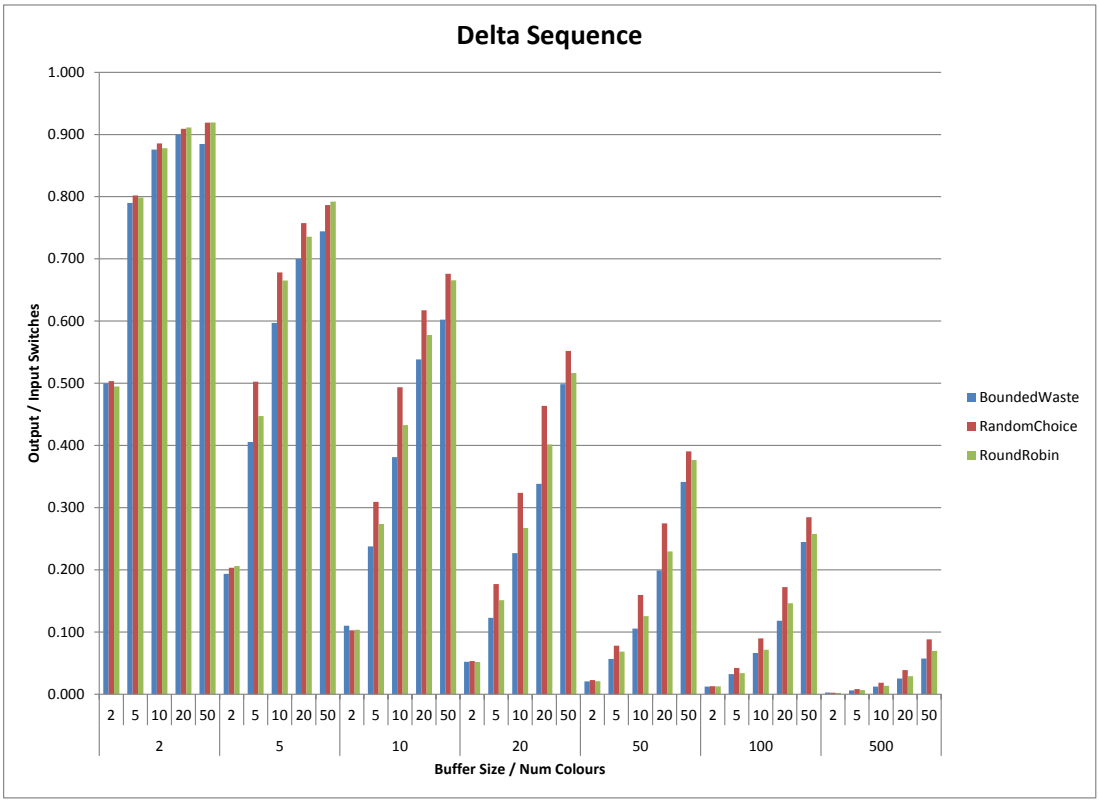

Figure 3: All algorithms achieve comparable performance

Random Sequences

We observe a similar trend as in the case of Delta Sequences across all buffer sizes and cost functions. As in the case of Delta Sequences, BW achieves a slightly better performance than RR and RC, however RR and RC achieve comparable reordering

ratios with RR achieving a marginally better performance than RC. These results are illustrated in Fig. 4.

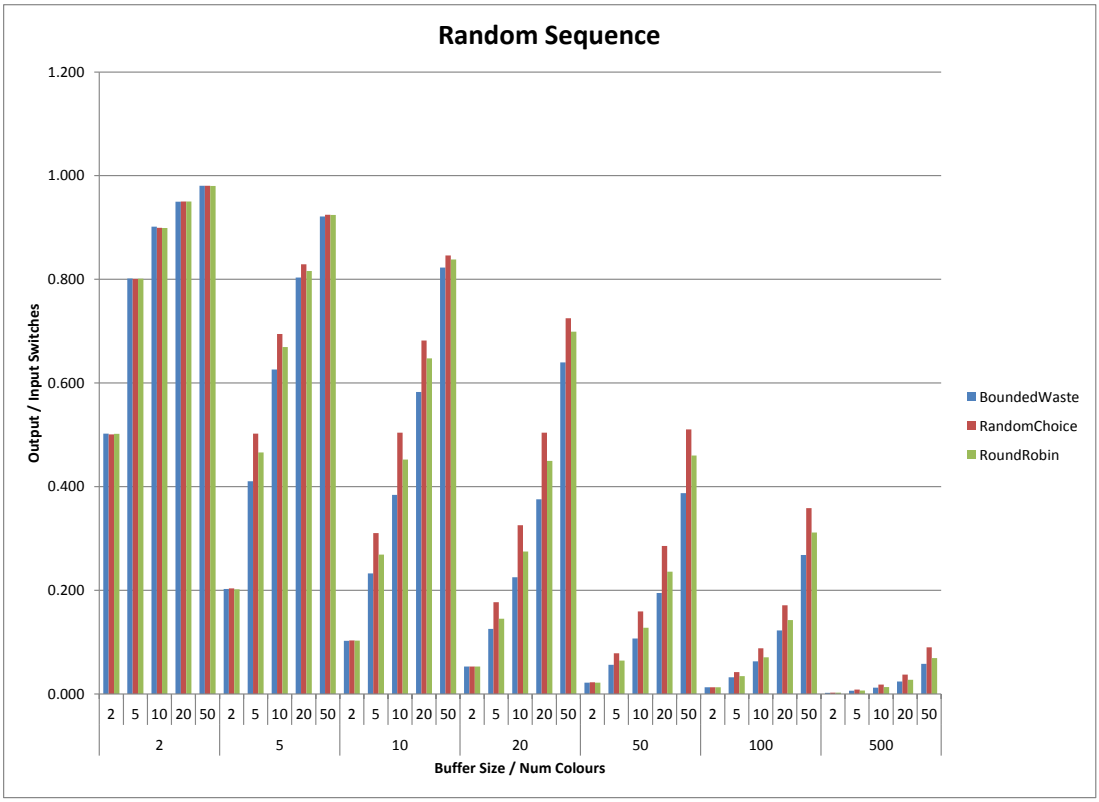

Figure 4: BW achieves better performance, RC and RR are comparable

Random Block Sequences

As expected with block sequences, all our algorithms fail to achieve a good reordering ratio as the number of colours increases, hence the reordering ratio for all our algorithms is closer to 1.0 as the number of colours increases.

For small number of colours and large buffer sizes, BW achieves a good performance followed by RR and then RC, as is the case with other input sequences as well.

These results are illustrated in Fig. 5.

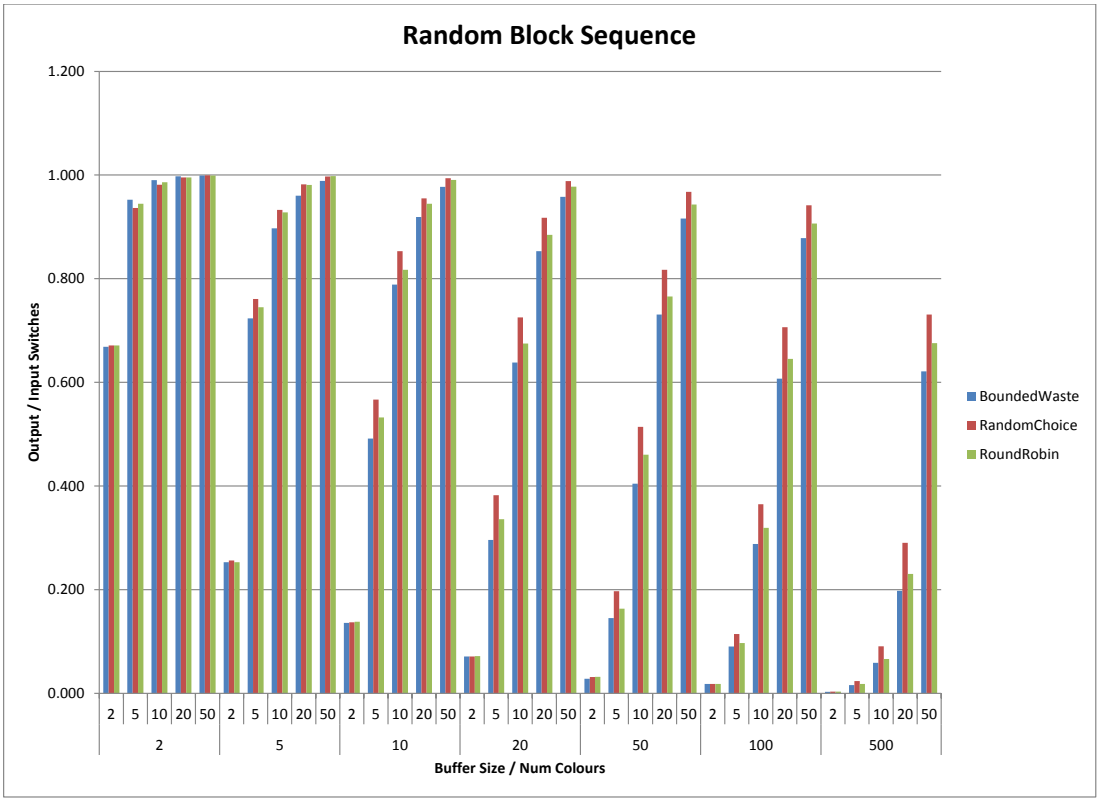

Figure 5: Reordering Ratio is small for large number of colours

Sequential Block Sequences

Since our block sizes are 5, we see no reordering for very small buffer sizes of 2 and 5. This is expected since only items of a single colour fill the buffer making reordering impossible for extremely small buffer sizes.

Unlike other input sequences, BW fails to achieve the best performance across some buffer sizes making it unsuitable for application scenarios where the input items come in fixed sized blocks. While RR and RC do achieve some reordering, BW fails

to reorder the input sequence making it unsuitable in these cases.

For large buffer sizes, all our algorithms achieve comparable performance and in these cases, BW does marginally better than RR and RC. These results are illustrated in Fig. 6.

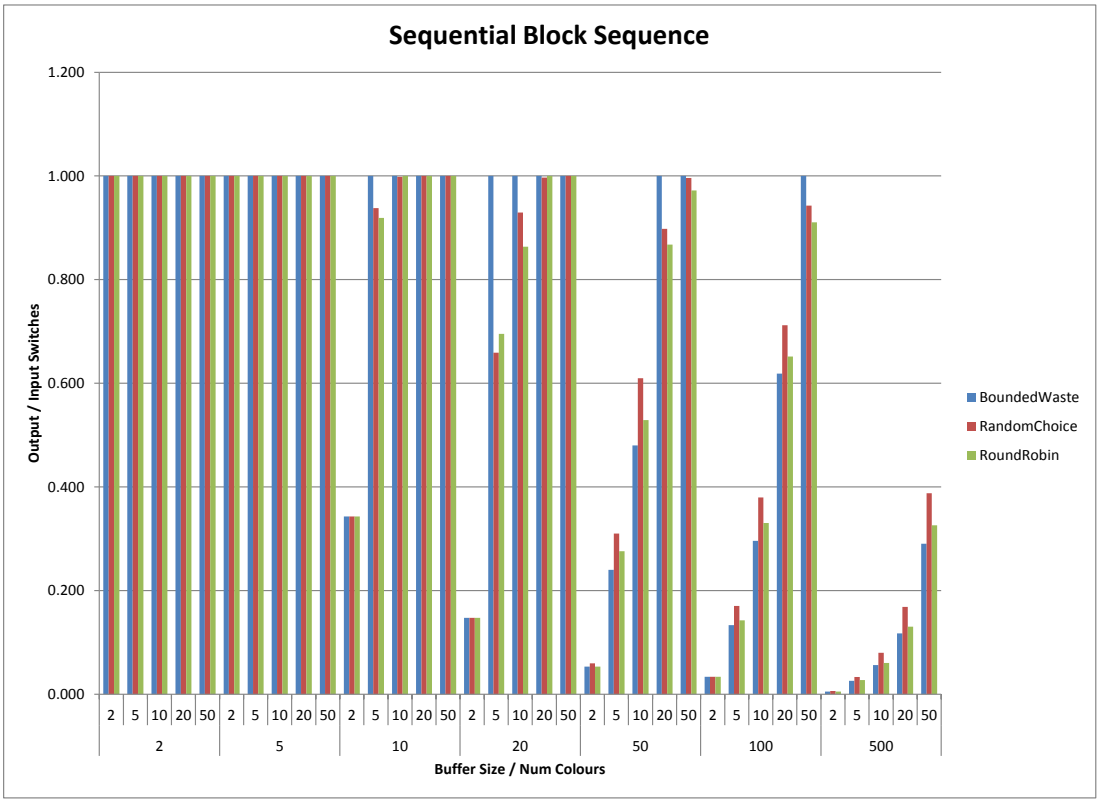

Figure 6: Variation in the reordering ratios across different buffer sizes

5.4.2 Non-Uniform Cost

Alternation Sequences

For Alternation Sequences with the Cost Equals Colour Cost Function, we see little or no reordering for very small buffer sizes (2 and 5). We observe that TLC achieves a better reordering switch ratio than MAP and TLC' when the number of colours is approximately two times the buffer size ($C = 2 * k$). In other cases, we observe that TLC' and MAP achieve comparable performance with TLC' slightly better than MAP. When comparing cost ratios, MAP and TLC' fail to achieve a good cost ratio for small buffer sizes, TLC' performs marginally better than MAP and TLC when the number of colours is lesser than or equals to the buffer size. As with switch ratio, TLC achieves a better performance when the number of colours is two times the buffer size. All out algorithms do not perform well when the number of colours are significantly larger than the buffer size.

For the Cost Equals Colour Quadratic model, TLC' performs better than MAP and TLC when the number of colours is less than or equal to the buffer size, this is the same as the Cost Equals Colour Model, however, an exception to this case is buffer size 2 and 5 colours where TLC' achieves a better performance than MAP and TLC. TLC achieves a good performance when the number of colours is greater than or equal to two times the size of the buffer. These results are illustrated in Fig. 7. TLC' achieves a better switch ratio than MAP or TLC when the number of colours is less than or equal to the number of colours, MAP and TLC achieve comparable performance in this case.

A similar trend is seen when comparing the algorithms against a Random Cost Function, TLC' achieves a better switch ratio than MAP and TLC when the number of colours is less than or equal to the size of the buffer. MAP achieves a better performance than TLC in this case. TLC achieves a better performance than MAP

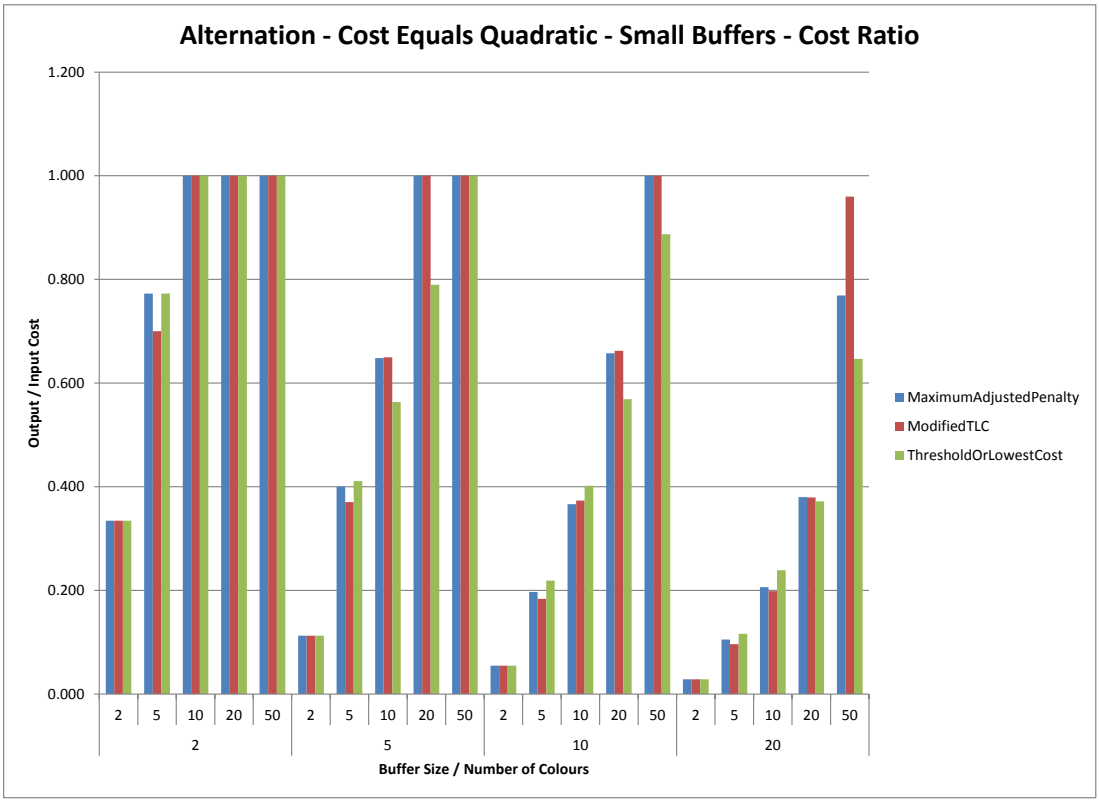

Figure 7: TLC achieves a good performance when $C = 2 * k$

and TLC' when the number of colours is about two times the size of the buffer. MAP and TLC' have comparable performance in this case. This trend remains consistent when comparing the cost ratios with TLC achieving a better cost ratio when the number of colours is two times the size of the buffer, TLC' and MAP fail to achieve good cost ratios for small buffer sizes with TLC' having the lowest performance.

When comparing the switches ratio for the Colour Difference cost function, TLC' achieves a better performance than MAP and TLC when the number of colours is less than or equal to the buffer size. It is interesting to note that MAP and TLC have exactly the same switch ratios for this cost function, these are slightly worse than

that of TLC'. This is illustrated in Fig. 8. Staying consistent with the trend, TLC' performs poorly when the number of colours is less than or equal to the buffer size for the Colour Difference cost model. However, MAP and TLC still have exactly the same cost ratios for this case as well, making them the choice of algorithm for the Colour Difference Cost Function.

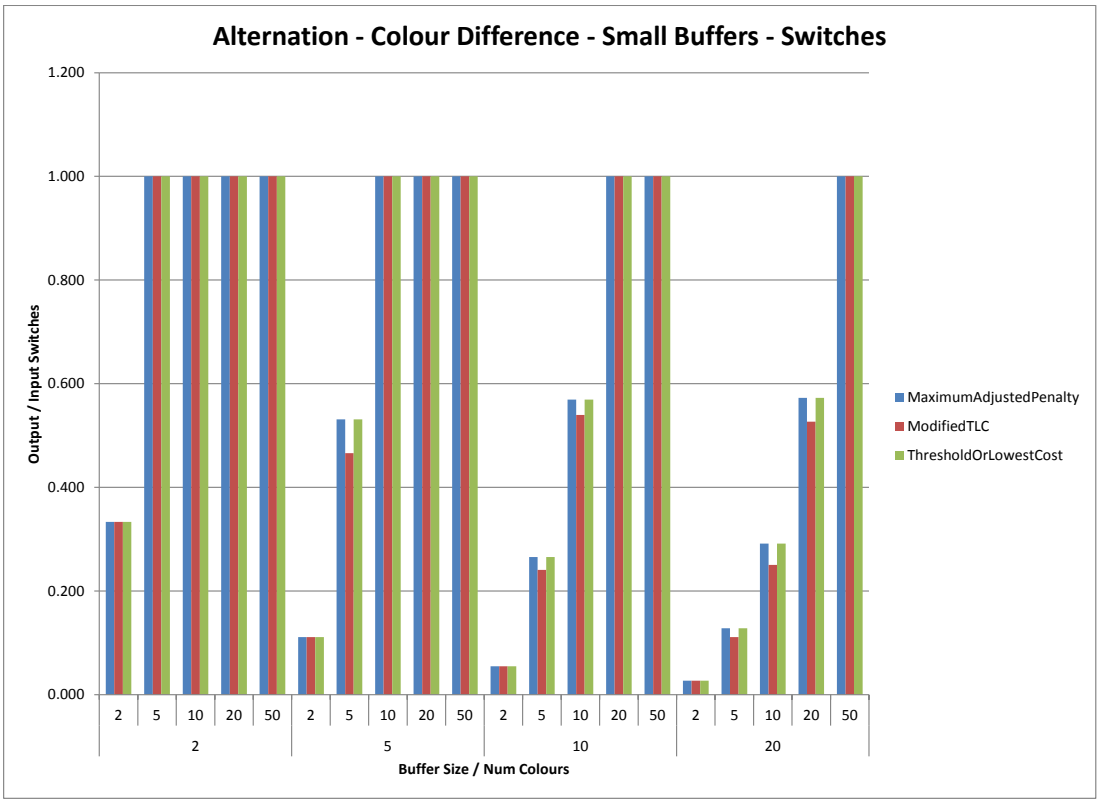

Figure 8: TLC' achieves a good performance when $C <= k$

For medium sized buffers (50, 100 and 500), we observe that TLC' achieves a better switch ratio than TLC and MAP over all combinations of buffer sizes and number of colours. However, when comparing cost ratios, we observe that MAP and TLC' have almost identical cost ratios, where as TLC has a slightly higher cost ratio

than the other two.

We observe a similar trend with the Cost Equals Colour Quadratic cost function, where TLC' has a significantly lower switch ratio than MAP and TLC, thereby making it the choice of algorithm for this application scenario. This is illustrated in Fig. 9. When comparing the cost ratios, we observe that TLC' has a higher cost ratio than MAP and TLC only for the case where the buffer size and number of colours are the same (50 and 50), for all other cases, TLC' and MAP have comparable cost ratios with TLC doing slightly worse than the other two in many cases.

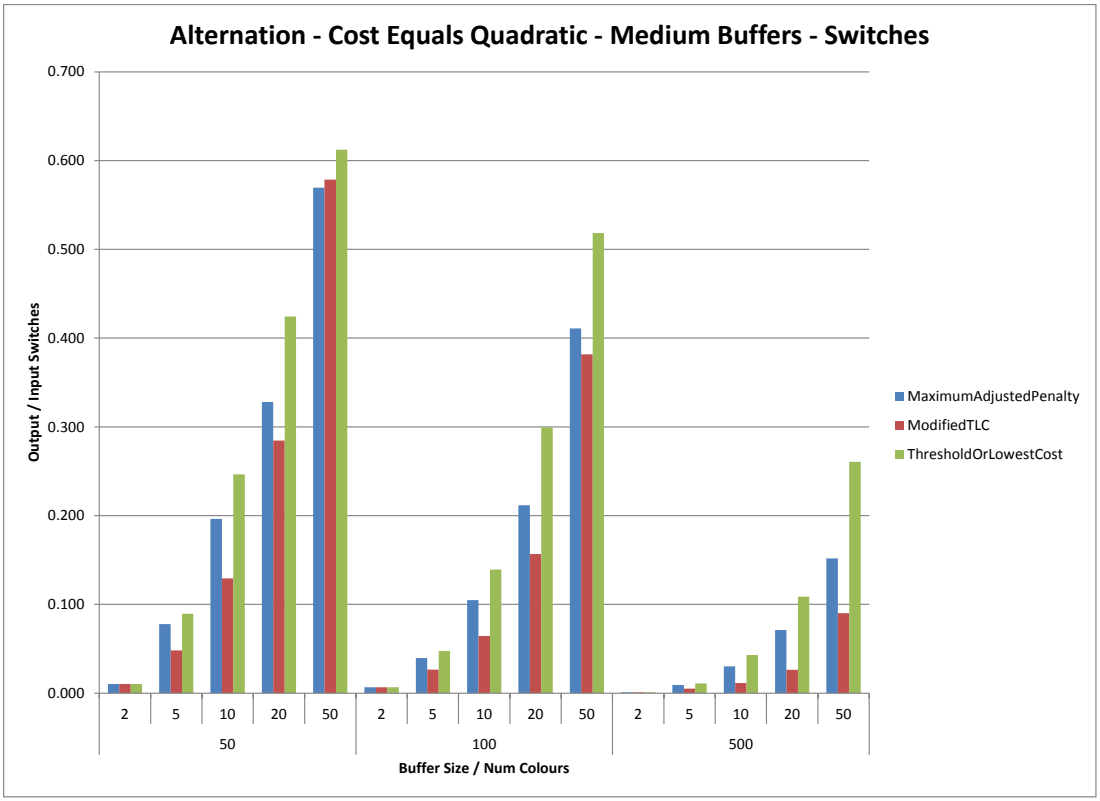

Figure 9: TLC' has significantly lower switches

Results observed for the Random cost function are consistent with the Cost Equals

Quadratic Colour and Cost Equals Colour cost functions where TLC' has a significantly better switch ratio, we also observe the exact case where TLC' has a worse performance than TLC and MAP when the buffer size and number of colours are the same (50 ad 50). As with the other cost functions, the cost ratios for the other cases remain comparable between MAP and TLC' with TLC having a marginally worse performance.

As observed in the case of small buffers, TLC' has a better switch ratio than TLC and MAP for the Colour Difference cost function. However, the the cost ratio for TLC' is significantly higher than that for TLC and MAP as illustrated in Fig. 10. It is also interesting to note that TLC and MAP have exactly the same ratios when comparing both switch and cost ratios, as we see in Fig. 10.

For the Uniform cost function, we observe that all out algorithm have almost identical switch and cost ratios, with only very marginal differences.

Delta Sequences

When comparing the cost ratios across our algorithms for the Cost Equals Colour cost function, we observe that TLC' achieves a slightly better performance than TLC and MAP for small buffers. This is illustrated in Fig. TLC' also achieves a better switch ratio than MAP and TLC with MAP performing slightly better than TLC for small buffers.

In the case of Cost Equals Colour Quadratic cost function, TLC' achieves a significantly better switch ratio than TLC and MAP across all small buffer sizes. This is illustrated in Fig. MAP performs better than TLC with TLC having the worst switch ratio for this cost model. When comparing the cost ratios, all our algorithms achieve comparable performance with TLC' and MAP having better performance than TLC; TLC' has a marginally better cost ratio than MAP in many cases.

All our algorithms achieve almost exactly the same cost ratios for the Random

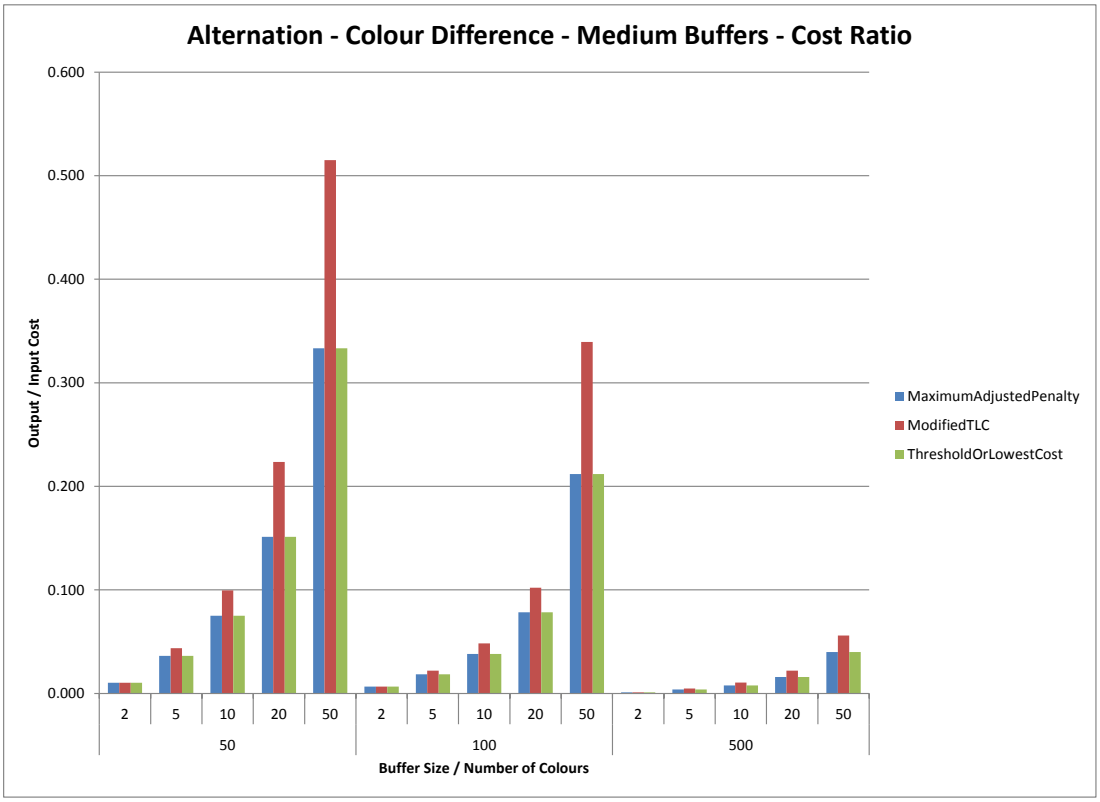

Figure 10: TLC' has significantly higher cost ratio

cost function, with TLC doing very marginally worse than the other two. However, when comparing the switch ratios for these algorithms, we observe that TLC' achieves a better switch ratio than MAP and TLC for most small buffer sizes, with buffer size 2 being an exception where all our algorithms achieve the same performance. We also observe that MAP performs better than TLC when switches are compared.

As in the case of Alternation sequences, although TLC' has a better switch ratio than MAP and TLC we observe that like with Alternation Sequences, the cost ratio is significantly higher for TLC' when compared to TLC and MAP. TLC achieves the best performance in terms of cost ratios while MAP has a comparable performance.

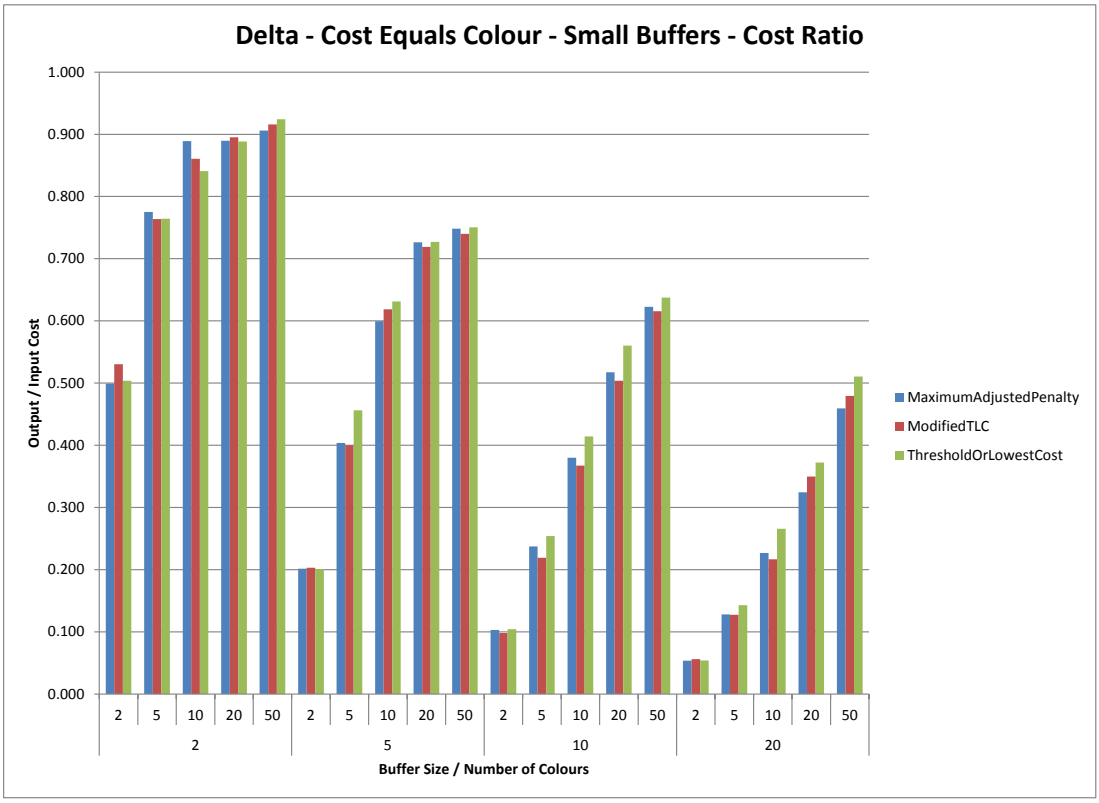

Figure 11: TLC' achieves a better performance than MAP and TLC

TLC has a significantly better switch ratio than MAP or TLC for medium buffers, while the cost ratio is not significantly better, TLC still has better cost ratios than TLC and MAP for all buffer sizes, except the case when $k = 50$ and $C = 50$. In this case, MAP does slightly better than TLC'.

In the case of the Cost Equals Quadratic Colour cost function, TLC' has a better switch ratio compared to TLC and MAP across all medium sized buffers. But when we compare the cost ratios, TLC' has a slightly better cost ratio for most cases except when the buffer size is 50 and number of colour is 50, in this case MAP does slightly better than TLC and TLC'. Another exception is when the buffer size is 100 and

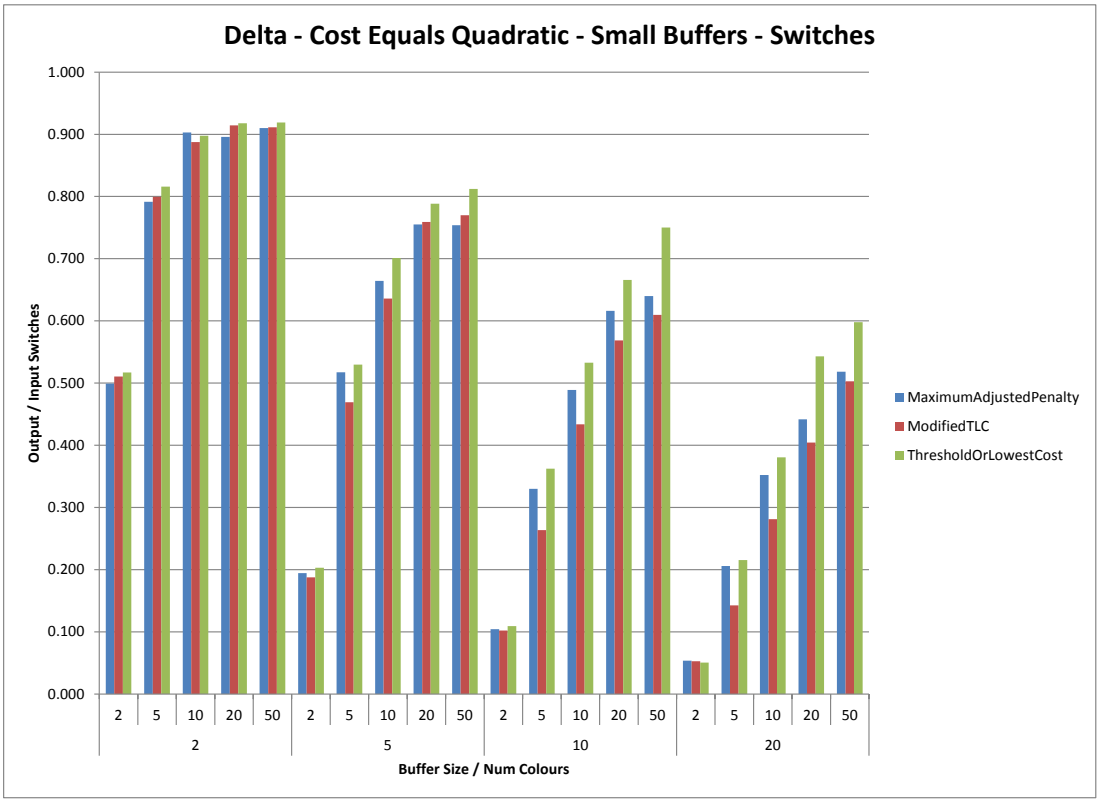

Figure 12: TLC' achieves a better switch ratio than MAP and TLC

number of colours is 50, TLC does slightly better than MAP and TLC'.

For Random costs, we see that TLC' does better than TLC and MAP when we compare both switch and cost ratios. While the performance is significantly better when comparing switches, it is slightly better when comparing costs as illustrated in the Fig. 13.

We observe that TLC' has lower switch ratios than TLC and MAP for the Colour Difference cost function. While the switch ratios are lower, we also observe that the cost ratios are significantly higher than what we see for TLC and MAP, as illustrated in Fig. 14. It is also to be noted that while the cost ratio is significantly higher

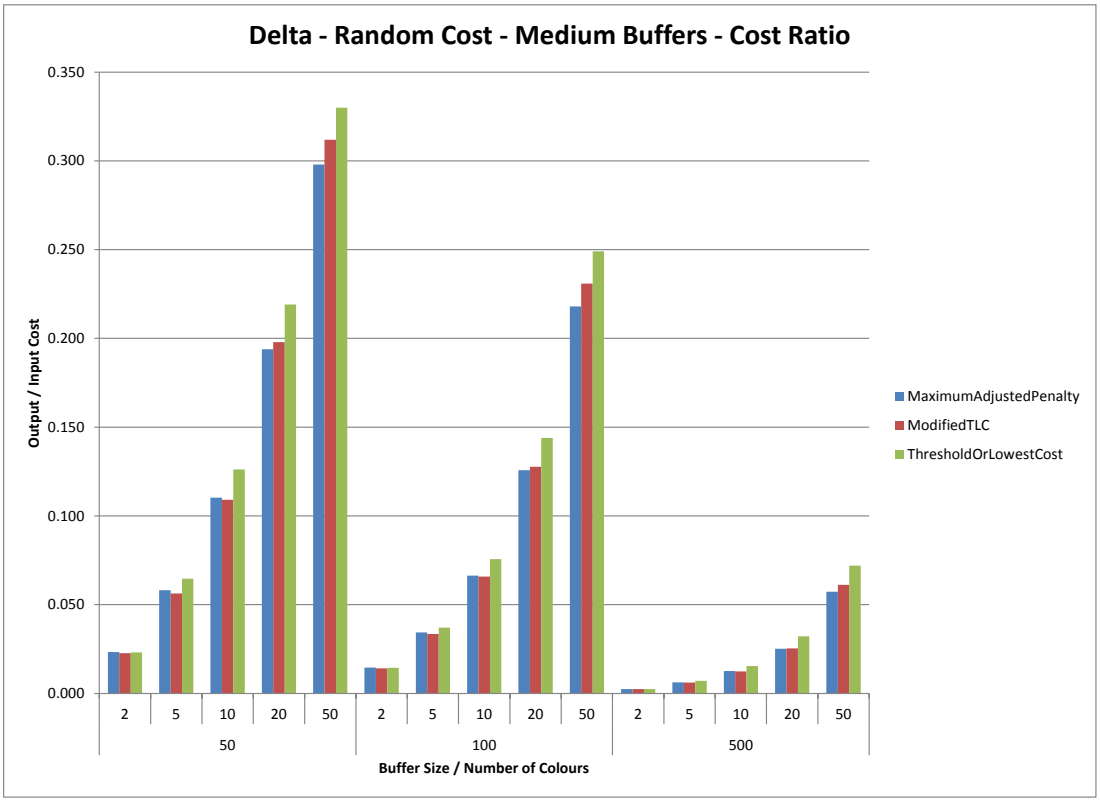

Figure 13: TLC' has a lower cost ratio than MAP and TLC

it is only when compared to TLC and MAP, overall all our algorithms achieve very good cost ratios as well below 45% indicating that more than half the cost has been eliminated. This can be inferred from Fig. 14.

All our algorithms achieve a good performance for the Uniform cost function having almost identical cost ratios. An exception is the case when the buffer size is 50 and number of colours is 50, in this case MAP does slightly worse than the other two algorithms.

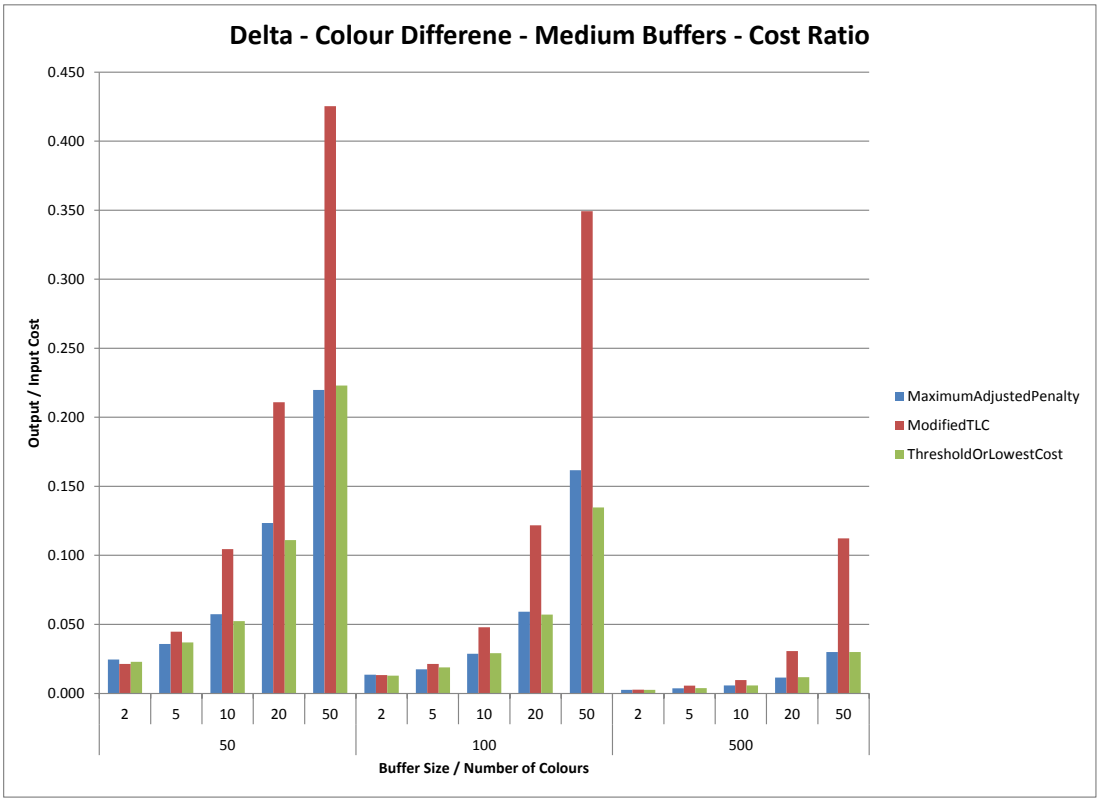

Figure 14: TLC' performs poorly when comparing cost ratios with MAP and TLC

Random Sequences

For the Cost Equals Colour cost function, we observer that TLC' performs only marginally worse than TLC and MAP when comparing cost functions when the number of colours is greater than the buffer size. For other cases, it has a better cost ratio than TLC. Overall, We also observe that the cost ratios all our algorithms have comparable cost ratios with only very small variations. However, when comparing switch ratios, we observe that TLC' achieves a better switch ratio than MAP and TLC across all buffer size and number of colour combinations.

We observe a similar trend with the Cost Equals Quadratic Colour cost function

where TLC' achieves a better switch ratio than TLC and MAP across all combinations, but has a higher cost ratio compared to the other two. When comparing costs, we observe that TLC' has a higher cost ratio than TLC and MAP for all combinations where the number of colours is greater than the buffer size. This is the same as the observation made for the Cost Equals Colour cost function.

TLC' has consistently similar results even for the Random cost function as well, where it achieves a better switch ratio than TLC and MAP across all buffer sizes. This is illustrated in Fig. 15Our results are also consistent when comparing the cost ratios of these three algorithms, as we observe that TLC' has marginally higher cost ratios when the number of colours is greater than the buffer size. TLC and MAP have comparable performance with TLC doing marginally better when the number of colours is two times the buffer size.

While our results follow the similar trend of TLC' having a better switch ratio than TLC and MAP, and having a worse cost ratio than the others, for the case of Colour Difference cost function we observe that the cost ratios for TLC' are significantly higher than the cost ratios for MAP and TLC. This result is illustrated in Fig. TLC and MAP still continue to retain comparable cost ratios with only marginal differences.

For the Uniform cost function, we observe that all our algorithms perform almost identically achieving almost the same reordering ratios for both switches and costs. We also observe that our algorithms do no achieve very significant reordering for large number of colours, as the reordering ratios are closer to 1.0.

All our algorithms achieve a very good performance when comparing the reordering ratios for medium sized buffers. We observe that for the Cost Equals Colour and Cost Equals Quadratic Colour cost functions, TLC' has significantly lower switch ratios than TLC and MAP. When comparing cost ratios, we observe that TLC' does slightly better than TLC and MAP for both these cost functions. These results are

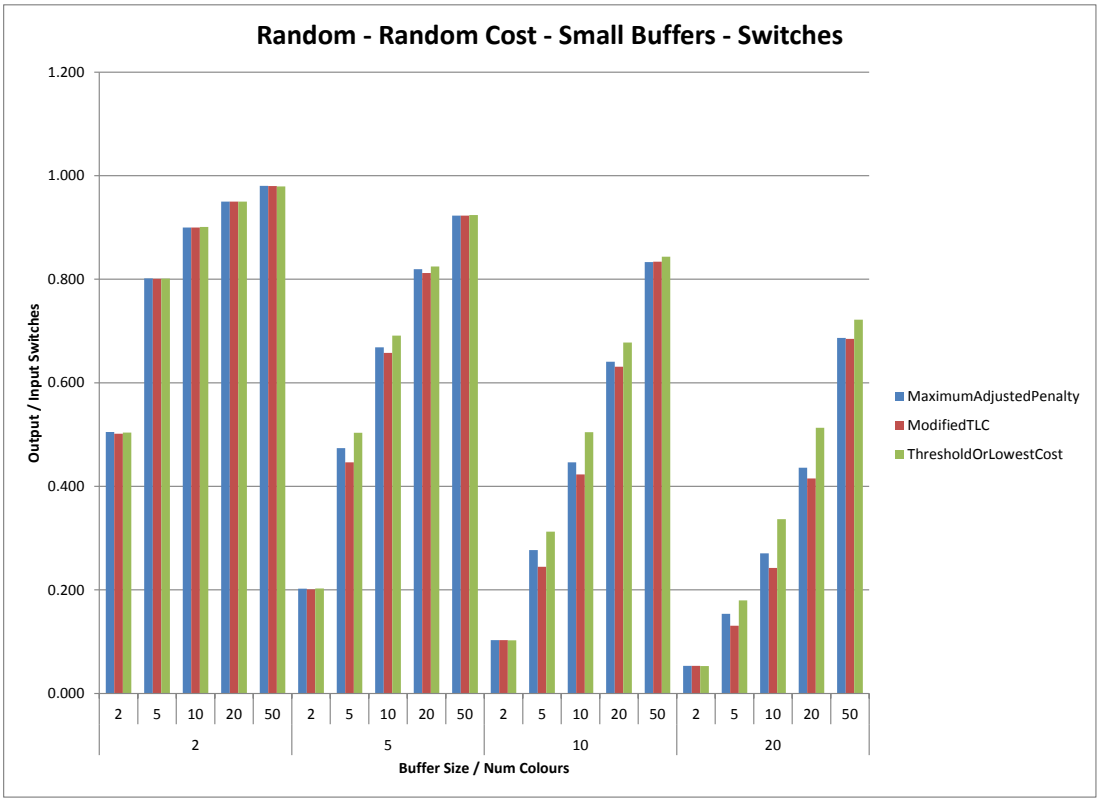

Figure 15: TLC' achieves a better switch ratio than MAP and TLC

illustrated in Fig. 17 and Fig. 18.

This trend where TLC' has a significantly lower switch ratio than MAP and TLC and where TLC' performs slightly better than TLC and MAP when cost ratios are compared extends to the Random cost function as well. Similar figures can be plotted to reveal this behaviour.

While all our algorithms perform extremely well for the Colour Difference cost function, we observe that TLC' has a significantly higher cost ratio when compared to TLC and MAP. However, we also observe that TLC' has fewer switches for exactly the same buffer size/number of colours combinations. MAP performs slightly worse

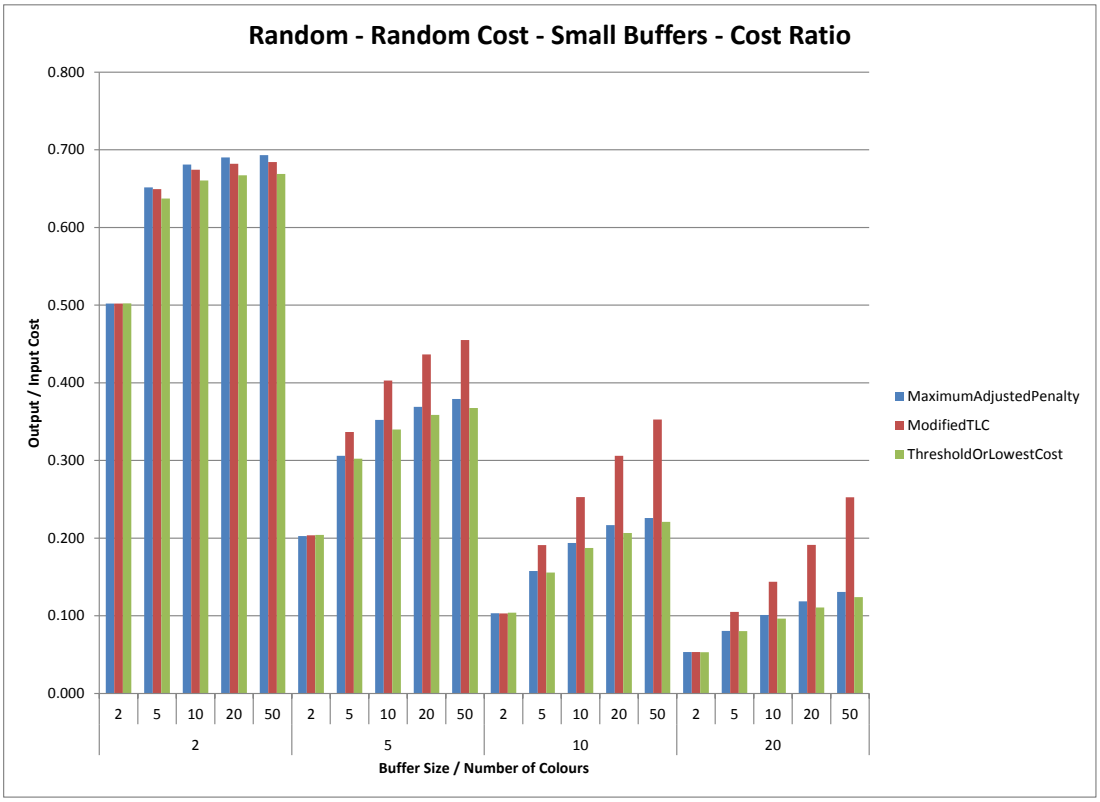

Figure 16: TLC' has significantly higher cost ratios than TLC and MAP

than TLC making TLC the best algorithm for the Colour Difference cost function.

For the Uniform cost function, we observe that all our algorithms have very comparable reordering ratios, with MAP performing slightly better than TLC and TLC' in some cases.

Random Block Sequences

All our algorithms achieve very comparable performance when comparing the cost ratio for the Cost Equals Colour cost function. Since the input sequence comes in variable sized blocks, all our algorithms can achieve a good reordering ratio only when

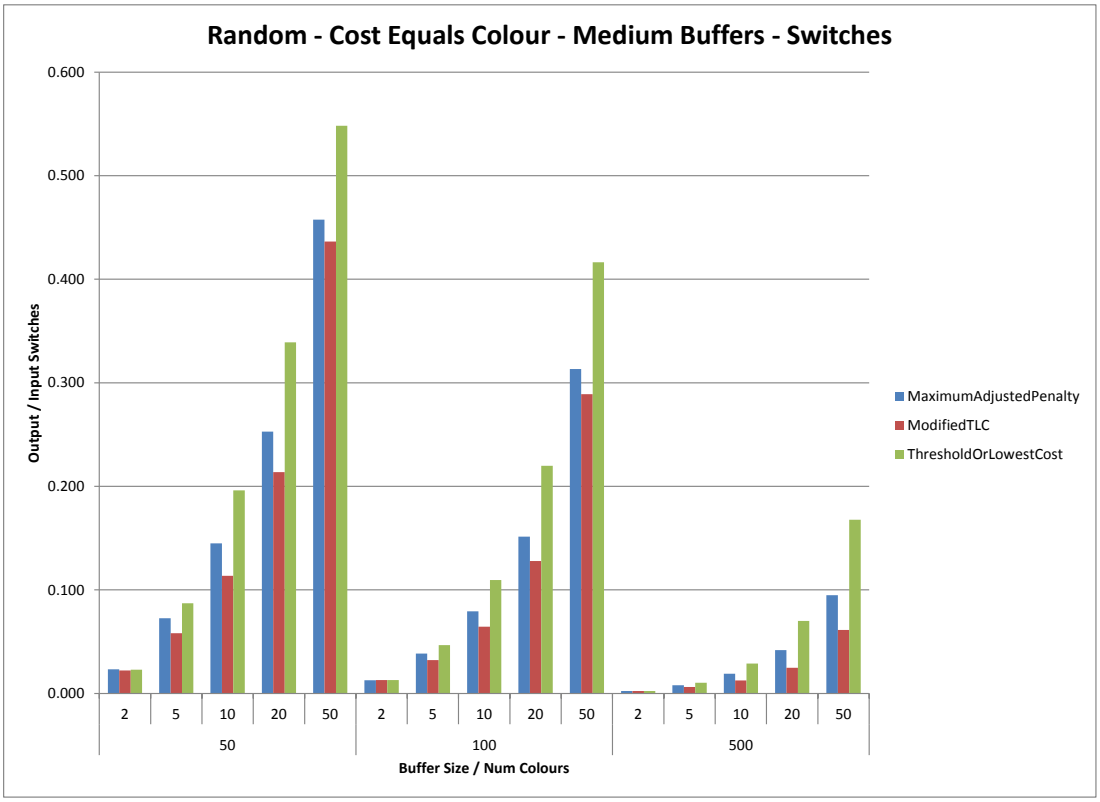

Figure 17: TLC' has significantly lower switch ratios than TLC and MAP

the size of the blocks are small. This can happen only when the number of colours are small since the block size increases as the number of colours increases. This is reflected in the results obtained where we see that the cost and switch ratios are closer to 1.0 for large number of colours and our algorithms achieve a good performance otherwise. When comparing the switch ratios, we observe that TLC' achieves a better performance than MAP and TLC for small buffer sizes with few colours, MAP and TLC have comparable performance with MAP doing marginally better in most cases. This is illustrated in Fig. 19.

We observe a similar trend as with the Cost Equals Quadratic Colour cost function

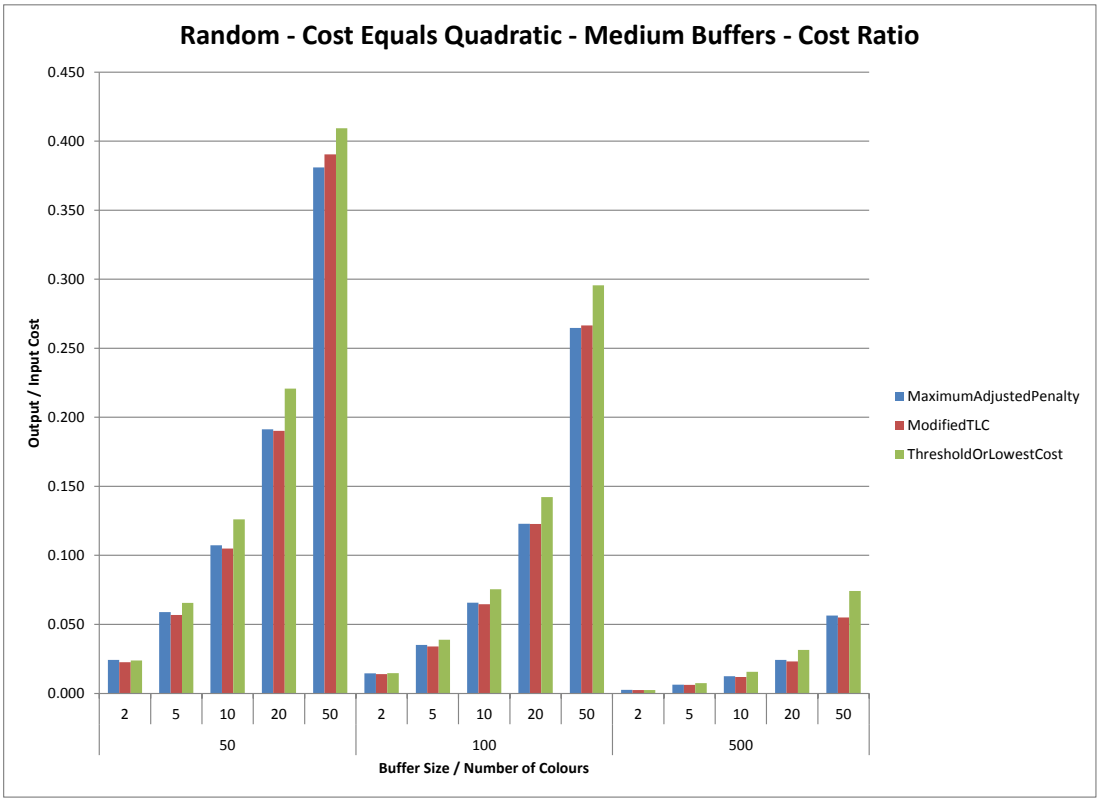

Figure 18: TLC' performs better than TLC and MAP when cost ratios are compared

where TLC' achieves a better switch ratio than TLC and MAP. When comparing cost ratios we observe that all our algorithms achieve very comparable ratios with TLC doing marginally better than MAP and TLC' in most cases.

Staying consistent with the trend, Random cost function also exhibits a similar behaviour where all our algorithms have extremely comparable performance, performing only marginally better or worse than the other when comparing cost ratios. This is illustrated in Fig. 20. When comparing switch ratios, we observe that TLC' achieves a better switch ratio than MAP and TLC when the number of colours is less than the buffer size, MAP and TLC have very similar ratios with MAP performing

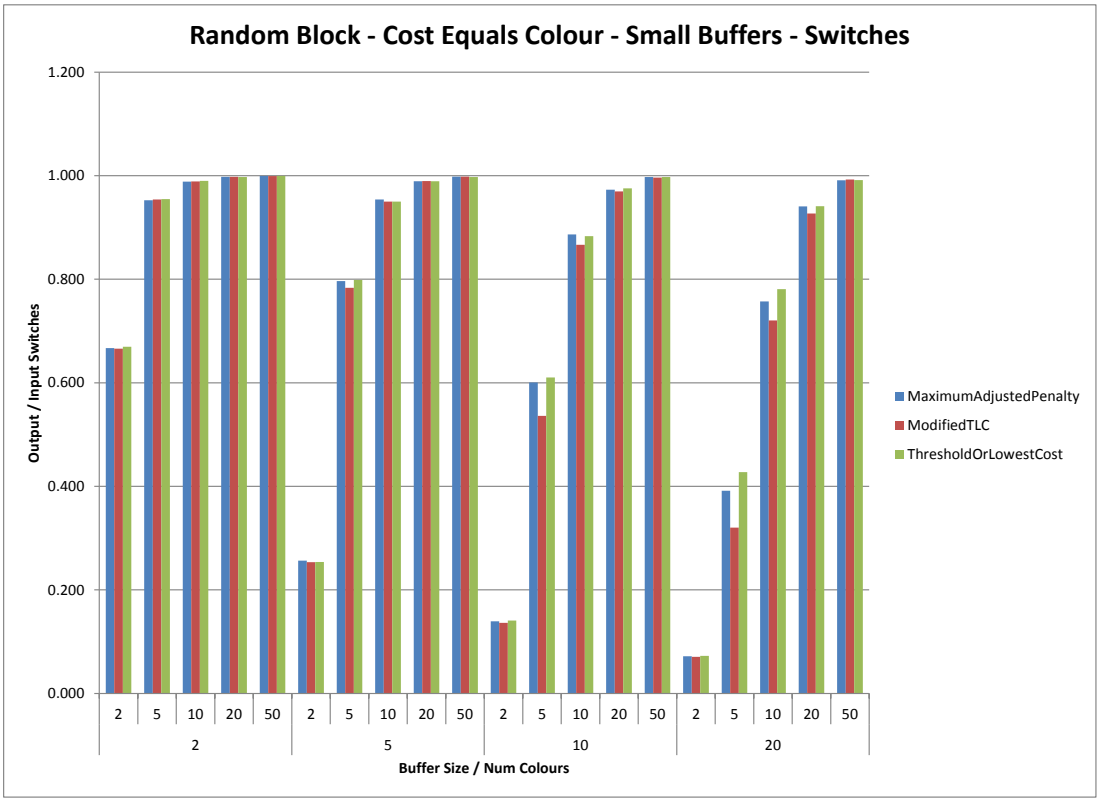

Figure 19: TLC' achieves a better switch ratio than MAP and TLC for small number of colours

marginally better.

All our algorithms have very comparable switch ratios for the Colour Difference cost function. As seen in the other cases, TLC' does slightly better when the number of colours is very small, but otherwise all our algorithms have very similar switch ratios. However, while comparing cost ratios we observe that TLC' performs poorly for this cost function, it has significantly higher cost ratios compared to TLC and MAP, which both have comparable cost ratios.

In the case of Uniform cost function, we observe that all our algorithms have

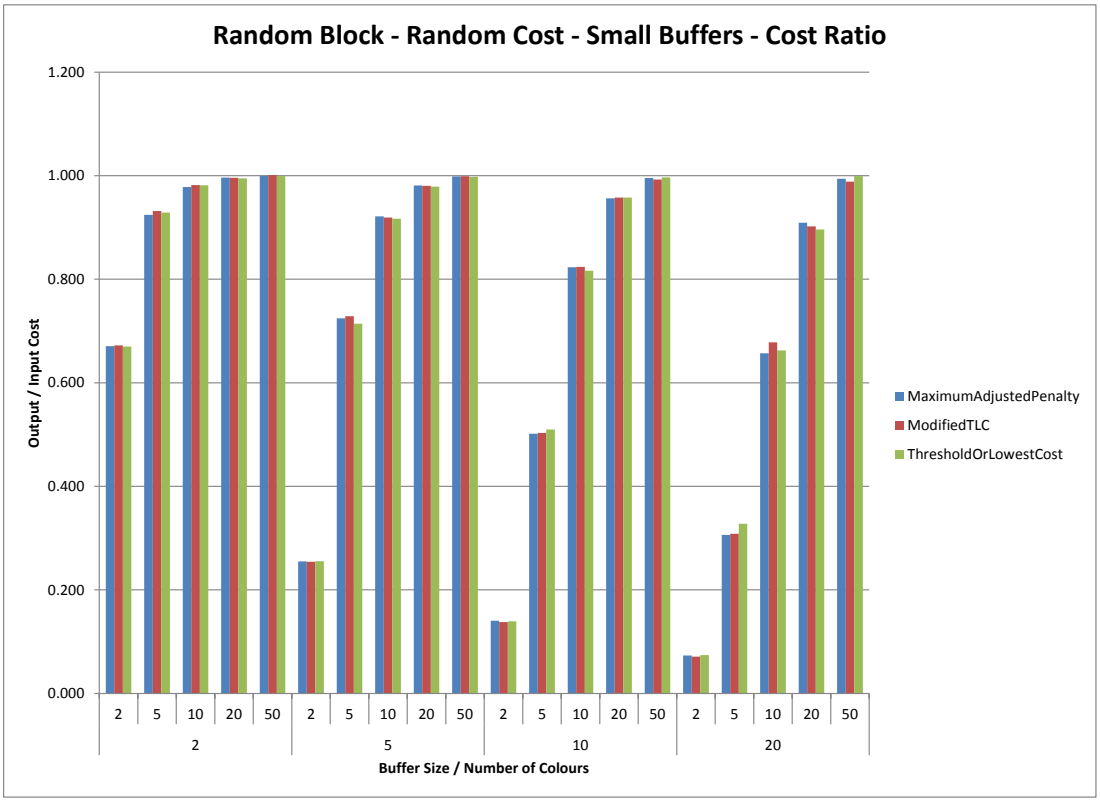

Figure 20: TLC' achieves a better switch ratio than MAP and TLC for small number of colours

almost similar cost and switch ratios, with TLC' and MAP being almost identical in a few cases, TLC' performs marginally better than MAP and TLC in other cases and TLC has the marginally higher cost ratios for this cost function. However, overall, we observe that all our algorithms have comparable performance.

As is the case with small buffers, we observe that the reordering ratios for all our algorithms are higher for large number of colours even in the case of medium buffers. This is expected behaviour since the block size increases as the number of colours increase leaving little scope for reordering. We observe that for the Cost Equals

Colour cost function, TLC' achieves a better performance than TLC and MAP when both the cost and switch ratios are compared.

We see a similar trend with the Cost Equals Quadratic Colour cost function, however, the similarity is only limited to the switch ratio comparison. As with the Cost Equals Colour cost function, we observe that TLC' has a better switch ratio than TLC and MAP, but when comparing the cost ratios we observe that there is no one algorithm that performs better than the other across all combinations. This is illustrated in Fig. 21. Also, for buffer sizes 50 and 100, when the number of colours are 50, we observe that the cost ratio for MAP and TLC exceeds 1.0 indicating that reordering increases the cost. These details are explained in a later section.

For the Random cost function, we find that TLC' has a better performance than TLC and MAP when the number of colours is small compared to the buffer size ($C < k$), when the switch ratios are compared. For all other cases all our algorithms have similar performance. When cost ratios are compared, we observe that TLC' has higher cost ratios than TLC and MAP when the number of colours increases.

When comparing the switch ratio for the Colour Difference cost function, we observe that all our algorithms achieve very similar ratios indicating that they all have similar reordering performance. However, when comparing the cost ratios, we observe that TLC' has significantly higher cost ratios than TLC and MAP, with the cost being more than two times the cost incurred for TLC and MAP. This result is illustrated in Fig. 22.

For Uniform cost function, all our algorithms have almost identical performance.

Sequential Block Sequences

For the Cost Equals Colour cost function, out algorithms fail to permute the sequence for very small buffer sizes (2 and 5). All our algorithms have a switch and cost ratio of 1.0 indicating that the output sequence is identical to the input sequence.

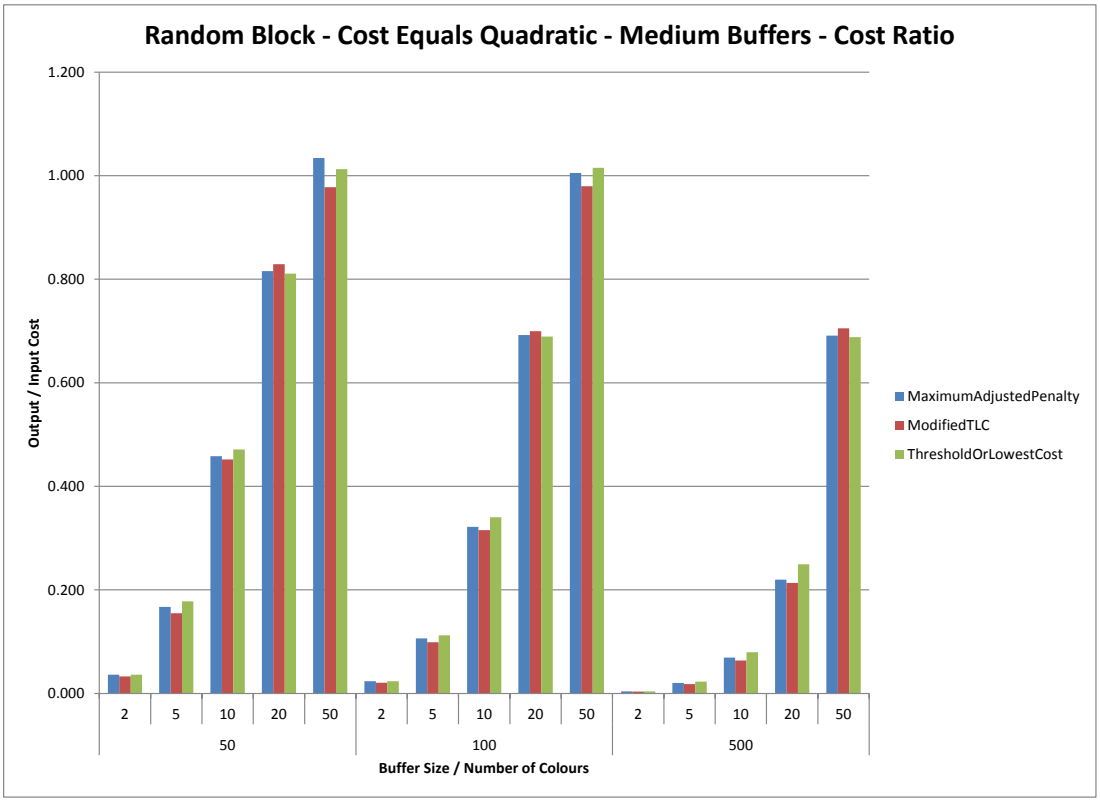

Figure 21: All algorithms are comparable

This behaviour is expected since our block sizes are 5 making it impossible for the algorithms to achieve any reordering. For buffer sizes 10 and 20, we observe that TLC achieves a better switch and cost ratio than MAP and TLC' when the number of colours is half the buffer size (that is for number of colours 5 and 10), while TLC' achieves a better performance than MAP and TLC for buffer size 20 and 5 colours. In all other cases we do not see any reordering as the number of colours and block sized nature of the input sequence does not permit any reordering.

As seen in the case of Cost Equals Colour cost model, the Cost Equals Colour Quadratic Coloir cost function also witnesses no reordering for very small buffer sizes

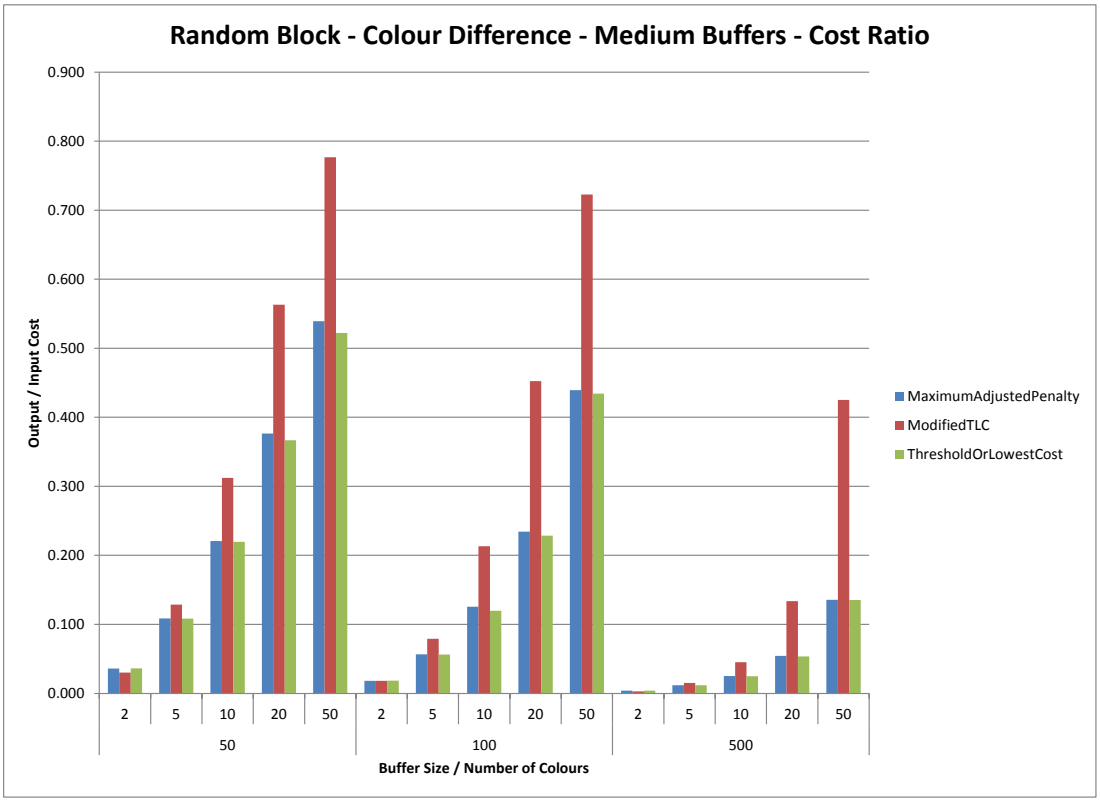

Figure 22: The cost ratio for TLC' is more than two times the cost ratio for TLC and MAP

(2 and 5). However, for buffer sizes 10 and 20, our algorithms achieve reordering ratios close to 1.0 indicating that there has been no significant reordering. For buffer size 20, TLC achieves the best switch and cost ratios as long as the number of colours are less than or equal to the buffer size. It is interesting to note that while TLC and MAP achieve some reordering for buffer sizes 10 and 20 when the number of colours is 5 and 10, TLC' fails to reorder the input sequence thereby having switch and cost ratios as 1.0. This is illustrated in Fig. 23.

Random cost function follows a similar trend where TLC' fails to achieve any

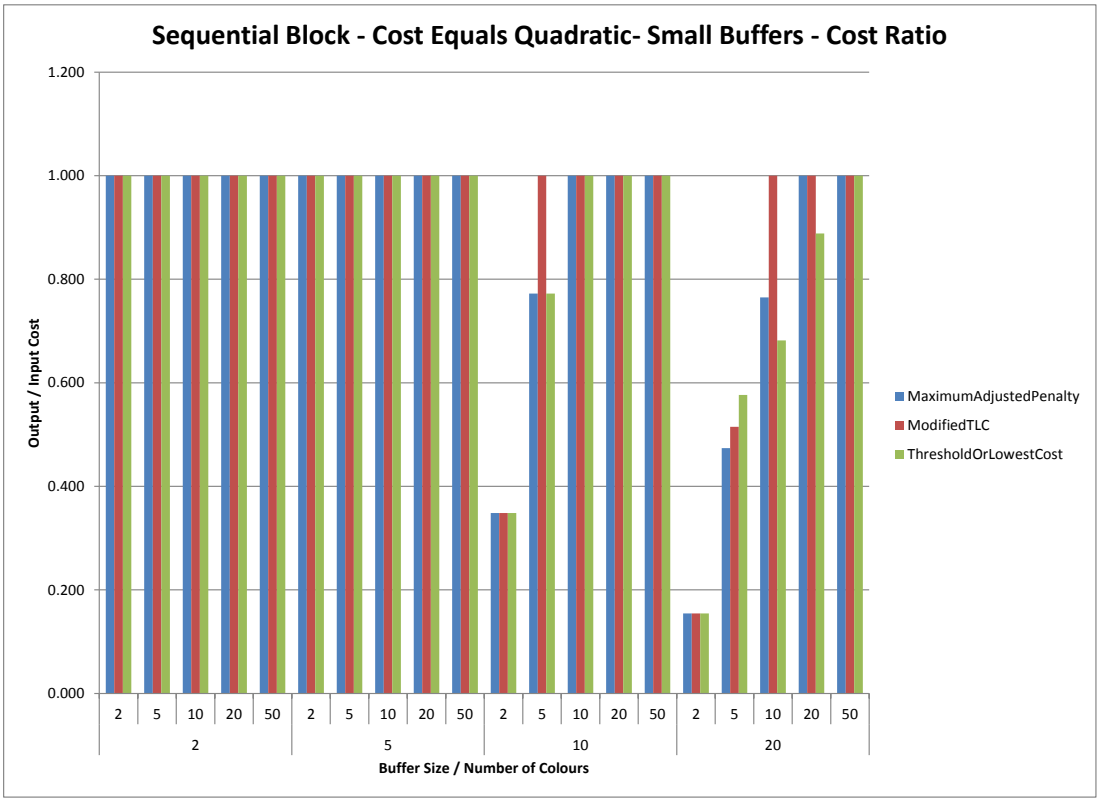

Figure 23: TLC' fails to permute for $k = 10, 20 and C = 5, 10$

reordering for exactly the same combination of buffer sizes and number of colours ($k = 10, 20$ and $C = 5, 10$). TLC and MAP achieve some reordering in this case with TLC performing better than MAP. However, for the Random cost function we also observe that the cost ratio for all our algorithms exceed 1.0 very marginally (of the order of 1.005) indicating that the cost of the output sequence is higher than the cost of the input sequence, this is analyzed in a later section.

For the Colour Difference cost function, we observe that all our algorithms achieve exactly the same cost ratios for all combinations of small buffer sizes and number of colours. This trend continues when we compare the switch ratios as well except for

the case when the buffer size is 20 and the number of colours is 5, in this scenario, we observe that TLC' achieves a better switch ratio than TLC and MAP, followed by MAP and then TLC. This is the only case where we observe different reordering ratios, all our algorithms have identical performance otherwise. This is illustrated in Fig. 24.

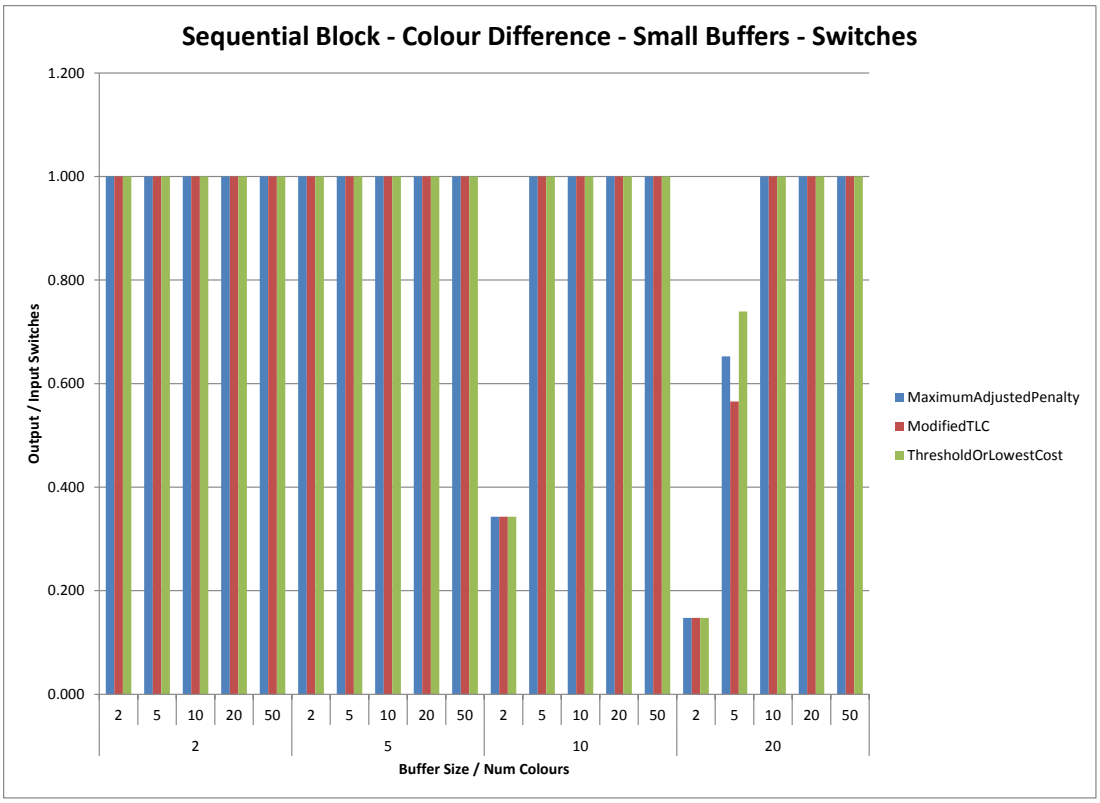

Figure 24: Non-identical switch ratio for $k = 20$ and $C = 5$

All our algorithms achieve exactly the same reordering ratios for the Uniform cost function, except for the case when the buffer size ($k = 20$ and $C = 5$), where MAP performs significantly better than TLC and TLC'. In this case TLC and TLC' fail to achieve any reordering of the input sequence. It may be noted that this is exactly the

same buffer size/number of colours combination that had a non-identical performance in the Colour Difference cost model.

For medium buffers, when comparing the switch ratios for the Cost Equals Colour cost function, TlC' has a better switch ratio than MAP and TLC when the number of colours is about half the size of the buffer. However, when comparing cost ratios we observe that TLC' has a better performance than MAP and TLC only when the number of colours is one-tenth of the size of the buffer. For all other cases, TLC and MAP have a better performance than TLC'.

The exact same trend continues for the Cost Equals Quadratic Colour cost function where TLC' has a better switch ratio than MAP and TLC when the buffer size is 5 times the number of colours; for cost ratios, the buffer size is 10 times the number of colours. This result is illustrated in Fig. for switch ratio comparison.

For the Random cost function, we observe the exact same trend as we have seen in the Cost Equals Colour and Cost Equals Quadratic Colour cost functions, except that when comparing cost ratios, we observe that for buffer sizes 50 and 100 and when the number of colours are 20 and 50, the cost ratios for MAP and TLC are marginally more than 1.0 indicating that the reordering has increased the cost. These details are analyzed in a later section.

For the Colour Difference cost funtion, we observe that TLC' has a better switch ratio than MAP and TLC, but the cost ratio is significantly higher for the cases where the switch ratio is lower. This is illustrated in Fig.

For the Uniform cost function, all our algorithms have almost identical switch and cost ratios indicating that they all achieve almost identical reordering.

5.4.3 Comparing TLC and TLC' for Large Data

In this section we present the experimental results for TLC and TLC' for large data sets. For these experiments, out input sequence size was 1000000, out buffer sizes

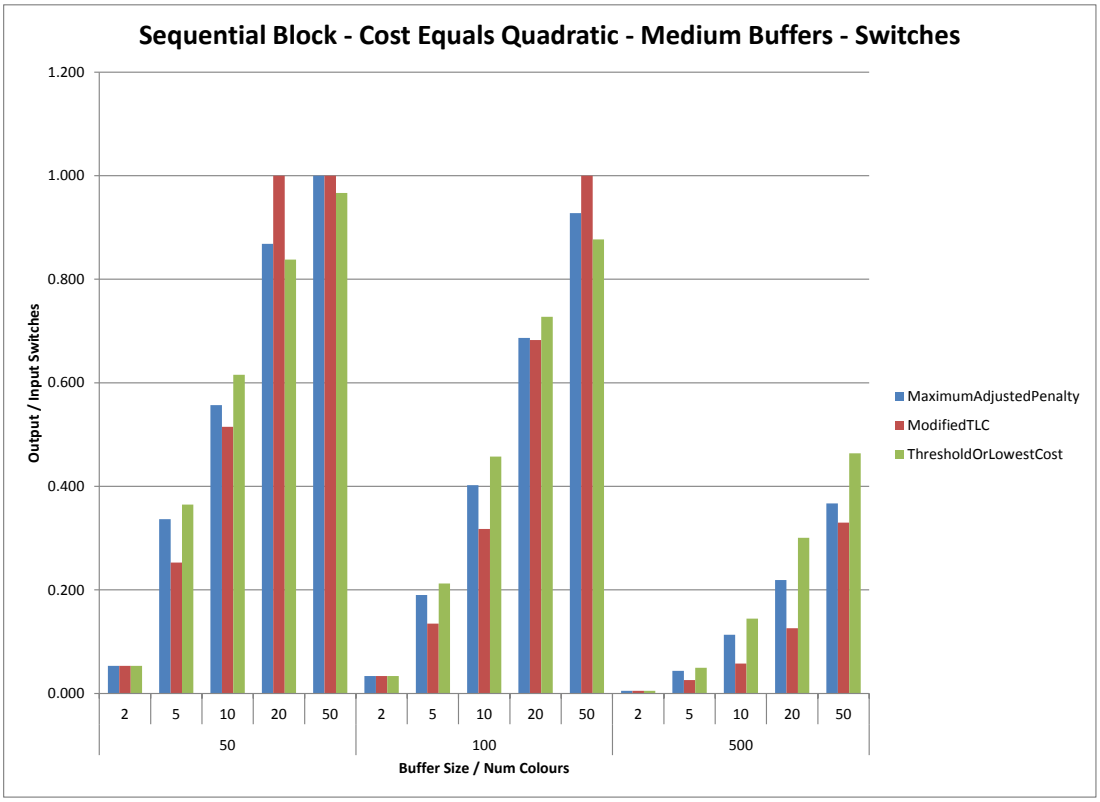

Figure 25: TLC' has better switch ratio when $k >= 5 \times C$

were 1000, 2000, 5000 and 10000 and the number of colour combinations were 100, 500 and 1000. Since our algorithm uses the same logic as TLC, we have used TLC as the base to compare all our experiments.

Owing to large size of the data sets, we have restricted our random sequences and random cost functions to 30 times and averaged the restults over these times, unlike the small data sets were we have 100 runs.

The following subsections present the results observed for each input sequence with different buffers and number of colours combinations.

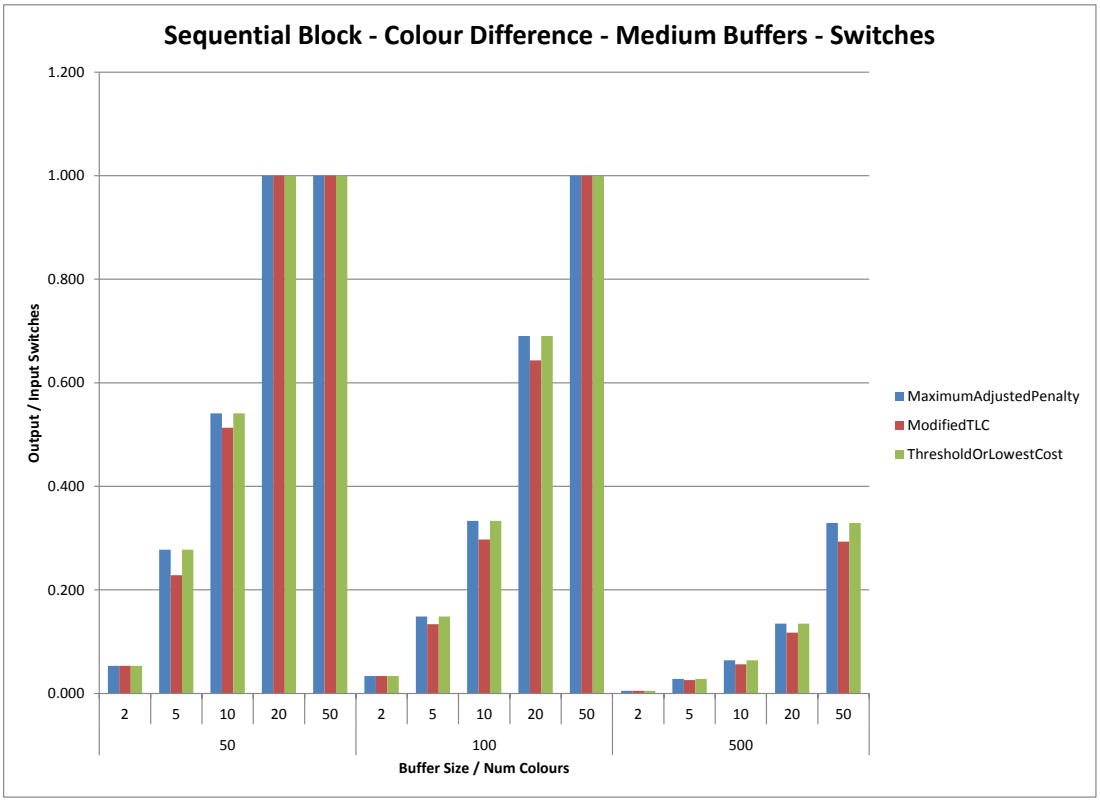

Figure 26: TLC' has better switch ratio

Alternation Sequences

For Alternation sequences with the Cost Equals Colour cost function, we observe that TLC' has a significantly better switch and cost ratio across all buffer size and number of colours combinations. This result is observed regardless of any inequality for the buffer sizes and number of colours. TLC' has the best reordering ratios when the number of colours are much smaller than the buffer size, but it still has a better performance than TLC when the number of colours are the same as the buffer size. This result is illustrated in Fig. 27.

When comparing switch ratios for the Cost Equals Quadratic Colour cost function,

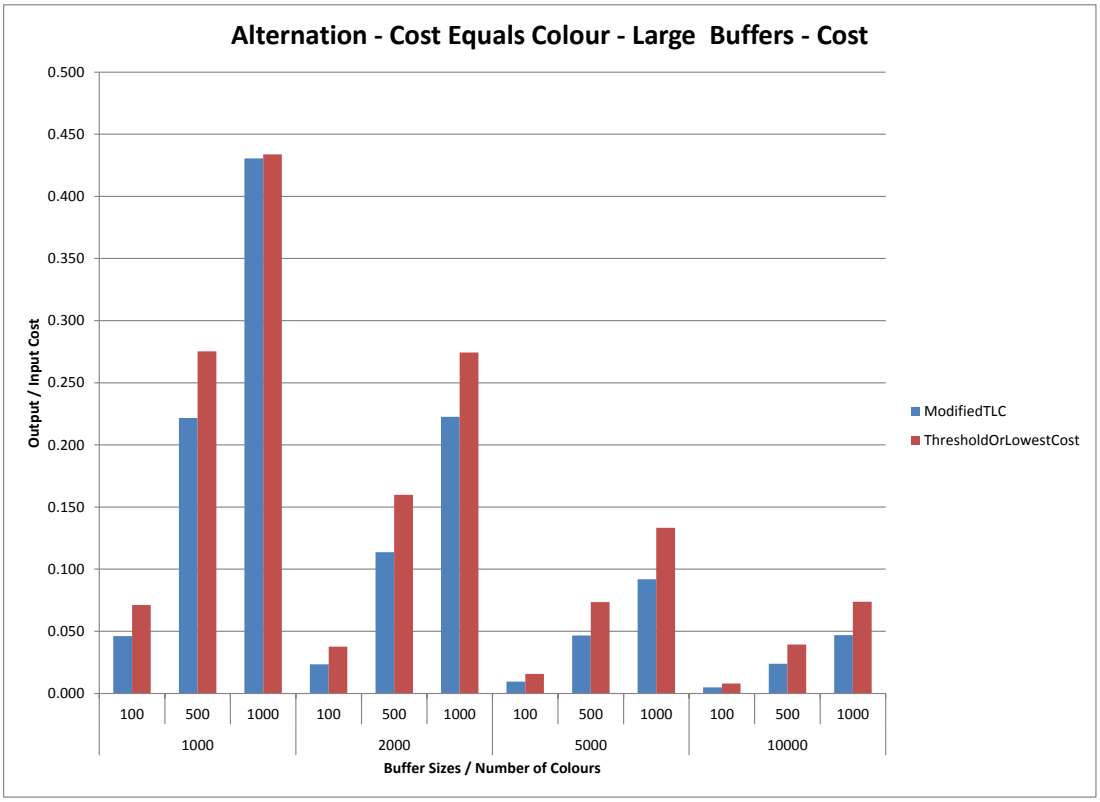

Figure 27: TLC' has better cost ratio

we observe that TLC' has a significantly better switch ratio across all buffer sizes and number of colours. Like with the Cost Equals Colour cost function, out best results are obtained when the number of colours is one-tenth of the buffer sizes. Overall, we still observe very good switch ratios for TLC' when compared to TLC. When it comes to comparing costs, we observe that TLC' has a better cost ratio when the buffer size is at least two times the number of colours ($k \geq 2 \times C$).

Our observations for Random cost are inline with the observations for Cost Equals Quadratic Colour cost function where we find that TLC' has a significantly better switch ratio across all buffer size and number of colour combinations and the cost

ratio for TLC' is better when the buffer size is at least two times the number of colours.

Staying consistent with the other cost functions, we observe that TLC' has a significantly better switch ratio for all buffer size and number of colours combinations for the Colour Difference cost function as well. However, as seen in the small data set results, TLC' fails to achieve a good performance when the cost ratios are compared for this cost function.

Both our algorithms have almost identical switch and cost ratios across all combinations of buffer sizes and number of colours.

Delta Sequences

Like with Alternation Sequences, we observe that TLC' has a better switch ratio than TLC for all buffer sizes for the Cost Equals Colour cost function. Likewise, this result also holds when comparing the cost ratios as we find that the cost ratios for TLC' are better than that for TLC across all buffers.

Similar observations hold good for the Cost Equals Quadratic Colour cost function where we observe that TLC' has significantly better switch ratios than TLC. This result is illustrated in Fig. 28. TLC' also has a better cost ratio than TLC for this cost function, but the difference is not as significant as that observed with comparing switch ratios.

As seen in the previous two cost functions, TLC' has a better switch and cost ratio than TLC even for the Random cost function. Also, like in the previous two cases, it is to be noted that although TLC' has a better cost ratio, our best performance is the switch ratio where TLC' has exceptionally low switches in the output sequence.

As seen with Alternation Sequences, TLC' has a better switch ratio than TLC for the Colour Difference cost function, however, the cost ratios are significantly higher than TLC. It is to be noted that both algorithms achieve a very good reordering,

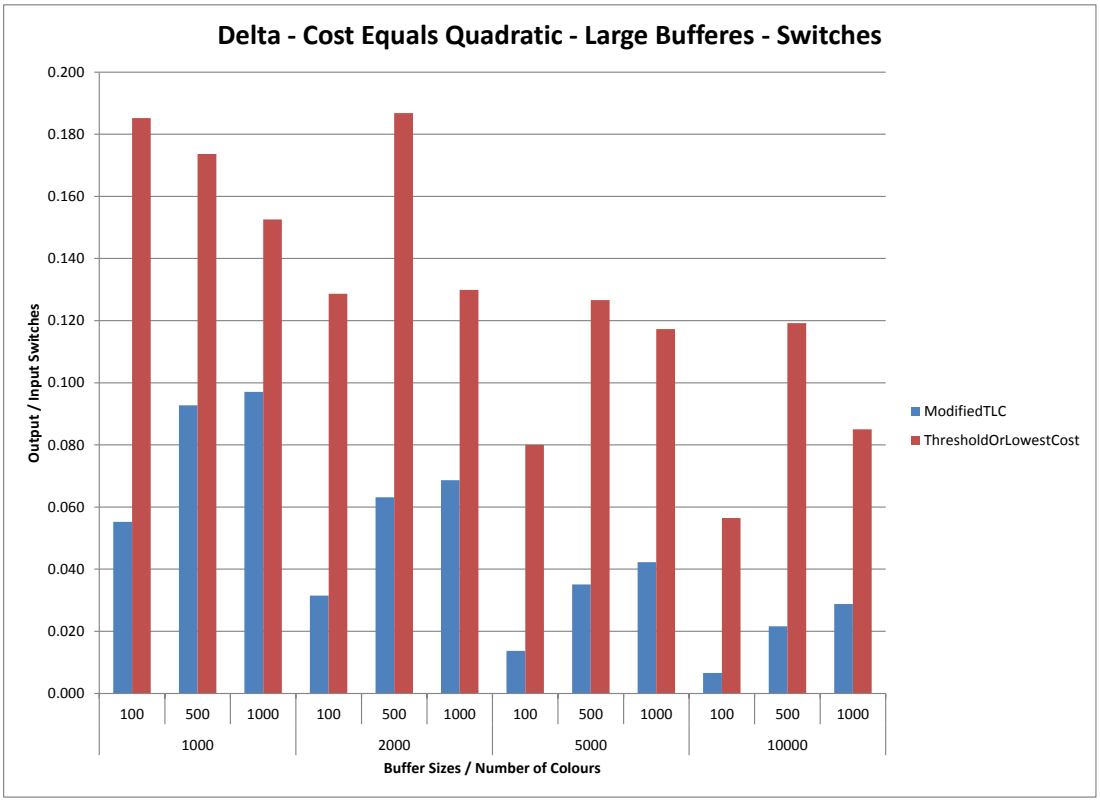

Figure 28: TLC' has a significantly better switch ratio

but TLC has an exceptionally low cost making it the choice of algorithm for this cost function.

For Delta sequences with a Uniform cost function, we observe that TLC' has a slightly worse performance than TLC, which is different from the results we have from the small data sets where both algorithms had almost identical performance for Uniform costs.

Random Sequences

For the Cost Equals Colour cost function, we observe that TLC' has a better switch and cost ratio than TLC for all buffer sizes and number of colours. Our best results are when the buffer size is much larger than the number of colours, in these cases, we achieve a switch ratios which are very small indicating that the output sequences have very few of the switches originally present in the input sequence.

We have the exact same observations when we compare the switch and cost ratios for the Cost Equals Quadratic Colour cost funtion. As in the previous case, we observe that TLC' has a significantly lower switch ratio across all combinations of buffer sizes tested. Likewise, TLC' has a lower cost ratio when the cost ratios of TLC and TLC' are compared regardless of the buffer sizes and number of colours.

For the Random cost, we observe that TLC' has a significantly better switch ratio regardless of the buffer sizes and number of colours. But when comparing cost ratios, we observe that TLC' has a better significant ratios only when the buffer size is strictly greater than the number of colours, that is $k >= 2 \times C$.

As seen with the other cost functions, we observe that TLC' has a better switch ratio for all combinations of buffers and colours. When comparing costs, both out algorithms do very well with the cost ratios being well below 4% in the worst case indicating that about 96% of the switches present in the input sequence were eliminated. However, TLC' has a poor performance compared to TLC as the cost ratios for TLC are all below 0.5% indicating that the output sequence basically has almost no switches. This is illustrated in Fig. 29.

For Uniform costs, we observe that TLC' has a marginally worse performance when the buffer size is small. Putting in other words, TLC' can achiever a better performance than TLC only when the buffer size is at least two times the number of colours.

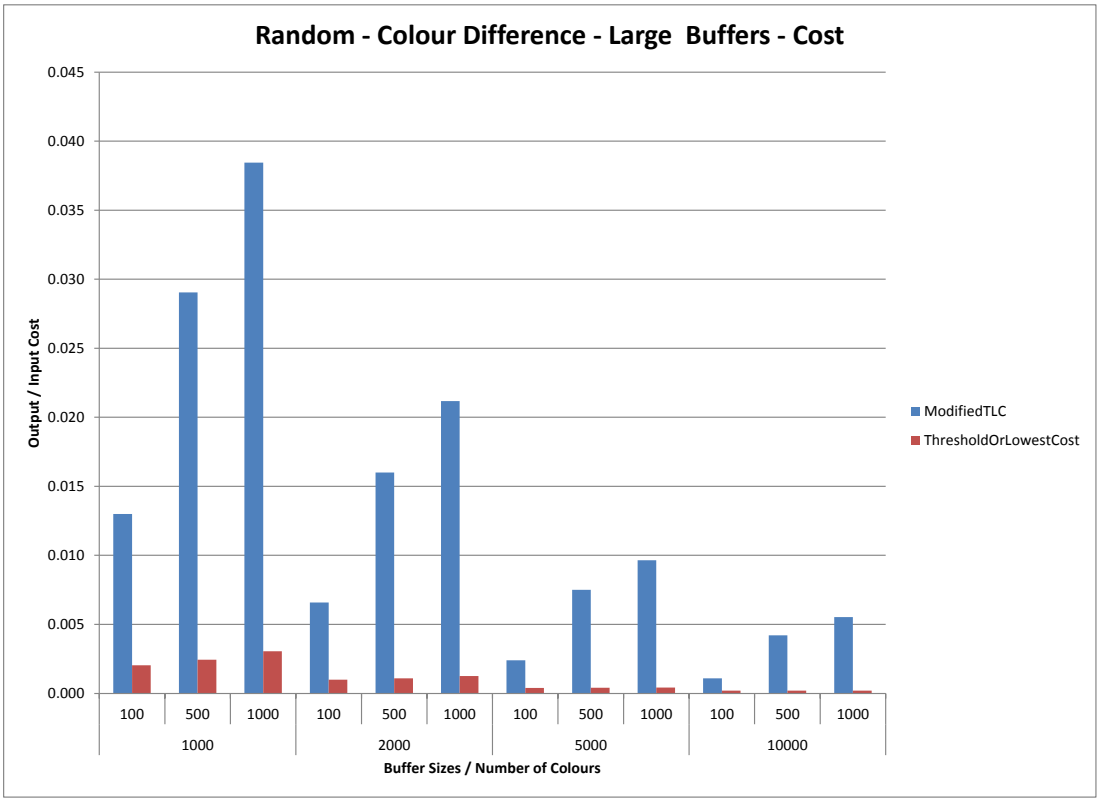

Figure 29: TLC performs exceptionally well

Random Block Sequences

When we compare the switch ratio for TLC and TLC' for the Cost Equals Colour cost function, we observe that TLC' has a significantly better switch ratio than TLC, when the number of colours is a fraction of the buffer size. For example, when the buffer size is 10 times the number of colours ($k = 10 \times C$), we observe that TLC' has 82% of the switches in the input sequence where as TLC has about 88% of switches in the input sequence. This is not a very significant difference, but when we compare the switch ratio for these two algorithms when the buffer size is a 100 times the number of colours ($k = 100 \times C$), we see a significant difference in the switch ratios. In this case,

TLC' has 23.5% of the switches present in the input sequence while TLC has 47.4% of the original switches. This is a significant result. However, when we compare the cost ratios for these two algorithms, we observe that TLC' only does slightly better than TLC in most cases, the best result being, for buffer size 10000 and 100 colours, TLC' has 23.5% of the original switches where as TLC has 30.1%.

We observe a similar trend when comparing the switch ratios for the Cost Equals Quadratic Colour cost function, where the switch ratio for TLC' gradually decreases as the size of the buffer increases, and it gets to around 50% of the switch ratio for TLC when the buffer size is significantly large as illustrated in Fig. 30. When comparing cost ratios, we observe that as the number of colours increases, TLC' performs marginally worse than TLC across all buffer sizes. Also, for 100 colours, we observe that TLC performs slightly better than TLC' for buffer sizes 1000 and 2000, where as TLC' performs slightly better than TLC for the other two buffer sizes (5000 and 10000). This indicates that TLC' is better suited for larger buffer sizes.

As in the previous two cost functions, we observe that TLC' achieves a better switch ratio for 100 colours across all buffer sizes. However, we also observe that while the switch ratio was always better for TLC', for the Random cost function, the switch ratio marginally increases for TLC' when the number of colours increases. This behaviour is consistent with all buffer sizes. As to cost ratio comparison, we observe that TLC' has a slightly higher cost ratio across all buffer size and number of colours combinations making TLC the better suited algorithm for this scenario.

For the Colour Difference cost function, we observe that the switch ratio is slightly better for 100 colours and as the number of colours increases, both have almost identical performance. But when cost ratios are compared, TLC outperforms TLC' by a very large margin as the cost ratio for TLC' for this cost function is very high compared to TLC.

Both TLC and TLC' have almost identical switch and cost ratios for the uniform

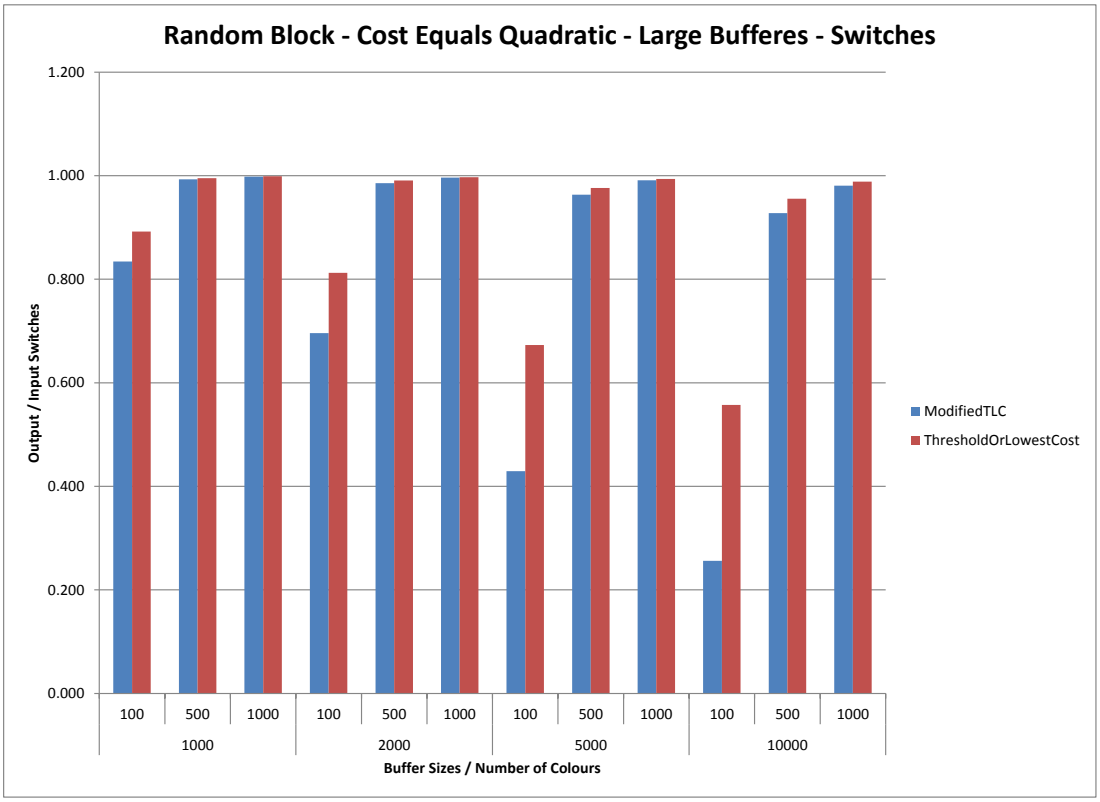

Figure 30: TLC' has a significantly better switch ratio

cost function making either of them suitable for these scenarios.

Sequential Block Sequences

For the Cost Equals Colour cost function, we observe that TLC' has a better switct ratio than TLC when the buffer size is at least 5 times the number of colours ($k = 5 \times C$). For other cases, we observe that TLC has a better switch ratio than TLC'. A similar result is obtained when we compare the cost ratios of the two algorithms, in this case we observe that TLC' has a better cost ratio than TLC when the buffer size is at least 10 times the number of colours ($k = 10 \times C$). For all other cases TLC

does better than TLC'.

We see a similar trend with the Cost Equals Quadratic Colour cost function where TLC' has a significantly better switch ratio than TLC when the buffer size is 5 times the number of colours ($k = 5 \times C$).However, when we compare the cost ratios we observe that unlike the Cost Equals Colour cost function, TLC' has a better cost ratio than TLC when the buffer size is at least 5 times the number of colours ($k >= 5 \times C$). This is the same inequality as observed for the switch ratios.

Our observations for the Random cost function are inline with those for the Cost Equals Colour cost function. We find that Random cost has the exact same inequalities for buffer sizes and number of colours for which TLC' achieves a better performance than TLC, which is $k \geq 10 \times C$ for switches and $k \geq 5 \times C$ for costs. This result is presented in Fig. 31.

The switch ratio for the Colour Difference cost function follows the same inequality as the switch ratio for the other cost functions, where TLC' has a better switch ratio than TLC when the buffer size is at least 5 times the number of colours. But when comparing cost ratios we observe that TLC' does not have a good performance across all buffer size and number of colours combinations. This is inline with the results obtained for other cost functions.

As seen with the other input sequence types, both TLC and TLC' have almost identical performance when the cost is Uniform.

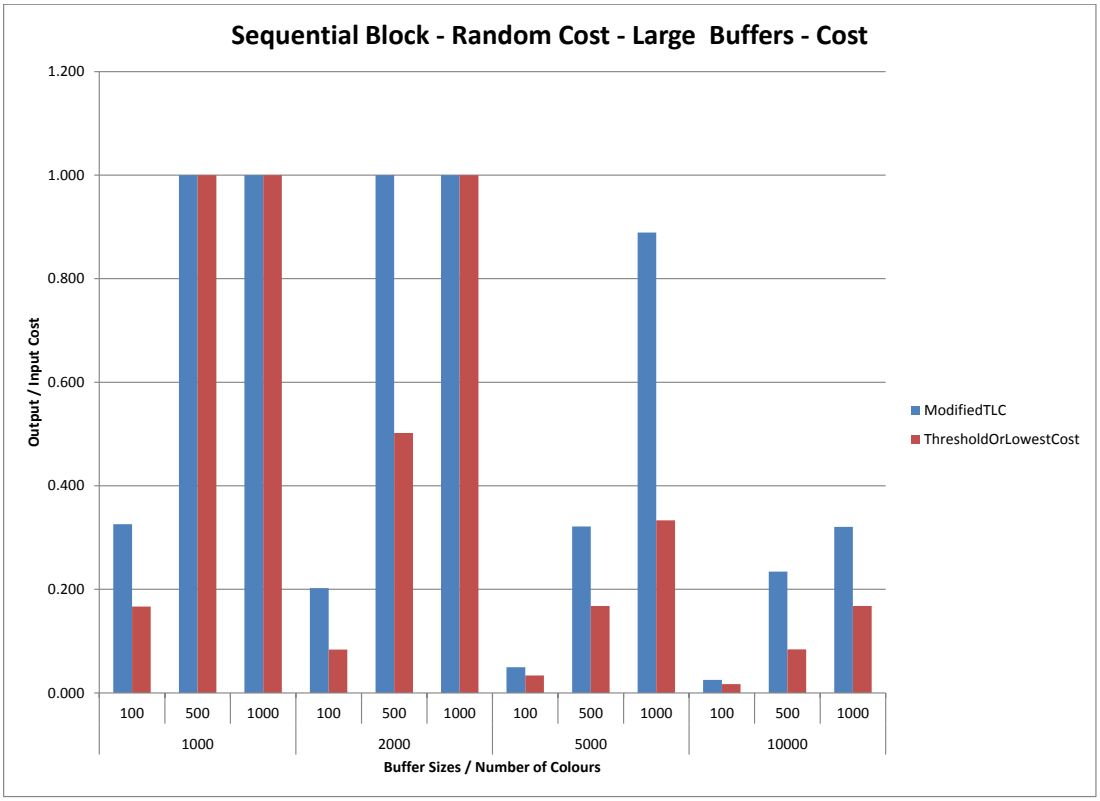

Figure 31: TLC' has a significantly better switch ratio when $k \geq 5 \times C$

Chapter 6

Conclusion and Future Work

6.1 Conclusions

6.1.1 Uniform Cost

Based on the results presented in Chapter **??**, it can be seen that Bounded Waste is the choice of algorithm for reordering when the input sequence alternates or is random. For input sequences that come in fixed sized blocks, BW fails to achieve a good performance for small buffers, in these cases Round Robin or Random Choice algorithms can be used and Bounded Waste can be used when the buffer sizes are significantly larger than the block size.

Recommend appropriate algorithms for graphics design, disk scheduling, file network routing and so on.

List of References

[1] H. Räcke, C. Sohler, and M. Westermann. "Online scheduling for sorting buffers." In "AlgorithmsESA 2002," pages 820–832. Springer (2002).

[2] J. Krokowski, H. Räcke, C. Sohler, and M. Westermann. "Reducing state changes with a pipeline buffer." In "VMV," page 217 (2004).

[3] M. Englert and M. Westermann. "Reordering buffer management for non-uniform cost models." In "Automata, Languages and Programming," pages 627–638. Springer (2005).

[4] J. S. Kohrt and K. Pruhs. *A constant approximation algorithm for sorting buffers*. Springer (2004).

[5] R. Bar-Yehuda and J. Laserson. "9-approximation algorithm for sorting buffers." In "Proceedings of the 3rd Workshop on Approximation and Online Algorithms," Citeseer (2005).

[6] N. Avigdor-Elgrabli and Y. Rabani. "A constant factor approximation algorithm for reordering buffer management." In "SODA," pages 973–984. SIAM (2013).

[7] N. Avigdor-Elgrabli and Y. Rabani. "An optimal randomized online algorithm for reordering buffer management." *arXiv preprint arXiv:1303.3386* (2013).

[8] M. Englert, H. Röglin, and M. Westermann. "Evaluation of online strategies for reordering buffers." *Journal of Experimental Algorithmics (JEA)* **14**, 3 (2009).

[9] A. Aboud. *Correlation clustering with penalties and approximating the reordering buffer management problem*. Ph.D. thesis, Masters thesis, The Technion-Israel Institute of Technology, Janurary 2008.[Technion TR archive]. 29 (2008).

[10] N. Avigdor-Elgrabli and Y. Rabani. "An improved competitive algorithm for reordering buffer management." In "Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on," pages 1–10. IEEE (2013).

[11] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. "Optimal online buffer scheduling for block devices." In "Proceedings of the 44th symposium on Theory of Computing," pages 589–598. ACM (2012).

[12] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. "Almost tight bounds for reordering buffer management." In "Proceedings of the 43rd annual ACM symposium on Theory of computing," pages 607–616. ACM (2011).

[13] Y. Azar, M. Englert, I. Gamzu, and E. Kidron. "Generalized reordering buffer management." In "LIPIcs-Leibniz International Proceedings in Informatics," volume 25. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2014).

[14] S. Im and B. Moseley. "New approximations for reordering buffer management." In "SODA," pages 1093–1111. SIAM (2014).

[15] A. Adamaszek, M. P. Renault, A. Rosén, and R. van Stee. "Reordering buffer management with advice." WAOA (2013).